Міністерство надзвичайних ситуацій України
Львівський державний університет безпеки життєдіяльності

Юрій ГРИЦЮК,
Тарас РАК

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ МОВОЮ С++

Навчальний посібник

Львів
Вид-во ЛДУ БЖД
2011

ББК 32.97я73
Г85
УДК 681.3.07

Автори
Ю.І. ГРИЦЮК – д-р техн. наук, доцент, завідувач кафедри управління інформаційною безпекою Львівського ДУ БЖД;
Т.Є. РАК – канд. техн. наук, доцент, підполковник сл. цив. захисту, начальник інституту цивільного захисту Львівського ДУ БЖД

Рецензенти:
А.Д. КУЗИК – канд. фіз.-мат. наук, доцент, доцент кафедри фундаментальних дисциплін Львівського ДУ БЖД (м. Львів);
В.М. СЕНЬКІВСЬКИЙ – д-р техн. наук, професор, завідувач кафедри електронних видань Української академії друкарства (м. Львів),
Д.Д. ПЕЛЕШКО – д-р техн. наук, доцент, професор кафедри автоматизованих систем управління НУ "Львівська політехніка" (м. Львів)

Відповідальний редактор В.В. ДУДОК

Затверджено до друку Вченою радою Львівського державного університету безпеки життєдіяльності (прот. № 8 від 19.05.2011 р.)

ISBN 978-966-3466-86-3

© Ю.І. Грицюк, 2011
© Т.Є. Рак, 2011
© Вид-во ЛДУ БЖД, 2011

ЗМІСТ

ПЕРЕДМОВА ... 9

ВСТУП ... 11

Розділ 1. ОСНОВНІ ОСОБЛИВОСТІ РОЗРОБЛЕННЯ ОБ'ЄКТНО-ОРІЄНТОВАНИХ ПРОГРАМ МОВОЮ С++ ... 16

1.1. Потреба використання об'єктно-орієнтованого програмування ... 16
1.2. Поняття про об'єктно-орієнтований підхід до розроблення складних програм ... 19
1.3. Основні компоненти об'єктно-орієнтованої мови програмування ... 22
1.4. Співвідношення між мовами програмування С і С++ ... 26
1.5. Поняття про універсальну мову моделювання ... 27

Розділ 2. КЛАСИ – ОСНОВА ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ ... 29

2.1. Базові поняття класу ... 29
2.2. Поняття про конструктори і деструктори ... 33
2.3. Особливості реалізації механізму доступу до членів класу ... 40
2.4. Класи і структури – споріднені типи ... 42
2.5. Об'єднання та класи – споріднені типи ... 45
2.6. Поняття про вбудовані функції ... 46
2.7. Особливості організації масивів об'єктів ... 49
2.8. Особливості використання покажчиків на об'єкти ... 52

Розділ 3. ОРГАНІЗАЦІЯ КЛАСІВ І ОСОБЛИВОСТІ РОБОТИ З ОБ'ЄКТАМИ ... 55

3.1. Поняття про функції-"друзі" класу ... 55
3.2. Особливості перевизначення конструкторів ... 59
3.3. Особливості механізму динамічної ініціалізації конструктора ... 61
3.4. Особливості механізму присвоєння об'єктів ... 64
3.5. Особливості механізму передачі об'єктів функціям ... 65
3.5.1. Конструктори, деструктори і передача об'єктів ... 66
3.5.2. Потенційні проблеми, які виникають при передачі об'єктів ... 68
3.6. Особливості механізму повернення об'єктів функціями ... 70
3.7. Механізми створення та використання конструктора копії ... 73
3.7.1. Використання конструктора копії для ініціалізації одного об'єкта іншим ... 74
3.7.2. Механізм використання конструктора копії для передачі об'єкта функції ... 75
3.7.3. Механізм використання конструктора копії при поверненні функцією об'єкта ... 77
3.7.4. Конструктори копії та їх альтернативи ... 78
3.8. Поняття про ключове слово this ... 78

Розділ 4. ОСОБЛИВОСТІ МЕХАНІЗМУ ПЕРЕВИЗНАЧЕННЯ ОПЕРАТОРІВ ... 80

4.1. Механізми перевизначення операторів з використанням функцій-членів класу 80
4.1.2. Перевизначення бінарних операторів додавання "+" і присвоєння "=" 81
4.1.2. Перевизначення унарних операторів інкремента "++" та декремента "--" ... 84
4.1.3. Особливості реалізації механізму перевизначення операторів..................90
4.2. Механізми перевизначення операторів з використанням функцій-не членів класу 91
4.2.1. Використання функцій-"друзів" класу для перевизначення бінарних операторів 91
4.2.2. Використання функцій-"друзів" класу для перевизначення унарних операторів 96
4.2.3. Перевизначення операторів відношення та логічних операторів 99
4.3. Особливості реалізації оператора присвоєння.. 100
4.4. Механізми перевизначення оператора індексації елементів масиву "[]" .. 104
4.5. Механізми перевизначення оператора виклику функцій "()" 108
4.6. Механізми перевизначення рядкових операторів.................................. 109
4.6.1. Конкатенація та присвоєння класу рядків з рядками класу 110
4.6.2. Конкатенація та присвоєння класу рядків з рядками, що закінчуються нульовим символом 111
Розділ 5. ОРГАНІЗАЦІЯ МЕХАНІЗМІВ УСПАДКУВАННЯ В КЛАСАХ.. 116
5.1. Поняття про успадкування в класах 116
5.2. Управління механізмом доступу до членів базового класу.................... 119
5.3. Механізми використання захищених членів класу 122
5.3.1. Використання специфікатора доступу protected для надання членам класу статусу захищеності 123
5.3.2. Використання специфікатора доступу protected для успадкування базового класу 127
5.3.3. Узагальнення інформації про використання специфікаторів доступу public, protected і private 129
5.4. Механізми успадкування декількох базових класів............................... 129
5.5. Особливості використання конструкторів і деструкторів при реалізації механізму успадкування 130
5.5.1. Послідовність виконання конструкторів і деструкторів..........................130
5.5.2. Передача параметрів конструкторам базового класу..............................134
5.6. Повернення успадкованим членам класу початкової специфікації доступу.. 138
5.7. Поняття про віртуальні базові класи 140
Розділ 6. ПОНЯТТЯ ПРО ВІРТУАЛЬНІ ФУНКЦІЇ ТА ПОЛІМОРФІЗМ.. 145
6.1. Покажчики на похідні типи – підтримка динамічного поліморфізму .. 145
6.2. Механізми реалізації віртуальних функцій.. 148
6.2.1. Поняття про віртуальні функції 148
6.2.2. Успадкування віртуальних функцій 151
6.2.3. Потреба у застосуванні віртуальних функцій..153

6.2.4. Приклад застосування віртуальних функцій ... 154
6.2.5. Поняття про суто віртуальні функції та абстрактні класи ... 158
6.2.6. Порівняння механізму раннього зв'язування з пізнім ... 160
6.2.7. Поняття про поліморфізм і пуризм ... 161

Розділ 7. РОБОТА З ШАБЛОННИМИ ФУНКЦІЯМИ ТА КЛАСАМИ .162

7.1. Поняття про узагальнені функції ... 162
7.1.1. Механізм реалізації шаблонної функції з одним узагальненим типом ... 163
7.1.2. Безпосередньо задане перевизначення узагальненої функції ... 165
7.1.3. Шаблонна функція з двома узагальненими типами ... 167
7.1.4. Механізм перевизначення специфікації шаблону функції ... 168
7.1.5. Використання стандартних параметрів у шаблонних функціях ... 168
7.1.6. Обмеження, які застосовуються при використанні узагальнених функцій ... 169
7.1.7. Приклад створення узагальненої функції abs() ... 170

7.2. Поняття про узагальнені класи ... 171
7.2.1. Створення класу з одним узагальненим типом даних ... 171
7.2.2. Створення класу з двома узагальненими типами даних ... 174
7.2.3. Приклад створення узагальненого класу для організації безпечного масиву ... 175
7.2.4. Використання в узагальнених класах аргументів, що не є узагальненими типами ... 176
7.2.5. Використання в шаблонних класах аргументів за замовчуванням ... 178
7.2.6. Механізм реалізації безпосередньо заданої спеціалізації класів ... 180

Розділ 8. МЕХАНІЗМИ ОБРОБЛЕННЯ ВИНЯТКОВИХ СИТУАЦІЙ ..182

8.1. Основні особливості оброблення виняткових ситуацій ... 182
8.1.1. Системні засоби оброблення винятків ... 182
8.1.3. Перехоплення винятків класового типу ... 188
8.1.4. Використання декількох catch-настанов ... 189

8.2. Варіанти оброблення винятків ... 191
8.2.1. Перехоплення всіх винятків ... 191
8.2.2. Накладання обмежень на тип винятків, які генеруються функціями ... 193
8.2.3. Повторне генерування винятку ... 194

8.3. Оброблення винятків, згенерованих оператором new ... 195

8.4. Механізми перевизначення операторів new і delete ... 197

Розділ 9. ОРГАНІЗАЦІЯ С++-СИСТЕМИ ВВЕДЕННЯ-ВИВЕДЕННЯ ПОТОКОВОЇ ІНФОРМАЦІЇ ... 203

9.1. Порівняння С- та С++-систем введення-виведення ... 203

9.2. Поняття "потоків" у мові програмування С++ ... 204
9.2.1. Файлові С++-потоки ... 204
9.2.2. Вбудовані С++-потоки ... 205
9.2.3. Класи потоків ... 205

9.3. Особливості механізмів перевизначення операторів введення-виведення даних ... 207
9.3.1. Створення перевизначених операторів виведення даних ... 207

9.3.2. Використання функцій-"друзів" класу для перевизначення операторів введення даних 209
9.3.3. Створення перевизначених операторів введення даних 211
9.3.4. Порівняння С- і С++-систем введення-виведення 213

9.4. Форматне введення-виведення даних 213

9.4.1. Форматування даних з використанням функцій-членів класу ios 214
9.4.2. Встановлення ширини поля, точності значення та символів заповнення 217
9.4.3. Використання маніпуляторів введення-виведення даних 218
9.4.4. Створення власних маніпуляторних функцій 221

9.5. Організація файлового введення-виведення даних 223

9.5.1. Відкриття та закриття файлу 223
9.5.2. Зчитування та запис текстових файлів 225
9.5.3. Неформатне введення-виведення даних у двійковому режимі 227
9.5.4. Зчитування та записування у файл блоків даних 229
9.5.5. Використання функції eof() для виявлення кінця файлу 230
9.5.6. Застосування С++ файлової системи для порівняння файлів 231
9.5.7. Використання інших функцій для двійкового введення-виведення даних 232
9.5.8. Перевірка статусу введення-виведення даних 235

9.6. Використання файлів довільного доступу 236

9.6.1. Функції довільного доступу 236
9.6.2. Приклади використання довільного доступу до вмісту файлу 237

9.7. Використання перевизначених операторів введення-виведення даних при роботі з файлами 238

Розділ 10. ДИНАМІЧНА ІДЕНТИФІКАЦІЯ ТИПІВ І ОПЕРАТОРИ ПРИВЕДЕННЯ ТИПУ 240

10.1. Динамічна ідентифікація типів 240

10.1.1. Отримання типу об'єкта у процесі виконання програми 240
10.1.2. Приклад RTTI-застосування 244
10.1.3. Застосування оператора typeid до шаблонних класів 246

10.2. Поняття про оператори приведення типів 249

10.2.1. Оператор приведення поліморфних типів dynamic_cast 250
10.2.2. Оператор перевизначення модифікаторів const_cast 255
10.2.3. Оператор неполіморфного приведення типів static_cast 256
10.2.4. Оператор перетворення типу reinterpret_cast 256
10.2.5. Порівняння звичайної операції приведення типів з новими чотирма cast-операторами 257

Розділ 11. ПОНЯТТЯ ПРО ПРОСТОРИ ІМЕН ТА ІНШІ ЕФЕКТИВНІ ПРОГРАМНІ ЗАСОБИ 258

11.1. Особливості організації простору імен 258

11.1.1. Поняття про простори імен 259
11.1.2. Застосування настанови using 262
11.1.3. Неіменовані простори імен 264
11.1.4. Застосування простору імен std 265

11.2. Застосування покажчиків на функції 267

11.2.1. Передача покажчиком на функцію її адреси іншій функції 268

11.2.2. Пошук адреси перевизначеної функції ... 270

11.3. Поняття про статичні члени-даних класу ... 272

11.5. Застосування до функцій-членів класу модифікаторів const і mutable ... 274

11.6. Використання explicit-конструкторів ... 276

11.7. Синтаксис механізму ініціалізації членів-даних класу ... 278

11.8. Використання ключового слова asm ... 280

11.9. Специфікатор компонування функцій ... 281

11.10. Оператори вказання на члени класу ".*" і "->" ... 282

11.11. Створення функцій перетворення типів ... 284

Розділ 12. ВВЕДЕННЯ В СТАНДАРТНУ БІБЛІОТЕКУ ШАБЛОНІВ ... 287

12.1. Огляд стандартної бібліотеки шаблонів ... 287

12.2. Поняття про контейнерні класи ... 290

12.3. Механізми роботи з векторами ... 291

12.3.1. Доступ до елементів вектора за допомогою ітератора ... 296

12.3.2. Вставлення та видалення елементів з вектора ... 297

12.3.3. Збереження у векторі об'єктів класу ... 298

12.3.4. Доцільність використання ітераторів ... 300

12.4. Механізми роботи зі списками ... 301

12.4.1. Використання базових операцій для роботи зі списком ... 302

12.4.2. Особливості сортування списку ... 306

12.4.3. Об'єднання одного списку з іншим ... 307

12.4.4. Зберігання у списку об'єктів класу ... 308

12.5. Поняття про відображення – асоціативний контейнер ... 310

12.5.1. Робота з відображеннями ... 313

12.5.2. Зберігання у відображенні об'єктів класу ... 314

12.6. Алгоритми оброблення контейнерних даних ... 316

12.6.1. Обчислення кількості елементів ... 318

12.6.2. Видалення та заміна елементів ... 320

12.6.3. Реверсування послідовності ... 321

12.6.4. Перетворення послідовності ... 322

12.7. Особливості використання об'єктів класу string ... 324

12.7.1. Клас string – частина С++-бібліотеки ... 324

12.7.2. Огляд функцій-членів класу string ... 328

12.7.3. Зберігання рядків у інших контейнерах ... 332

Розділ 13. ОСОБЛИВОСТІ РОБОТИ ПРЕПРОЦЕСОРА С++ ... 333

13.1. Поняття про директиви препроцесора С++ ... 333

13.2. Директиви умовного компілювання ... 338

13.3. Оператори препроцесора "#" і "##" ... 343

13.4. Зарезервовані макроімена ... 344

Розділ 14. ФОРМАЛІЗАЦІЯ ПРОЦЕСУ РОЗРОБЛЕННЯ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ... 346

14.1. Удосконалення процесу розроблення сучасного програмного забезпечення ... 346
14.2. Моделювання варіантів використання ... 348
14.3. Предметна область програмування ... 352
14.4. Програма landlord: етап удосконалення ... 354
14.5. Перехід від варіантів використання до класів ... 358
14.6. Кроки написання коду програми ... 364
Додаток А. ОСОБЛИВОСТІ РОЗРОБЛЕННЯ КОНСОЛЬНИХ ПРОГРАМ У СЕРЕДОВИЩІ BORLAND C++ BUILDER ... 382
А.1. Основні правила роботи у середовищі C++ Builder ... 382
А.2. Компілювання, зв'язування та запуск консольних програм на виконання ... 385
А.3. Додавання заголовного файлу до консольного проекту ... 386
А.4. Проекти з декількома початковими файлами ... 387
А.5. Відлагодження коду програми ... 388
Додаток Б. ОСОБЛИВОСТІ РОЗРОБЛЕННЯ КОНСОЛЬНИХ ПРОГРАМ У СЕРЕДОВИЩІ MICROSOFT VISUAL C++ ... 390
Б.1. Елементи вікна середовища MVC++ ... 390
Б.2. Робота з однофайловими консольними програмами ... 390
Б.3. Робота з багатофайловими консольними програмами ... 392
Б.4. Програми з консольною графікою ... 394
Б.5. Відлагодження програм ... 395
Додаток Д. .NET-РОЗШИРЕНИЯ ДЛЯ С++ ... 397
Д.1. Ключові слова .NET-середовища ... 398
Д.2. Розширення препроцесора ... 399
ЛІТЕРАТУРА ... 401

ПЕРЕДМОВА

Протягом останніх декількох десятиліть комп'ютерні технології розвивались вражаючими темпами. Мови програмування також потерпіли значної еволюції. Поява більш потужних комп'ютерів дала життя більш об'ємним і складним програмам, які, водночас, висвітлювали нові проблеми в області керування програмами, а також їх подальшому супроводу. Ще у 70-ті роки такі мови програмування, як С та Pascal, допомогли людству увійти в епоху структурного програмування, яке на той час відчайдушно потребувало наведення у цій області деякого порядку. Мова програмування С дала в розпорядження програмістів інструменти, необхідні для реалізації структурного програмування, а також забезпечила створення компактних, швидко працюючих програм і можливість адресації апаратних ресурсів (наприклад, можливість керування портами зв'язку і накопичувачами на магнітних носіях). Ці властивості допомогли мові С у 80-ті роки дещо домінувати над іншими мовами програмування. В той же час з'явилася і нова технологія створення та удосконалення програм – об'єктно-орієнтоване програмування (ООП), втіленням якого стала спочатку мова С++.

Об'єктно-орієнтоване програмування – одна з парадигм програмування[1], яка розглядає програму як множину "об'єктів", що взаємодіють між собою. В ній використано декілька технологій від попередніх парадигм, зокрема успадкування, модульність, поліморфізм та інкапсуляцію. Незважаючи на те, що ця парадигма з'явилась ще в 1960-тих роках, вона не мала широкого застосування до 1990-тих. Сьогодні багато мов програмування (зокрема, Java, C#, C++, Python, PHP, Ruby та Objective-C, ActionScript 3) підтримують ООП.

Об'єктно-орієнтоване програмування сягає своїм корінням до створення мови програмування Симула в 1960-тих роках, одночасно з посиленням дискусій про кризу програмного забезпечення. Разом із тим, як ускладнювалось апаратне та програмне забезпечення, було дуже важко зберегти якість програм. Об'єктно-орієнтоване програмування частково розв'язало цю проблему шляхом наголошення на модульності програми. На відміну від традиційних поглядів, коли програму розглядали як набір підпрограм, або як перелік нас-

---

[1] *Парадигма програмування* – основні принципи програмування (не плутати з розробленням програм), або, парадигмне програмування. Парадигма програмування надає (та визначає) те, як програміст розглядає роботу програми. Наприклад, в об'єктно-орієнтованому програмуванні, програміст розглядає програму як множину взаємодіючих між собою об'єктів, водночас як у функційному програмуванні програму можна представити як обчислення послідовності функцій без станів.

Основні парадигми програмування: процедурне програмування (англ. Procedural programming); модульне програмування (англ. Modular programming); об'єктно-орієнтоване програмування (англ. Object-oriented programming); функційне програмування (англ. Functional programming); імперативне програмування (англ. Imperative programming); декларативне програмування (англ. Declarative programming); прототипне програмування (англ. Prototype-based programming); аспектно-орієнтоване програмування (англ. Aspect-oriented programming); предметно-орієнтоване програмування (англ. Subject-oriented programming); функціонально-орієнтоване програмування (англ. Feature-oriented programming).

танов комп'ютеру, ООП програми можна вважати сукупністю об'єктів. Відповідно до парадигми об'єктно-орієнтованого програмування, кожний об'єкт здатний отримувати повідомлення, обробляти дані, та надсилати повідомлення іншим об'єктам. Кожен об'єкт – своєрідний незалежний автомат з окремим призначенням та відповідальністю.

У цьому навчальному посібнику огляд особливостей об'єктно-орієнтованого програмування починається з поняття "об'єкт". Загалом *об'єкт* – це абстрактна суть, наділена характеристиками об'єктів, що оточує нас в реальному світі. Створення об'єктів і маніпулювання ними – це зовсім не привілей мови програмування С++, а швидше результат технології програмування, що втілює в кодових конструкціях опису об'єктів і операції над ними. Кожен об'єкт програми, як і будь-який реальний об'єкт, відрізняється власними атрибутами і характерною поведінкою. Об'єкти можна класифікувати за різними категоріями: наприклад, цифровий наручний годинник "Orient" належить до класу годинника. Програмна реалізація годинника входить як стандартний додаток до складу операційної системи будь-якого комп'ютера.

Кожен клас займає певне місце в ієрархії класів, наприклад, всі годинники належать класу приладів вимірювання часу (вищому в ієрархії), а клас годинника сам містить множину похідних варіацій на ту ж тему. Таким чином, будь-який клас визначає деяку категорію об'єктів, а всякий об'єкт є екземпляр деякого класу. Маючи цілісне уявлення про об'єкти і класи, далі розглядаються шаблони і як вони реалізуються стандартом мови С++. У процесі викладення матеріалу опис тих чи інших положень мови С++ продемонстровано невеликими навчальними програмами, які студенту нескладно скопіювати і самостійно випробувати на власному комп'ютері та проаналізувати отримані результати.

Матеріал цього навчального посібника повністю відповідає робочій навчальній програмі такої дисципліни як "Технології програмування", яка викладається у Львівському державному університеті безпеки життєдіяльності для курсантів і студентів напрямку підготовки "Управління інформаційною безпекою" та вивчають комп'ютерні методи захисту інформації.

Ми (автори, рецензенти і літературний редактор) приклали багато зусиль, щоби зробити матеріал цього навчального посібника конкретним, простим для розуміння, засвоєння та, що найважливіше, цікавим. Наша мета полягала у тому, щоби, внаслідок вивчення того чи іншого матеріалу, студенти могли створювати свої власні навчальні програми і, водночас, знаходити в цьому як користь, так і задоволення.

Автори з вдячністю приймуть будь-які конструктивні зауваження стосовно викладеного у посібнику матеріалу, які можна надсилати за адресою:

Кафедра управління інформаційною безпекою

Львівський ДУ БЖД, вул. Клепарівська 35, м. Львів, 79007

***Тел. моб.**: 067-944-11-15. **E-mail**: grycіuk.yura@gmail.com*

ВСТУП

Основне завдання цього навчального посібника – навчити студентів розробляти програми мовою C++ з використанням технології об'єктно-орієнтованого програмування. Не секрет, що протягом декількох останніх десятиліть на ринку програмного забезпечення відбулися значні зміни, які є очевидними навіть для не професіонала. Зокрема, мова C++ стала універсальною мовою програмування[1] високого рівня з підтримкою декількох сучасних парадигм програмування: об'єктно-орієнтованої, узагальненої та процедурної. Тому ще одне завдання цього навчального посібника полягає в тому, щоб викласти основні концепції ООП мовою C++ у контексті розвитку сучасного програмного забезпечення.

Серед студентів часто побутує думка, згідно з якою мову програмування C++ важко вивчити. Насправді вона має багато спільного з такими мовами програмування як Pascal чи Visual Basic, за винятком декількох вдало реалізованих нових ідей. Забігаючи наперед, зазначимо, що ці ідеї є цікавими та захопливими, тому їх вивчення навряд чи виявиться для Вас нудним. Окрім цього, про зміст цих ідеї необхідно мати уявлення хоча би тому, що вони є не тільки основою сучасних технологій програмування та розроблення програмних продуктів, але й становлять значну частину "культури програмування".

Будемо сподіватися на те, що цей навчальний посібник допоможе студентам розібратися як зі специфікою об'єктно-орієнтованого підходу до розроблення програм мовою C++, так і у загальних концепціях розроблення програмних продуктів. Запропонований матеріал буде корисним як студентам, що вивчають програмування, так і професійним програмістам – не фахівцям з області ООП, а також усім тим, хто має інтерес до даної області знань.

Нові концепції розроблення програмних продуктів

Донедавна більшість професійних програмістів реалізовували свої програмні проекти з використанням технології структурного чи процедурного програмування. Із збільшенням обсягів оброблюваної інформації, зростанням вимог користувачів до програмних продуктів почали значно зростати розміри кодів розроблюваних програм. Щоразу перед початком реалізації нового програмного продукту програмістам ставало дедалі очевидніше, що такі підходи є невдалими. Проблема полягала, передусім, у непропорційному зростанні складності процесу розроблення самих кодів програм порівняно зі зростанням вимог до програмного продукту.

Напевно буде справедливим твердження про те, що досконалі та вишукані програми, без перебільшення, належать до найскладніших творінь людини. Проте, зважаючи на надлишкову складність, такі програми нерідко міс-

[1] http://uk.wikipedia.org/wiki/C%2B%2B

тять помилки – добре, якщо не значні. Бо інколи серйозні помилки у програмному забезпеченні потенційно спричиняють матеріальні збитки (наприклад, помилки у бухгалтерських чи банківських розрахунках), а іноді і загрожують життю багатьох людей (наприклад, під час керування атомними реакторами, авіаперельотами чи космічним кораблем).

Внаслідок довготривалої боротьби з проблемою складності процесу написання кодів програм були вироблені нові концепції розроблення програмних продуктів, а саме:

- об'єктно-орієнтоване програмування (ООП);
- уніфікована мова моделювання програм (UML);
- спеціалізовані засоби розроблення сучасного програмного забезпечення.

У декого може виникнути слушне запитання: чому об'єктно-орієнтований підхід до програмування став пріоритетним під час розроблення більшості сучасних програмних продуктів? Відповідь може бути такою: ООП пропонує новий потужний механізм вирішення проблеми процесу написання складних кодів програм. А саме: замість того, щоби розглядати програму як набір послідовно виконуваних настанов, у ООП код програми представляється у вигляді сукупності деяких об'єктів, що мають схожі властивості і набори відповідних дій, які можна з ними проводити. Можливо, все те, про що тут буде сказано, здаватиметься Вам спочатку незрозумілим. Але це тільки доти, доки Ви не почнете детально вивчати відповідні розділи ООП з цього навчального посібника. З поступовим засвоєнням відповідного матеріалу Ви не раз переконаєтеся в тому, що застосування об'єктно-орієнтованого підходу до процесу програмування робить коди програм більш зрозумілими, надійними і простими для удосконалення.

Уніфікована мова моделювання (UML) – це не що інше, як графічне подання алгоритму розв'язання задачі з застосуванням об'єктно-орієнтованого підходу. Згідно з визначенням, UML – це графічна мова, що містить безліч різних діаграм, які допомагають фахівцям з системного аналізу створювати алгоритми, а програмістам – з'ясовувати принципи роботи коду програми. UML є потужним інструментом, який дає змогу зробити процес програмування більш легким і ефективним. Кожен новий засіб реалізації основних елементів UML викладено у тому місці, де він стає корисним для ілюстрації основних концепцій ООП. Наприклад, діаграми класів UML вивчаються одночасно з взаємодією різних класів, а узагальнення – у зв'язку з поняттям успадкування. Таким чином, у читача з'являється можливість, не докладаючи зайвих зусиль, засвоїти основні концепції мови UML, яка одночасно сприятиме більш ефективному засвоєнню ООП мовою С++.

Одним із спеціалізованих засобів розроблення сучасного програмного забезпечення є система візуального ООП під назвою С++ Builder 6 і попередні її версії. Система призначена для розроблення завершених додатків для Windiws найбільш різноманітної спрямованості, від суто обчислювальних і логічних – до графічних і мультимедійних. Наголосимо, що у цьому посібнику, окрім вивчення процедурного та об'єктно-орієнтованого програмування мовою С++, зовсім не приділено уваги спеціалізованим засобам розроблення

сучасного програмного забезпечення, оскільки цей матеріал належить до іншої дисципліни.

Сучасні об'єктно-орієнтовані мови програмування

Зі всіх об'єктно-орієнтованих мов програмування найчастіше використовується мова C++. Мова програмування C++ отримала набагато більшу популярність, ніж такі мови як Pascal, Visual Basic чи C, бо вона стала потужним інструментом для розроблення складного програмного забезпечення. У синтаксичному плані мови C і C++ дуже схожі між собою. Понад це, мова програмування C++ є надбудовою мови C. Завдяки цьому відпала потреба перед вивченням мови програмування C++ вивчати мову C. Ті студенти, які мають навики роботи мовою C, можуть знехтувати частиною матеріалу, викладеного на початкову цього посібника, проте решта інформації буде для них новою і корисною для засвоєння.

Окрім цього, ООП мовою C+ містить декілька нових концепцій, які можуть бути незнайомі тим, хто практикує програмування такими традиційними мовами, як Pascal, Visual Basic і C. До цих концепцій належать *класи, успадкування* і *поліморфізм*, які становлять основу об'єктно-орієнтованого підходу до розроблення сучасних програмних продуктів. Проте, вивчаючи специфічні особливості об'єктно-орієнтованих мов, необізнаним програмістам досить легко піти манівцями від цих понять. Багато сучасних авторів разом з описом різних мов програмування вдаються до детального опису можливостей тієї чи іншої мови, не приділяючи зовсім уваги тому, чому ці можливості існують. У цьому навчальному посібнику нюанси ООП мовою C++ розглянуто у безпосередньому його зв'язку з базовими концепціями сучасних технологій програмування та розроблення програмних продуктів.

Мова програмування Java, що є останньою розробкою в області об'єктно-орієнтованих мов, позбавлена таких складових як покажчики, шаблони і множинне успадкування, що зробило її менш могутньою і гнучкою порівняно з мовою C++. З огляду на те, що синтаксис мов Java і C++ дуже схожі, тому отримані студентами знання стосовно мови програмування C++ з успіхом можуть бути використані під час самостійного вивчення мови Java. Деякі інші об'єктно-орієнтовані мови, серед яких є і мова C#, також успішно розвиваються, проте їх розповсюдження значною мірою поступається мові C++.

Донедавна мова C++ розвивалася поза рамками відповідних стандартів, тобто кожен виробник компіляторів по-своєму реалізовував окремі нюанси мови. Проте комітет зі стандарту мови C++ організації ANSI/ISO[1] розробив документ, який на сьогодні відомий під назвою *"Стандарт C++"*. Згідно з цим документом, мова програмування C++ містить ряд уточнень і доповнень, які стосувалися раніше мов C і C++, а також багато додаткових можливостей, наприклад, *стандартну бібліотеку шаблонів* (*бібліотека STL*). Ця бібліотека надає програмісту доступ до шаблонних класів і функцій загального призначення, які реалізують багато популярних і часто використовуваних алгорит-

---

[1] ANSI є скороченням від англійської назви Американського Національного Інституту Стандартів, а ISO – від Міжнародної Організації Стандартів.

мів і структур даних. Наприклад, вона містить підтримку векторів, списків, черг і стеків, а також визначає різні функції, які забезпечують до них доступ.

Оскільки основну увагу в цьому навчальному посібнику сконцентровано на ООП мовою С++, то ми вилучили з детального розгляду ті засоби мови С, які рідко використовуються і не мають відношення до об'єктно-орієнтованого підходу. Усі розглянуті у цьому посібнику коди програм повністю задовольняють вимогам стандарту мови програмування С++ за незначними винятками, які зумовлені некоректною роботою компіляторів. Окремий розділ присвячено бібліотека STL (стандартній бібліотеці шаблонів), що є частиною стандарту мови програмування С++.

Базові знання, необхідні студентам для роботи з цим навчальним посібником

Матеріал, викладений у цьому навчальному посібнику, доступний навіть тим студентам, хто вивчає об'єктно-орієнтоване програмування "з нуля". Проте наявність досвіду програмування такими мовами, як Pascal чи С++, не знаходитиметься на заваді при засвоєнні матеріалу з цього посібника. Бажано також, щоб студенти, вивчаючи матеріал з цього навчального посібника, мали уявлення про основні операції Microsoft Windows, такі, як запуск програм на виконання, копіювання файлів тощо. Можливо, викладачам і іншим нашим читачам, які вже володіють мовою С++, буде цікаво детальніше ознайомитися з нашою концепцією висвітлення матеріалу, а також підходом, який ми пропонуємо під час вивчення ООП мовою С++.

У цьому посібнику вивчення процесу розроблення програм з використанням об'єктно-орієнтованого підходу починається з найпростіших прикладів, поступово ускладнених і закінчуючи повноцінними програмними комплексами. Інтенсивність подання нового матеріалу є такою, що у студента є можливість без зайвого поспіху його вивчити і, при потребі, повторити пройдений. Для полегшення процесу засвоєння нового матеріалу у кожному розділі цього посібника наведено достатню кількість наочних прикладів з програмною їх реалізацією, а також детальним аналізом їх програмних блоків.

Технічне і програмне забезпечення

Найбільш популярним середовищем для розроблення програм мовою С++ є продукт, який був запропонований спільно компаніями Microsoft і Borland, призначений для роботи під управлінням операційних систем Microsoft Windows. Наведені у цьому навчальному посібнику приклади кодів програм розроблено так, що вони підтримуються як компіляторами Microsoft, так і компіляторами Borland (детальнішу інформацію про компілятори можна знайти у відповідних документаціях до "Microsoft Visual C++" і до "Borland C++ Builder". Інші компілятори, розраховані на стандарт мови програмування С++, безпомилково сприйматимуть більшість кодів програм у тому вигляді, у якому їх наведено у цьому навчальному посібнику.

Для того, щоби будь-який з компіляторів міг працювати на Вашому комп'ютері, Вам потрібно забезпечити достатній вільний дисковий простір,

відповідний об'єм оперативної пам'яті та швидкодію процесора. Ці параметри можна дізнатися у настанові з експлуатації відповідних пристроїв.

Зверніть увагу на те, що у цьому навчальному посібнику містяться навчальні приклади програм, початкові коди яких є консольними, тобто такими, що виконуються в текстовому вікні компілятора або безпосередньо в управлінському вікні MS DOS. Це зроблено для того, щоби уникнути зайвих труднощів, які виникають під час роботи з повноцінними графічними додатками Windows.

Процес розроблення прикладного програмного забезпечення стає все більш і більш важливим аспектом сучасних технологій програмування. Шкода, але часто процес розроблення об'єктно-орієнтованої програми залишається для студентів загадкою. Це послугувало для нас приводом помістити у навчальний посібник спеціальний розділ, що стосується процесу розроблення сучасного програмного забезпечення, де основну увагу приділено реалізації концепції об'єктно-орієнтованому підходу (див. дод. А).

Орієнтовний порядок вивчення ООП

У цьому навчальному посібнику перший розділ присвячено основним особливостям розроблення об'єктно-орієнтованих програм мовою С++. Після цього ООП мовою С++ починається з безпосереднього вивчення механізму реалізації класів – фундаменту, на якому побудовано С++-підтримку ООП, а також ядро багатьох набагато складніших програмних засобів. Особливу увагу тут необхідно звернути на ті розділи, які стосуються роботи з класами, механізму перевизначення операторів, успадкування в класах, застосування віртуальних функцій та поліморфізму, шаблонів у класах, оброблення виняткових ситуацій, С++-системи введення-виведення потокової інформації тощо.

З самого початку висвітлення матеріалу у цьому навчальному посібнику використано С++-систему введення-виведення різних даних, але не дано детальних пояснень з механізму її реалізації. Оскільки С++-система введення-виведення побудовано на ієрархії класів, то її теорію і деталі реалізації неможливо засвоїти, не розглянувши спочатку механізм реалізації класів, успадкування в класах і оброблення виняткових ситуацій. На нашу думку, якраз після цього і настає момент для детального вивчення С++-засобів введення-виведення потокової інформації.

У самому кінці навчального посібника подано матеріал, який стосується формалізації процесу розроблення об'єктно-орієнтованого програмного забезпечення. Зокрема, висвітлюються такі основні питання, як удосконалення процесу розроблення програмного забезпечення, наводяться діаграми варіантів використання, досліджується предметна область програмування, наводиться код конкретної програми та етапи її удосконалення, проаналізовано перехід від варіантів використання до класів, а також подаються конкретні кроки написання коду програми.

Розділ 1. ОСНОВНІ ОСОБЛИВОСТІ РОЗРОБЛЕННЯ ОБ'ЄКТНО-ОРІЄНТОВАНИХ ПРОГРАМ МОВОЮ С++

Складність не в тому, щоби створити витвір мистецтва; складність в умінні зробити це.

Костянтин Бранкузі

Вивчивши цей розділ навчального посібника, студент отримає основні навики розроблення програм мовою С++, яка підтримує *об'єктно-орієнтоване програмування* (ООП). Проте багато хто зі студентів, маючи деякий досвід попереднього програмування, може задати такі запитання:

- для чого потрібно знати ООП?
- які переваги ООП мовою С++ перед такими традиційними мовами програмування, як Pascal, Visual Basic чи С++?
- що є основою ООП?
- який зміст термінів – *об'єкти* і *класи*?
- як пов'язані між собою мови С і С++?

Відповіді на ці запитання якраз і можна буде отримати у цьому розділі. Тут також розглядатимемо засоби ООП мовою С++, про які йтиметься у інших розділах цього навчального посібника. Не варто турбуватися про те, що матеріал, викладений саме у цьому розділі, видасться Вам занадто абстрактним. Всі механізми і ключові концепції ООП, які згадуються тут, детально описано в подальших розділах цього навчального посібника.

1.1. Потреба використання об'єктно-орієнтованого програмування

Розвиток об'єктно-орієнтованої технології створення сучасних програмних продуктів пов'язаний деякими обмеженими можливостями інших технологій програмування, які широко застосовувалися раніше. Щоб краще зрозуміти і оцінити значення ООП, необхідно з'ясувати, у чому ж полягають ці обмеження та як вони відображаються в традиційних мовах програмування.

Процедурні мови програмування. Мови програмування Pascal, Visual Basic, С чи інші схожі з ними мови належать до категорії *процедурних мов.* Кожен оператор такої мови дає вказівку комп'ютеру виконати певну дію чи їх послідовність, наприклад, прийняти дані від користувача, виконати з ними певні арифметичні операції чи логічні дії та вивести отриманий результат на екран чи принтер. Програми, написані такими процедурними мовами, складаються з послідовності певних настанов. Для невеликих програм не вимагається додаткової внутрішньої їх організації – внутрішньої *парадигми.* Програміст записує перелік настанов, а комп'ютер здійснює дії, які відповідають цим настановам.

Поділ програми на функції та модулі. Коли розмір програми зростає, то зміст виконуваних настанов стає надзвичайно громіздким і заплутаним. На сьогодні професійних програмістів, здатних запам'ятати більше 500 рядків коду програми, яку не розділено на дрібні логічні частини, є не так вже й багато. Застосування ж відповідних *функцій користувача* значно полегшує сприйняття коду програми під час його аналізу. Програмний код, побудований на основі структурного підходу, поділяється на відповідні функції, кожна з яких у ідеальному випадку здійснює певну завершену послідовність дій і має явно виражені зв'язки з іншими функціями коду програми.

Розвиток ідеї поділу коду програми на функції користувача призвів до того, що їх почали об'єднувати по декілька в *програмний модуль*, який можна записати окремим файлом. Однак навіть при такому підході зберігається структурний принцип: код програми поділяється на декілька структурних компонент, кожна з яких є набором відповідних настанов.

Поділ коду програми на функції та модулі є основою технології структурного програмування, яка протягом багатьох десятиліть, доки не було розроблено концепцію ООП, залишалася важливим способом організації кодів програм і популярною методикою розроблення програмного забезпечення.

Недоліки технології структурного програмування. У безперервному зростанні розміру програми, повсякчасному ускладнення її логіки чи виконуваних нею дій розробники програмних продуктів поступово почали виявляти недоліки технології структурного програмування. По-перше, існують обмежені можливості доступу функцій до глобальних даних. По-друге, поділ програми на глобальні дані та функції, які є основою технології структурного програмування, погано відображають "картину реального світу" – фізичну сутність технічного завдання.

Проаналізуємо ці недоліки на прикладі розроблення програми ведення складського обліку, наприклад, матеріальних цінностей. У такій програмі глобальними даними є записи в обліковій книзі. Різні функції отримуватимуть доступ до цих даних для виконання різних операцій: створення нового запису, виведення запису на екран, зміни полів наявного запису і т.д.

Неконтрольований доступ до даних. У структурній програмі, написаній, наприклад, мовою Pascal, існує два типи даних. *Локальні дані* знаходяться усередині будь-якої функції та призначені для використання тільки нею. Наприклад, у програмі ведення складського обліку функція, яка здійснює виведення запису на екран, може використовувати локальні дані для зберігання вартості деяких матеріалів, якщо відома їх закупівельна ціна та наявна кількість. Локальні дані функції недоступні нікому, окрім неї самої, і не можуть бути змінені іншими функціями.

Якщо існує потреба сумісного використання одних і тих самих даних декількома функціями, то ці дані мають бути оголошені як *глобальні.* Це, як правило, стосується тих даних програми, які є найважливішими. Прикладом тут може слугувати вже згадана вище ціна та кількість матеріалу. Будь-яка функція має доступ до глобальних даних (ми не розглядаємо випадок групу-

вання функцій у програмні модулі). Схему, яка ілюструє концепцію області видимості локальних і глобальних даних, наведено на рис. 1.1.

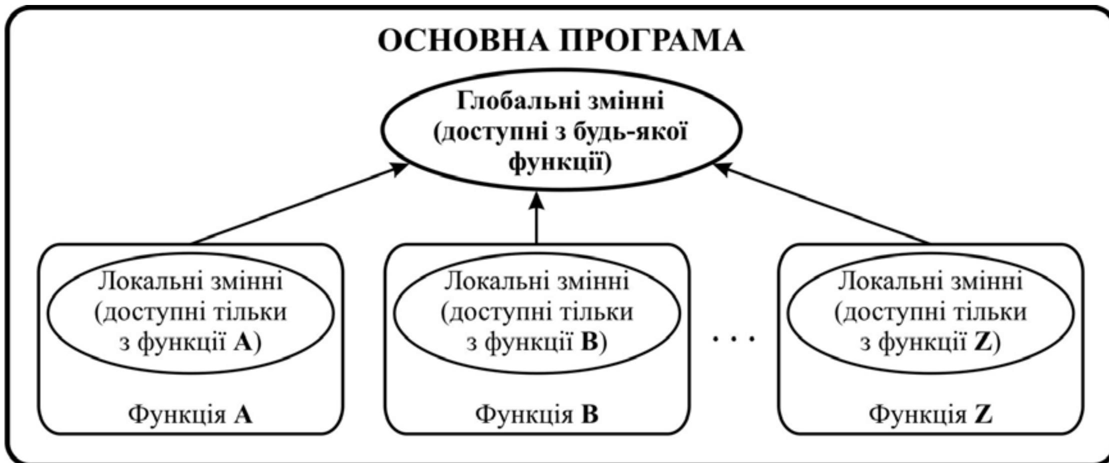

Рис. 1.1. Концепція області видимості глобальних і локальних даних

Великі за розміром програми зазвичай містять багато різних функцій і згрупованих глобальних даних. Проблема структурного підходу полягає в тому, що кількість можливих зв'язків між різними групами глобальних даних і використовуваними функціями може бути дуже великою, як це показано на рис. 1.2.

Велика кількість зв'язків між функціями і групами даних зазвичай породжує декілька додаткових проблем. По-перше, ускладнюється структура коду програми. По-друге, у код програми важко вносити нові зміни. Окрім цього, будь-яка зміна структури глобальних даних може вимагати коректування всіх функцій, які використовують ці дані. Наприклад, якщо розробник програми ведення складського обліку матеріальних цінностей вирішить зробити заміну деяких глобальних даних з 5-значного коду на 12-значний, то необхідно змінити відповідний тип даних з **short** на **long**. Це означає, що в усіх функціях, які оперують цими даними, потрібно внести зміни у локальні дані, тобто оголосити їх типом **long**.

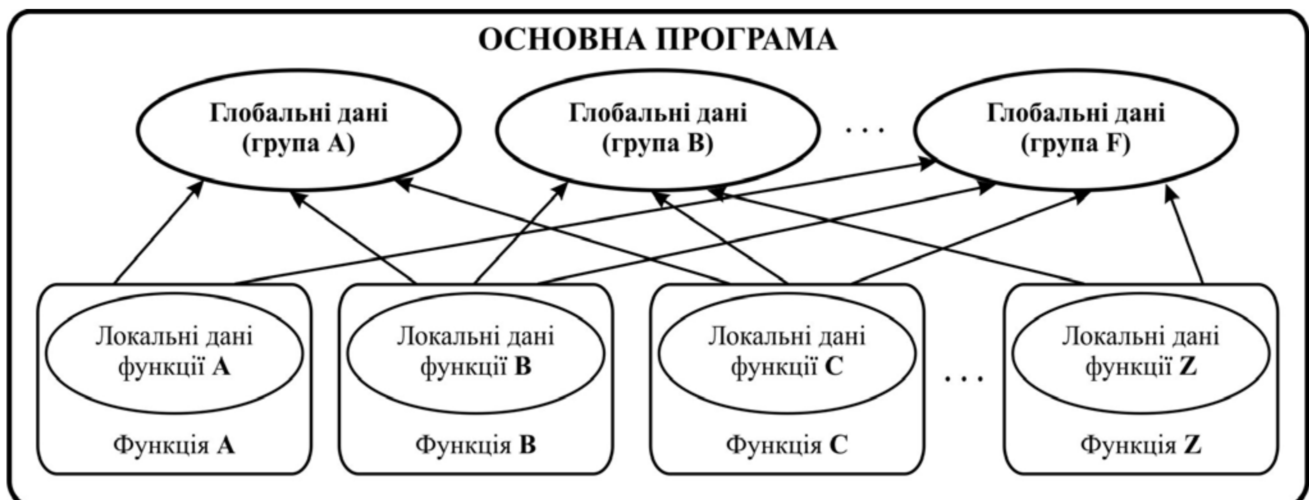

Рис. 1.2. Структурний підхід до встановлення зв'язків між глобальними даними і функціями програми

Коли зміни вносяться в глобальні дані великих програм, то не завжди можна швидко визначити, які функції при цьому необхідно скоректувати. На-

віть тоді, коли це вдається зробити оперативно, то згодом через значну кількість зв'язків між функціями і даними виправлені функції починають некоректно працювати з іншими глобальними даними. Таким чином, будь-яка зміна у коді програми призводить до виникнення негативних подальших дій і, як наслідок, появи відповідних проблем.

Відображення картини реального світу. Другий, набагато важливіший недолік структурного програмування полягає в тому, що відокремлення даних від функцій виявляється малопридатним для достовірного "відображення картини реального світу", тобто, адекватного відтворення фізичного змісту технічного завдання. Йдеться про те, що у реальному світі нам доводиться мати справу з фізичними об'єктами, такими, наприклад, як люди або машини. Ці об'єкти не можна віднести ні до даних, ні до функцій, оскільки реальні речі характеризує сукупність певних *властивостей* чи їх *поведінку.*

Прикладами ***властивостей*** (іноді їх називають характеристиками) для людей можуть бути колір очей або місце роботи; для машин – потужність двигуна і кількість дверей. Таким чином, властивості об'єктів рівносильні даним у програмах: вони набувають певні значення, наприклад *карий* – для кольору очей або 4 – для кількості дверей автомобіля.

Поведінка – це певна реакція фізичного об'єкта у відповідь на зовнішню дію. Наприклад, Ваш батько на прохання про виділення певної суми на кишенькові витрати може дати відповідь "так" або "ні". Якщо Ви натиснете на кнопку ліфта "Пуск", то це призведе до його руху – вверх або вниз. Ствердна відповідь і рух ліфта є прикладами поведінки. Поведінка схожа з роботою функції: Ви викликаєте функцію для того, щоб зробити яку-небудь дію (наприклад, вивести на екран обліковий запис), і функція здійснює цю дію.

Таким чином, ні окремо взяті глобальні дані, ні відокремлені від них функції не здатні адекватно відображати фізичні об'єкти реального світу.

1.2. Поняття про об'єктно-орієнтований підхід до розроблення складних програм

Об'єднання даних і дій над ними в єдине ціле. Визначальною ідеєю об'єктно-орієнтованого підходу до розроблення сучасних програмних продуктів є поєднання *даних* і *дій, що виконуються над ними,* в єдине ціле, яке називають *об'єктом.*

Функції об'єкта, які у мові програмування C++ називають *методами* або *функціями-членами* класу*,* зазвичай призначені для доступу до даних об'єкта та виконання певних дій над ними. Наприклад, якщо необхідно зчитувати будь-які значення даних об'єкта, то потрібно викликати відповідну функцію, яка їх зчитає та поверне об'єкту. Зазвичай прямий доступ до даних є неможливим, тому вони приховані від зовнішніх дій, що захищає їх від випадкових змін. Вважають, що дані та методи класу між собою *інкапсульовані.* Терміни *приховання* та *інкапсуляція* даних є ключовими в описі об'єктно-орієнтованих мов програмування.

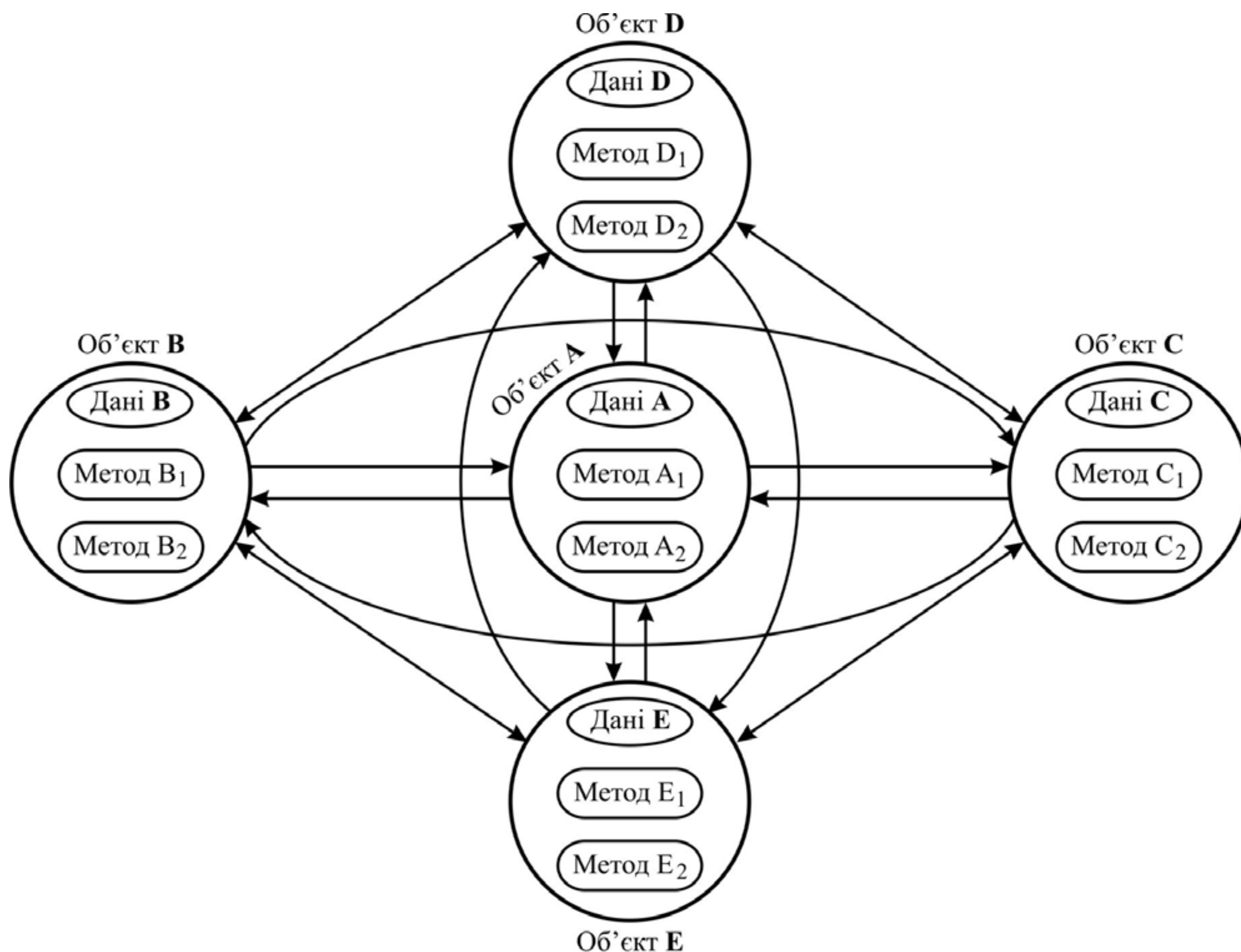

Рис. 1.3. Об'єктно-орієнтований підхід до встановлення зв'язків між даними і методами (функціями)

Якщо необхідно змінити значення даних об'єкта, то, очевидно, ця дія також має бути покладена на відповідну функцію об'єкта. Ніякі інші функції не можуть змінювати значення даних об'єкта, тобто дані класу. Такий підхід полегшує написання, відлагодження та використання програми. Таким чином, типова програма, написана мовою С++, складається з сукупності об'єктів, що взаємодіють між собою за допомогою методів, які викликають один одного. Об'єктно-орієнтований підхід до встановлення зв'язків між даними і функціями (методами) класу, написаної мовою С++, наведено на рис. 1.3.

Виробнича аналогія. Для кращого розуміння призначення об'єктів, уявімо собі деяку промислову фірму, яка складається з бухгалтерії, відділу кадрів, відділу реалізації продукції, відділів головного технолога та головного механіка (рис. 1.4) і т.д. Поділ фірми на відділи є важливою частиною структурної організації її виробничої діяльності. Для більшості фірм (за винятком невеликих) в обов'язки окремого співробітника не входить вирішення одночасно кадрових, виробничих і обліково-бухгалтерських питань. Різні обов'язки чітко розподіляються між підрозділами, тобто у кожного підрозділу є дані, з якими він працює: у бухгалтерії – заробітна плата, у відділі реалізації продукції – інформація, яка стосуються торгівлі, у відділі кадрів – персональна інформація про співробітників, у відділі головного технолога – стан технологічного процесу виготовлення продукції, у відділі головного механіка – технічний стан обладнання та устаткування, транспортних засобів і т.д.

Співробітники кожного відділу виконують операції тільки з тими даними, які їм належать.

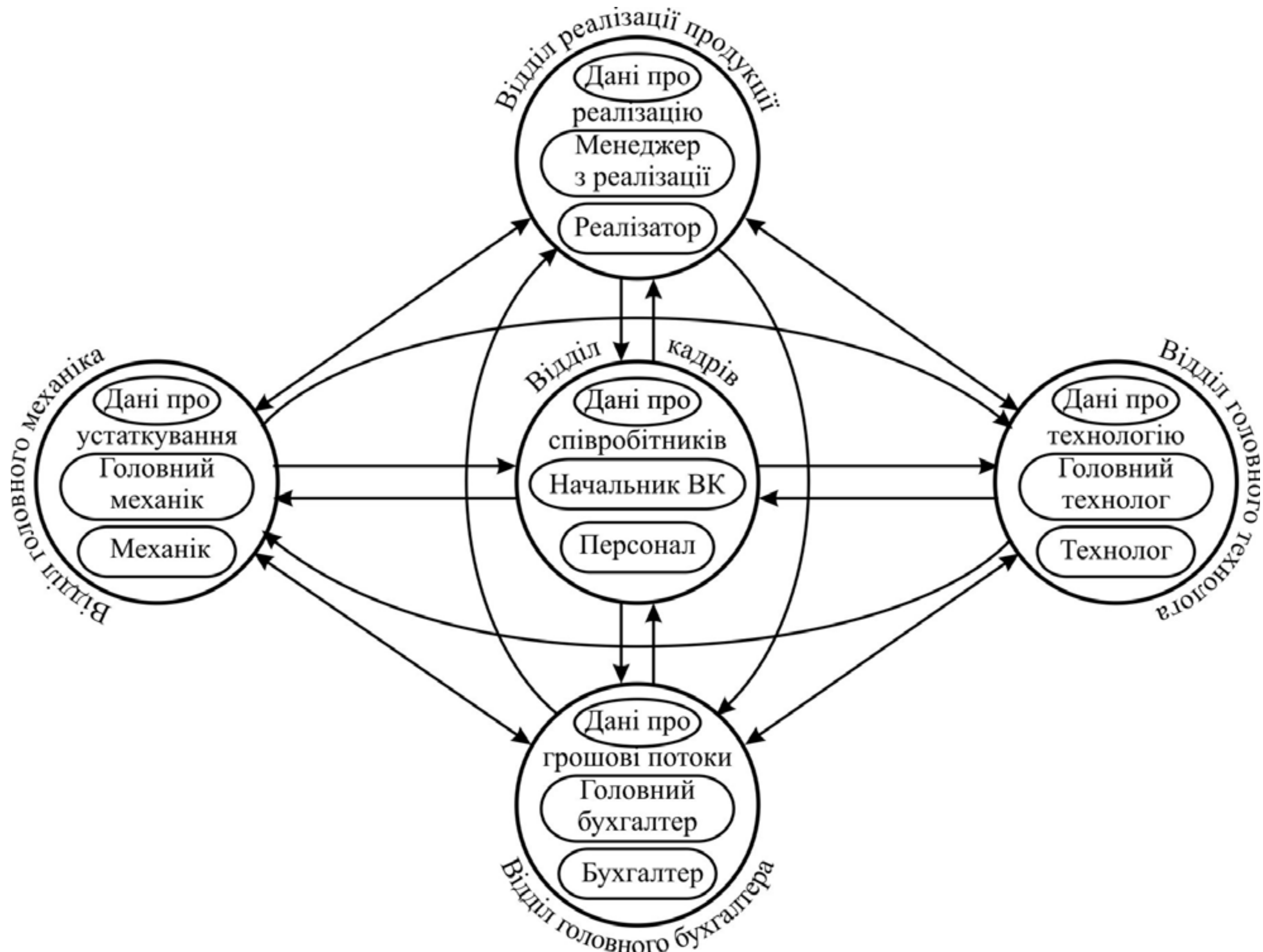

Рис. 1.4. Корпоративний підхід до встановлення зв'язків між відділами і їх виробничими функціями

Структурний поділ обов'язків дає змогу легко стежити за виробничою діяльністю фірми та контролювати її, а також підтримувати цілісність інформаційного простору фірми. Наприклад, бухгалтерія несе відповідальність за інформацію, яка стосується нарахування заробітної плати, сплати податків, здійснення обліку матеріальних цінностей тощо. Якщо у менеджера з реалізації продукції виникне потреба дізнатися про загальний оклад співробітників фірми за деякий місяць, то йому не потрібно йти в бухгалтерію і ритися в картотеках; йому достатньо послати запит бухгалтеру з нарахування заробітної плати, який має знайти потрібну інформацію, обробити її та надати достовірну відповідь на запит. Така схема організації роботи фірми забезпечує правильне оброблення даних, а також здійснює їх захист від можливої дії сторонніх осіб. Подібну структуру зв'язків між відділами фірми та їх виробничими функціями зображено на рис. 1.4. Аналогічні об'єкти коду програми створюють таку саму її організацію, яка має забезпечити цілісність її даних, доступ до них і виконання над ними відповідних дій.

ООП – ефективний спосіб організації програми. ООП ніяк не пов'язане з процесом виконання програми, а є тільки способом її ефективної організації. Велика частина операторів мови C++ є аналогічною операторам проце-

дурних мов програмування, зокрема мови С. Зовні функція класу у мові програмування С++ дуже схожа на звичайну функцію мови С, і тільки за контекстом програми можна визначити, чи є функція частиною структурної С-програми або об'єктно-орієнтованої програми, написаної мовою С++.

1.3. Основні компоненти об'єктно-орієнтованої мови програмування

Розглянемо декілька основних компонент, що входять до складу будь-якої об'єктно-орієнтованої мови програмування, у тому числі і мови С++: об'єкти, класи, успадкування, повторне використання коду програми, типи даних користувача, поліморфізм і перевизначення операторів тощо.

Поділ програми на об'єкти. Якщо Вам доведеться розв'язувати деяку прикладну задачу з використанням об'єктно-орієнтованого підходу, то замість проблеми поділу програми на функції наштовхнетесь на проблему поділу її на об'єкти. Згодом Ви зрозумієте, що мислення в термінах об'єктів виявляється набагато простішим і більш наочним, ніж у термінах функцій, оскільки програмні об'єкти схожі з фізичними об'єктами реального світу.

Тут ми дамо відповідь тільки на таке запитання: що у коді програми треба представляти у вигляді об'єктів? Остаточну відповідь на це запитання може дати тільки Ваш власний досвід програмування з використанням об'єктно-орієнтованого підходу, а також Ваше досконале знання фізичної сутності розв'язуваної задачі. Проте нижче наведено декілька прикладів, які можуть виявитися корисними як для початківців-студентів, так і для програмістів-нефахівців з ООП:

- ***фізичні об'єкти***: верстати та устаткування під час моделювання перебігу технологічного процесу виготовлення продукції; транспортні засоби під час моделювання процесу переміщення продукції; схемні елементи під час моделювання роботи ланцюга електричного струму; виробничі підприємства під час розроблення економічної моделі; літальні апарати під час моделювання диспетчерської системи тощо;
- ***елементи інтерфейсу***: вікна програми; меню користувача; графічні об'єкти (лінії, прямокутники, круги); миша, клавіатура, дискові пристрої, принтери, плоттери тощо;
- ***структури даних***: звичайні та розріджені масиви; стеки; одно- і двозв'язні списки; бінарні дерева тощо;
- ***групи людей***: співробітники; студенти; покупці; продавці тощо;
- ***сховища даних***: описи обладнання та устаткування; інформація про виготовлену продукцію; списки співробітників; словники; географічні координати точок тощо;
- ***типи даних користувача***: час; довжини; грошові одиниці; величини кутів; комплексні числа; точки на площині чи у просторі;
- ***учасники комп'ютерних ігор***: автомобілі на перегонах; позиції в настільних іграх (шашки, шахи); тварини в іграх, пов'язані з живою природою; друзі та вороги в пригодницьких іграх.

Відповідність між програмними і реальними об'єктами є наслідком об'єднання даних і відповідних їм функцій. Об'єкти, які отримано внаслідок такого об'єднання, свого часу викликали фурор, адже жодна програмна модель, розроблена на основі структурного підходу, не відображала наявні речі так точно, як це вдалося зробити за допомогою об'єктів.

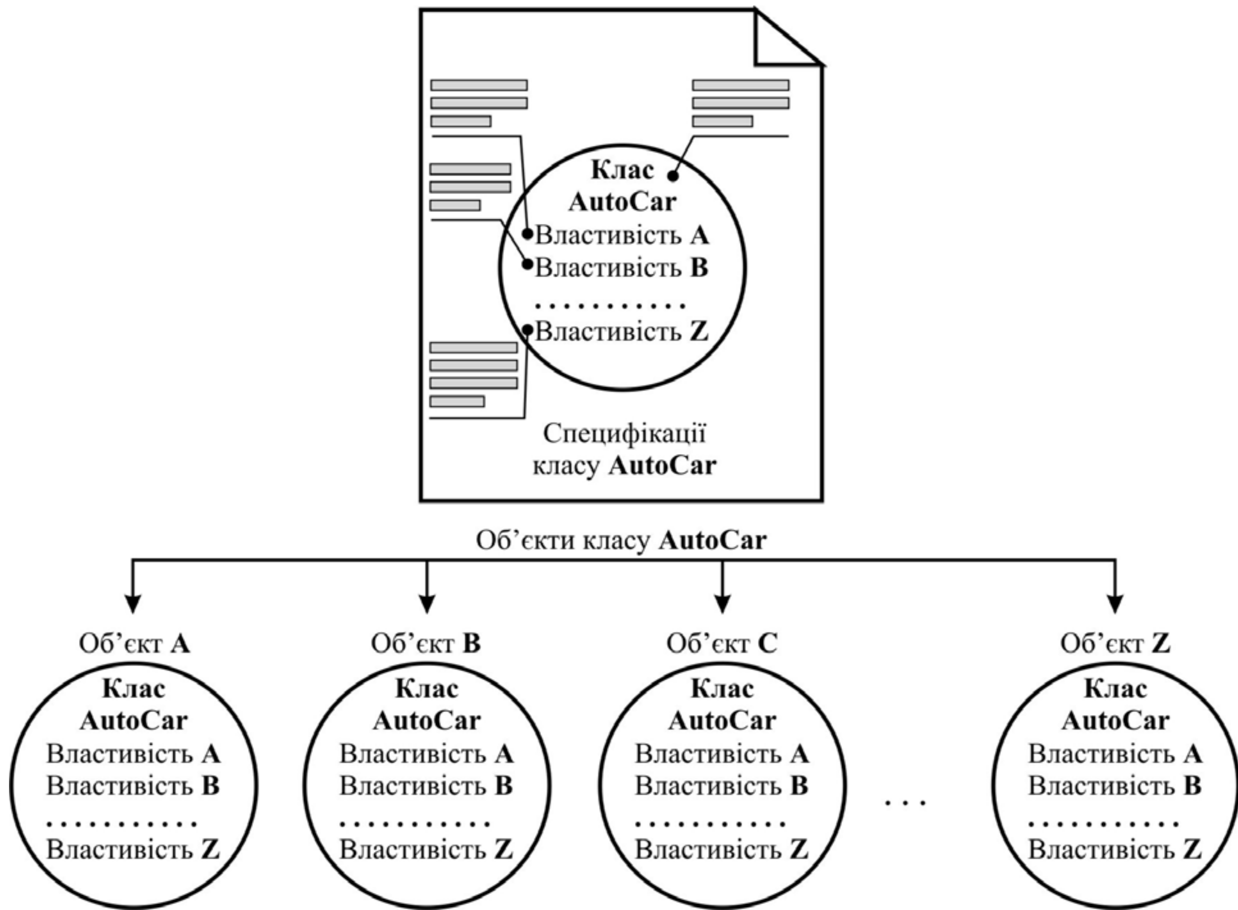

Рис. 1.5. Визначення класу і його об'єктів

Визначення класу. Коли йдеться про об'єкти, то вважається, що вони є екземплярами класів. Що це означає? Розглянемо таку тривіальну аналогію. Практично всі комп'ютерні мови мають стандартні типи даних; наприклад, у мові програмування C++ є цілий тип **int**. Ми можемо визначати змінні таких типів у наших програмах:

```
int day, count, divisor, answer;
```

За аналогією ми можемо також визначати об'єкти класу, як це показано на рис. 1.5. Тобто, клас – це тип форми, що визначає, які дані та функції будуть включені в об'єкт. Під час оголошення класу не створюються ніякі об'єкти цього класу, за аналогією з тим, що існування типу **int** ще не означає наявність змінних цього типу.

Таким чином, визначальним для класу є тип сукупності об'єктів, схожих між собою. Це відповідає нестрогому в технічному сенсі розумінню терміну "клас": наприклад, Prince, Sting і Madonna належать до класу музикантів. Не існує конкретної людини з іменем "рок-музикант", проте люди зі своїми унікальними іменами є об'єктами цього класу, якщо вони володіють певним набором характеристик. Об'єкт класу часто також називають екземпляром *класу.*

Поняття про успадкування в класах. Поняття класу тісно пов'язане з поняттям *успадкування в класах.* У повсякденному житті ми часто маємо справу з поділом класів на підкласи: наприклад, клас *тварини* можна розбити на підкласи *ссавці, земноводні, комахи, птахи* і т.д. Клас *наземний транспорт* поділяється на класи *автомобілі, вантажівки, автобуси, мотоцикли, автокрани* і т.д.

Принцип, закладений в основу такого поділу, полягає в тому, що кожен підклас володіє властивостями, притаманними тому класу, з якого виділений даний підклас. Автомобілі, вантажівки, автобуси і мотоцикли мають колеса і двигун, який є характеристиками наземного транспорту. Окрім тих властивостей, які є загальними у даних класу і підкласу, підклас може мати і власні: наприклад, автобуси мають велику кількість посадкових місць для пасажирів, тоді як вантажівки володіють значним простором і потужністю двигуна для перевезення вантажів, негабариту, в т.ч. і інших машин. Ілюстрацію ідеї успадкування в класах наведено на рис. 1.6.

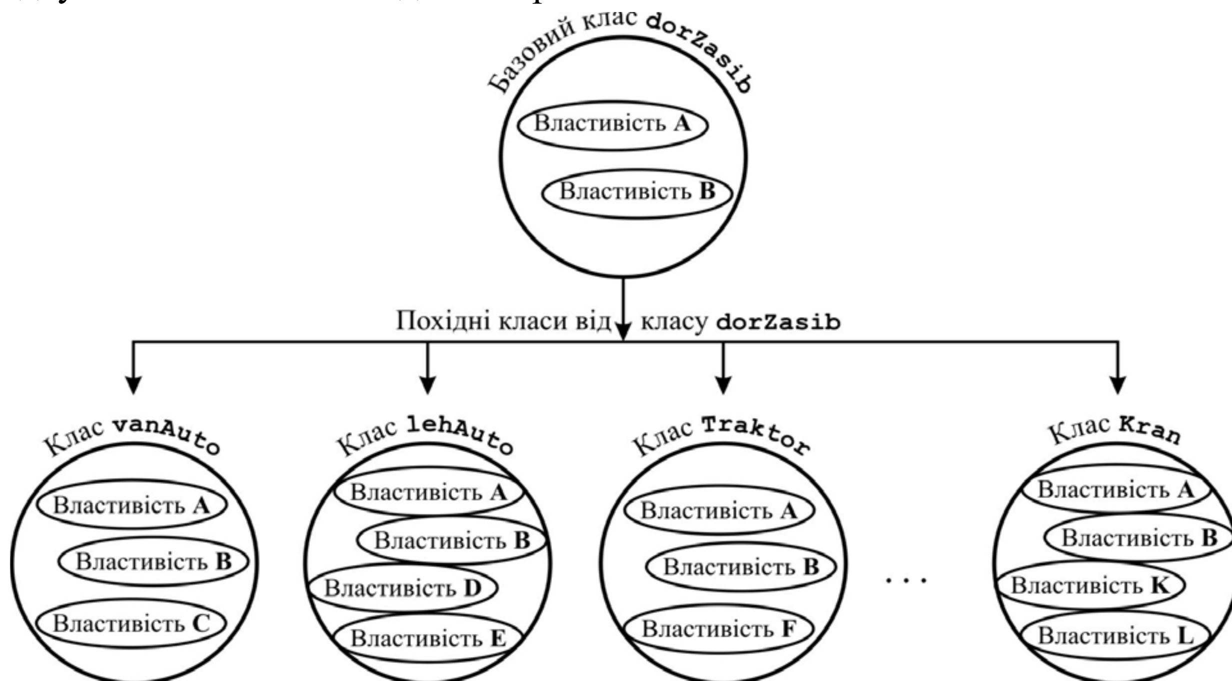

Рис. 1.6. Ілюстрація ідеї успадкування в класах

Подібно до наведеного вище класу наземного транспорту, у програмуванні клас також може породити множину підкласів. У мові програмування С++ клас, який породжує всю решту класів, називається *базовим класом;* решта класів успадковує його властивості, водночас мають власні характеристики. Такі класи називають *похідними.*

Не проводьте помилкових аналогій між відносинами "об'єкт-клас" і "базовий клас – похідний клас"! Об'єкти, які існують в пам'яті комп'ютера, є втіленням властивостей, притаманних класу, до якого вони належать. Похідні класи мають властивості як успадковані від базового класу, так і свої власні.

Успадкування можна вважати аналогом використання функцій в структурному підході. Якщо ми виявимо декілька функцій, які здійснюють схожі дії, то вилучимо з них однакові частини і винесемо їх у окрему функцію. Тоді початкові функції будуть однаковою мірою викликати свою загальну части-

ну, і водночас у кожній з них міститимуться свої власні настанови. Базовий клас містить елементи, які є загальними для групи похідних класів. Значення успадкування в ООП таке ж саме, як і функцій у структурному програмуванні, – скоротити розмір коду програми і спростити зв'язки між її елементами.

Повторне використання коду. Розроблений раніше клас часто можна використовувати в інших програмах. Ця властивість називається *повторним використанням коду.* Аналогічну властивість в структурному програмуванні мають бібліотеки функцій, які можна вносити в різні програмні проекти.

У ООП концепція успадкування відкриває нові можливості повторного використання коду програми. Програміст може узяти наявний клас, і, нічого не змінюючи в ньому, додати до нього свої елементи. Всі похідні класи успадкують ці зміни і, водночас, кожний з похідних класів також можна окремо модифікувати.

Припустимо, що Ви розробили клас, який представляє систему меню, аналогічну графічному інтерфейсу Microsoft Windows або іншому графічному інтерфейсу користувача (GUI). Ви не хочете змінювати цей клас, але Вам необхідно мати можливість встановлювати та знімати опції. У цьому випадку Ви створюєте новий клас, який успадковує всі властивості початкового класу, і додаєте до нього необхідний код програми.

Зручність повторного використання кодів уже написаних програм є важливою перевагою технології ООП над іншими технологіями. Багато комп'ютерних компаній стверджують, що можливість вносити в нові версії програмного забезпечення старі коди програм сприяє зростанню їх прибутків, які вони приносять. Детальніше це питання розглядатиметься в інших розділах цього навчального посібника.

Поняття про типи даних користувача. Одним з достоїнств об'єктів є те, що вони дають змогу програмісту створювати свої власні типи даних. Уявіть собі, що Вам необхідно працювати з об'єктами, які мають дві координата, наприклад x і y. Вам хотілося б здійснювати звичайні арифметичні операції над такими об'єктами, наприклад:

```
Ob1 = Ob2 + obj;
```

де змінні Ob1, Ob2 і obj є наборами з двох координат. Описавши клас, який містить пару координат, і оголосивши об'єкти цього класу з іменами Ob1, Ob2 і obj, ми фактично створимо новий тип даних. У мові програмування C++ є засоби, які полегшують створення подібних типів даних користувача.

Поняття про поліморфізм і перевизначення операторів. Зверніть увагу на те, що операції присвоєння (=) і додавання (+) для типу Coordinate мають виконувати дії, які відрізняються від тих, які вони виконують для об'єктів стандартних типів, наприклад **int**. Об'єкти Ob1 та інші не є стандартними, оскільки визначені користувачем як такі, що належать до класу Coordinate. Як же оператори "–" і "+" розпізнають, які дії необхідно зробити над операндами? Відповідь на це запитання полягає в тому, що ми самі можемо задати ці дії, зробивши потрібні оператори методами класу Coordinate.

Використання окремо операцій і функцій залежно від того, з якими типами величин їм доводиться у даний момент працювати, називають *поліморфізмом.* Коли наявна операція, наприклад, "–" або "+", наділяється можливістю здійснювати дії над операндами нового типу, то вважається, що така операція є *перевизначеною.* Перевизначення є окремим випадком поліморфізму і є важливим інструментом ООП.

1.4. Співвідношення між мовами програмування С і С++

Мова програмування С++ успадкувала можливості мови С. Строго кажучи, мова С++ є розширенням мови С: будь-яка конструкція, написана мовою С, є коректною для мови С++; водночас зворотне твердження – є хибним. Найбільш значні нововведення, які присутні у мові С++, стосуються класів, об'єктів і ООП (первинна назва мови С++ – "С з класами"). Проте є і інші удосконалення, пов'язані із способами організації введення/виведення і написання коментарів. Ілюстрацію співвідношення між мовами С і С++ наведено на рис. 1.7.

На практиці існує значно більше відмінностей між мови С і С++, ніж може видатися спочатку. Незважаючи на те, що мовою С++ можна написати коди програм такі ж самі як і мовою С, однак навряд чи комусь прийде в голову це так робити. Програмісти, які застосовують мову С++, не тільки користуються перевагами цієї мови перед мовою С, але і по-новому використовують її можливості, частина з яких успадкована від мови С. Якщо Ви знайомі з мовою С, то це означає, що у Вас вже є деякі знання відносно мови програмування С++, але ймовірніше за все, що значна частина матеріалу виявиться для Вас новою.

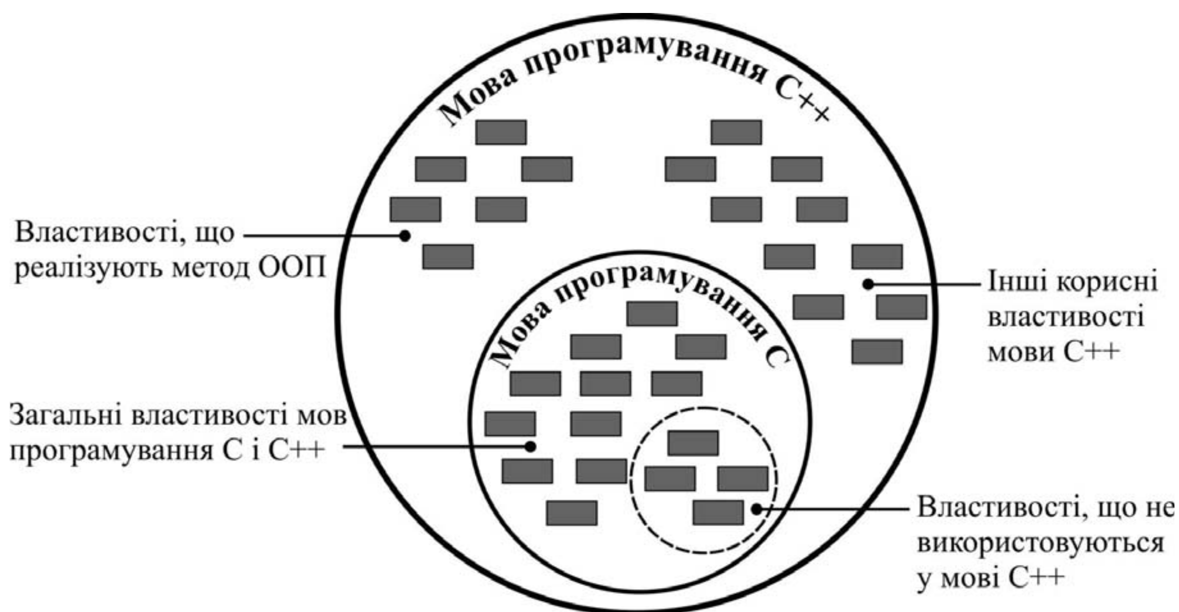

Рис. 1.7. Співвідношення між мовами С і С++

Завдання цього навчального посібника полягає в тому, щоб якнайшвидше навчити Вас створювати об'єктно-орієнтовані програми мовою програмування С++. Оскільки, як було вже сказано раніше, значна частина можливос-

тей мови С++ успадкована від її попередниці – мови С, то навіть при об'єктно-орієнтованій структурі програми її основу становлять "старомодні" процедурні засоби.

Якщо Ви вже знайомі з мовою С, все ж пам'ятайте про певні відмінності між мовами програмування С і С++, які є очевидними, а також можуть бути непомітними при неуважному її вивченні. Тому ми радимо програмістам-нефахівцям з ООП нашвидкуруч проглянути той матеріал, який йому відомий, а основну увагу сконцентрувати на відмінностях між мовами С і С++.

Детальний виклад основних положень ООП починається в наступному розділі. Потім на конкретних прикладах вивчається робота з класами, перевизначення операторів і успадкування в класах, віртуальні функції та поліморфізм, шаблони в класах і оброблення виняткових ситуацій, С++-система введення-виведення, динамічна ідентифікація типів і оператори приведення типу, простір імен і інші ефективні програмні засоби, введення в стандартну бібліотеку шаблонів і особливості роботи препроцесора С++.

1.5. Поняття про універсальну мову моделювання

Універсальну мову моделювання (Unified Modeling Language – UML) можна умовно назвати графічною мовою, призначеною для моделювання структури комп'ютерних кодів програм. У програмуванні під моделюванням розуміють процес розроблення наочної візуальної інтерпретації будь-якого процесу, в т.ч. і структури програмного продукту. Окрім цього, мова UML дає змогу створювати подібну інтерпретацію кодів програм високорівневої ієрархічної організації.

Родоначальниками мови UML стали три незалежні мови моделювання, розробниками яких були відповідно Граді Буч[1], Джеймс Рембо і Івар Джекобсон[2]. У кінці 90-х років XX ст. вони об'єднали свої розробки, внаслідок чого отримали продукт під назвою *універсальна мова моделювання* (*UML*), яка була схвалена OMG (Object Management Group) консорціумом компаній, які визначають промислові стандарти.

Яка ж основна необхідність використання UML при створенні сучасного програмного продукту? По-перше, часто буває важко встановити взаємовідносини між частинами великої програми за допомогою тільки безпосереднього аналізу її коду. Як уже зазначалося раніше, ООП є прогресивнішим, ніж структурне. Але навіть при цьому підході для того, щоби розібратися в конкретних функціональних діях сучасного програмного продукту, необхідно, як мінімум, уявляти собі зміст його програмного коду. Проблема аналізу коду програми полягає в тому, що він є дуже детальним. Набагато простіше було б

[1] Граді Буч (Grady Booch; *27 лютого 1955 р., Техас) – американський вчений в галузі інформаційних технологій і програмування. Автор класичних праць з об'єктно-орієнтованого аналізу. Один з творців мови UML.

[2] Івар Яльмар Джекобсон (*2 вересня 1939 р.) – Шведський вчений в галузі комп'ютерних технологій, відомий як основний розробник таких мов як UML, Objectory, RUP і об'єктно-орієнтованого підходу до розроблення програмного забезпечення.

поглянути на його загальну структуру, яка відображає тільки основні частини програми і їх взаємодію. UML забезпечує таку можливість.

Найбільш важливим засобом UML є набір різних видів діаграм. *Діаграми класів* ілюструють відносини між різними класами, *діаграми об'єктів* – між окремими об'єктами, *діаграми зв'язків* відображають зв'язки між об'єктами у часі і т.д. Усі ці діаграми, по суті, відображають погляди на структуру програми і її функціонування з різних точок зору.

Окрім ілюстрації структури коду програми, UML має немало інших корисних можливостей. У деяких розділах цього посібника йдеться про те, як за допомогою UML можна розробити первинну структуру коду програми. Фактично UML можна використовувати на всіх етапах реалізації проекту – від усвідомлення та аналізу завдання, розроблення та відлагодження програми до документування, тестування і підтримки.

Проте не варто розглядати мову UML як засіб розроблення програмного продукту. Середовище UML є тільки засобом для ілюстрації структури проекту, який розробляється. Незважаючи на можливість застосування середовища UML до будь-якої мови програмування, однак вона є найбільш корисною під час застосування об'єктно-орієнтованого програмування мовою С++.

* * *

ООП є способом організації коду програми. Основну увагу під час його вивчення приділено організації програми, а не питанням написання коду програми. Головним компонентом об'єктно-орієнтованого коду програми є об'єкт, який містить глобальні дані та функції для роботи з ними. Клас є формою або зразком для визначення множини схожих між собою об'єктів.

Механізм успадкування дає змогу створювати нові класи на основі наявних класів, не вносячи змін у останні. Породжений клас успадковує всі дані та функції свого попередника, але має також і свої власні дані та функції. Успадкування уможливлює повторне використання коду програми, тобто внесення одного разу розробленого класу в будь-які інші коди програм.

Мову програмування С++ створено внаслідок розширення мови С, яка дає змогу реалізувати концепцію ООП, а також містить деякі додаткові можливості. Частина засобів мови С, незважаючи на їх підтримку мови С++, визнано застарілими в контексті нових підходів до програмування і тому використовується рідко, як правило, замінюється новішими засобами мови С++. Внаслідок цього відмінності між мовами С і С++ є набагато значнішими, ніж це здається на перший погляд.

Універсальна мова моделювання (UML) є стандартизованим засобом візуалізації структури і функціонування коду програми за допомогою діаграм.

Ідеї, які розглядалися в цьому розділі, конкретизуються у міру вивчення ООП мовою С++. Можливо, в процесі ознайомлення з іншими розділами цього навчального посібника Ви відчуєте потребу повернутися до матеріалу, викладеного в цьому розділі або у першій книзі [9].

Розділ 2. КЛАСИ – ОСНОВА ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ

У цьому розділі ми познайомимося з таким поняттям ООП як класом. ***Клас*** – це основа C++-підтримки ООП, а також ядро багатьох складних програмних засобів. Клас – це базова одиниця інкапсуляції (поєднання даних і дій над ними в єдине ціле), яка забезпечує механізм побудови об'єктів.

2.1. Базові поняття класу

Об'єктно-орієнтований під до розроблення програмних продуктів побудований на такому понятті як класи. *Клас* визначає новий тип даних, який задає формат *об'єкта.* Клас містить як дані, так і коди програм, призначені для виконання дій над ними. Загалом, *клас пов'язує дані з кодами програми.* У мові програмування C++ специфікацію класу використовують для побудови об'єктів. *Об'єкти* – це примірники класового типу. Загалом, клас є набором планів і дій, які вказують на властивості об'єкта та визначають його поведінку. Важливо розуміти, що клас – це логічна абстракція, яка реально не існує доти, доки не буде створено об'єкт цього класу, тобто це те, що стане фізичним представленням цього класу в пам'яті комп'ютера.

Визначаючи клас, оголошують ті глобальні дані, які міститимуть об'єкти, і програмні коди, які виконуватимуться над цими даними. Хоча інколи прості класи можуть містити тільки програмні коди або тільки дані, проте більшість реальних класів містять обидва компоненти. У класі дані оголошуються у вигляді змінних, а програмні коди оформляють у вигляді функцій. Змінні та функції називаються *членами* класу. Змінну, оголошену в класі, називають *членом даних,* а оголошену в класі функцію – *функцією-членом класу.* Іноді замість терміна *член даних класу* використовують термін *змінна класу*, а замість терміна *функція-член класу* використовують термін *метод класу.*

Оголошення класу синтаксично є подібним до оголошення структури. Загальні формати оголошення класу мають такий вигляд:

Варіант 1

```
class ім'я_класу {
private:
    закриті дані та функції класу
public:
    відкриті дані та функції класу
} перелік_об'єктів_класу;
```

Варіант 2

```
class ім'я_класу {
private:
    закриті дані та функції класу
public:
    відкриті дані та функції класу
};
ім'я_класу перелік_об'єктів_класу;
```

У цих оголошеннях елемент *ім'я_класу* означає ім'я "класового" типу. Воно стає іменем нового типу, яке можна використовувати для побудови об'єктів цього класу. Об'єкти класу інколи створюють шляхом вказання їх імен безпосередньо за

закритою фігурною дужкою оголошеного класу як елемент *перелік_об'єктів_класу* (варіанти 1). Проте найчастіше об'єкти створюють в міру потреби після оголошення класу (варіант 2).

Наприклад, наведений нижче клас під іменем myClass визначає тип майбутнього об'єкта, призначеного для обчислення значення арифметичного виразу $a = x^{1.3} + \sqrt[3]{\cos^2 |3.2x - y|^{0.4}}$ при заданих значеннях аргументів $x = 2,6$ і $y = 7.1$:

```
class myClass {                        // Оголошення класового типу
        double a;
public:
        void Init();                   // Ініціалізація даних класу myClass.
        void Get(double, double);      // Введення в об'єкт значення.
        double Put();                  // Виведення з об'єкта значення.
};
```

Розглянемо детально механізм визначення цього класу. Усі члени класу myClass оголошені в тілі настанови **class**. Членом даних класу myClass є змінна a. Окрім цього, тут визначено три функції-члени класу: Init() – ініціалізація об'єта, Get() – введення в об'єкт значення і Put() – виведення з об'єкта значення.

Будь-який клас може містити як закриті, так і відкриті члени. За замовчуванням усі члени, визначені в класі, є закритими (**private**-членами). Наприклад, змінна a є закритим членом даних класу. Це означає, що до неї можуть отримати доступ тільки функції-члени класу myClass; ніякі інші частини програми цього зробити не можуть. У цьому полягає один з проявів інкапсуляції: програміст повною мірою може керувати доступом до певних елементів даних. Закритими можна оголосити і функції (у наведеному вище прикладі таких немає), внаслідок чого їх зможуть викликати тільки інші члени цього класу.

Щоб зробити члени класу відкритими (тобто доступними для інших частин програми), необхідно визначити їх після ключового слова **public**. Усі змінні або функції, визначені після специфікатора доступу **public**, є доступними для всіх інших функцій програми, в тому числі і функції **main**(). Отже, в класі myClass функції Init(), Get() і Put() є відкритими. Зазвичай у програмі організовується доступ до закритих членів класу через його відкриті функції. Зверніть увагу на те, що після ключового слова **public** знаходиться двокрапка.

Вартоазнати! *Об'єкт утворює зв'язки між різними частинами коду програми і її даними. Так, будь-яка функція-член класу має доступ до закритих елементів класу. Це означає, що функції Init(), Get() і Put() мають доступ до змінної a. Щоб додати будь-яку функцію в клас, необхідно оголосити її прототип у визначенні цього класу, після чого вона стає функцією-членом класу.*

Визначивши клас, можна створити об'єкт цього "класового" типу, якщо використати ім'я класу. Отож, ім'я класу стає специфікатором нового типу. Наприклад, у процесі виконання наведеної нижче настанови створюється два об'єкти ObjA і ObjB типу myClass:

```
myClass ObjA, ObjB;    // Створення об'єктів класу
```

Після створення об'єктів класу кожен з них набуває власні копії членів-даних, які утворює клас. Це означає, що кожний з об'єктів ObjA і ObjB матиме власні копії змінної a. Отже, дані, пов'язані з об'єктом ObjA, відокремлені (ізольовані) від даних, які пов'язані з об'єктом ObjB.

Щоб отримати доступ до відкритого члена класу через об'єкт цього класу, використовують оператор "крапка" (саме так це робиться і під час роботи із структурами). Наприклад, щоб вивести на екран значення змінної a, яка належить об'єкту ObjA, використовують таку настанову:

```
cout << " ObjA.a= " << ObjA.a;
```

Нео! хідноапам'ятатио *У мові програмування C++ клас створює новий тип даних, який можна використовувати для побудови об'єктів. Зокрема, клас створює логічну конструкцію, яка визначає відносини між її членами – даними і кодами функцій. Оголошуючи так звану змінну класу, ми створюємо об'єкт, який фізично існує і є конкретним примірником класу*[1]*. Понад це, кожен об'єкт класу має власну копію даних, які визначено у цьому класі.*

У оголошенні класу myClass містяться прототипи функцій-членів класу. Щоб реалізувати код функції, яка є членом класу, необхідно повідомити компілятор, до якого класу вона належить, кваліфікувавши ім'я цієї функції з іменем відповідного класу. Наприклад, ось як можна записати код функції Get():

```
// Введення в об'єкт значення.
void myClass::Get(double x, double y)
{
        double a1 = pow(x,1.3);
        double a2 = pow(fabs(3.2*x - y),0.4);
        double a3 = pow(pow(cos(a2),2),1./3);
        a = a1+a3;
}
```

За своєю сутністю такий запис повідомляє компілятор про те, що дана версія функції Get() належить класу myClass. Іншими словами, оператор дозволу "::" заявляє компілятору, що функція Get() знаходиться у області видимості класу myClass. Різні класи можуть використовувати однакові імена функцій. Однак, компілятор самостійно визначить, до якого класу належить та чи інша функція за допомогою імені класу та оператора дозволу області видимості.

Функції-члени класу можна викликати тільки зі своїми об'єктами. Щоб викликати функцію-члена з частини програми, яка знаходиться поза класом, необхідно використовувати ім'я об'єкта і оператор "крапка". Наприклад, у процесі виконання такого коду програми буде викликано функцію Init() для об'єкта ObjA:

```
myClass ObjA, ObjB;
ObjA.Init();
```

Під час виклику функції ObjA.Init() дії, визначені у функції Init(), будуть спрямовані на копії даних, які належать тільки об'єкту ObjA. Необхідно мати на увазі, що ObjA і ObjB – це два окремі об'єкти. Це означає, що ініціалізація змінних об'єкта ObjA зовсім не приводить до ініціалізації змінних об'єкта ObjB. Єдине, що зв'язує об'єк-

[1] Іншими словами, об'єкт займає певну область пам'яті, а визначення типу – ні.

ти ObjA і ObjB, це те, що вони мають один і той самий класовий тип і функції-члени класу, які обробляють його дані.

Якщо одна функція-член класу викликає іншу функцію-члена того ж самого класу, то не потрібно вказувати ім'я об'єкта і використовувати оператор "крапка". У цьому випадку компілятор вже точно знає, який об'єкт піддається обробленню. Ім'я об'єкта і оператор "крапка" необхідні тільки тоді, коли функція-член класу викликається кодом програми, розташованим поза класом. З тих самих причин функція-член класу може безпосередньо звертатися до будь-якого члена даних свого класу, але програмний код, розташований поза класом, повинен звертатися до змінної класу, використовуючи ім'я об'єкта і оператор "крапка".

У наведеному нижче коді програми механізм оголошення та застосування класу myClass продемонстровано повністю. Для цього об'єднаємо всі разом вже знайомі нам частини коду програми і додано відсутні деталі.

Код програми 2.1. Демонстрація механізму оголошення класу та його застосування

```
#include <vcl>
#include <iostream>                 // Для потокового введення-виведення
#include < math>                    // Для використання математичних функцій
#include <conio>                    // Для консольного режиму роботи
using namespace std;                // Використання стандартного простору імен

class myClass {                     // Оголошення класового типу
        double a;
public:
        void Init();                // Ініціалізація даних класу
        void Get(double, double);   // Введення в об'єкт значення
        double Put();               // Виведення з об'єкта значення
};

// Ініціалізація даних класу myClass.
void myClass::Init()
{
        a = 0;
}

// Введення в об'єкт значення.
void myClass::Get(double x, double y)
{
        double a1 = pow(x,1.3);
        double a2 = pow(fabs(3.2*x - y),0.4);
        double a3 = pow(pow(cos(a2),2),1./3);
        a = a1+a3;
}

// Виведення з об'єкта значення.
double myClass::Put()
{
        return a;
```

```
}

int main()
{
	myClass ObjA, ObjB;	// Створення двох об'єктів класу.
	double x = 2.6, y = 7.1;

	ObjA.Init(); ObjB.Init();

	ObjA.Get(x,y); ObjB.Get(x+y,y/x);

	cout << "Вміст об'єкта ObjA: " << ObjA.Put() << endl;
	cout << "Вміст об'єкта ObjB: " << ObjB.Put() << endl;

	getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Вміст об'єкта ObjA: 4.06663
Вміст об'єкта ObjB: 20.0294
```

Закриті члени класу доступні тільки функціям, які є членами цього класу. Наприклад, таку настанову

```
ObjA.a = 0;
```

не можна помістити у функцію **main**() нашої програми.

2.2. Поняття про конструктори і деструктори

Як правило, певну частину об'єкта, перш ніж його можна буде використовувати, необхідно ініціалізувати. Наприклад, розглянемо клас myClass, який було представлено вище у цьому підрозділі. Перш ніж об'єкти класу myClass можна буде використовувати, змінній a потрібно надати нульове значення. У нашому випадку ця вимога виконувалася за допомогою функції Init(). Але, оскільки вимога ініціалізації членів-даних класу є достатньо поширеною, то у мові програмування C++ передбачено реалізацію цієї потреби при створенні об'єктів класу. Така автоматична ініціалізація членів-даних класу здійснюється завдяки використанню конструктора.

Конструктор – це спеціальна функція-член класу, яка викликається при створенні об'єкта, а її ім'я обов'язково збігається з іменем класу. Ось, наприклад, як може виглядати клас myClass після застосування у ньому конструктора для ініціалізації членів-даних класу:

```
class myClass {				// Оголошення класового типу
	double a;
public:
	myClass();			// Оголошення конструктора
	void Get(double, double);	// Введення в об'єкт значення
	double Put();			// Виведення з об'єкта значення
};
```

Зверніть увагу на те, що в оголошенні конструктора myClass() відсутній тип значення, що повертається. У мові програмування C++ конструктори не повертають значень і, як наслідок, немає сенсу вказувати їх тип[1].

Тепер наведемо код функції myClass():

```
// Визначення конструктора
myClass::myClass()
{
    a = 0; cout << "Об'єкт ініціалізовано" << endl;
}
```

У цьому випадку у процесі виконання конструктора ним виводиться повідомлення "Об'єкт ініціалізовано", яке слугує виключно ілюстративній меті. На практиці ж здебільшого конструктори не виводять ніяких повідомлень.

Конструктор об'єкта викликається при створенні об'єкта. Це означає, що він викликається у процесі виконання настанови створення об'єкта. Конструктори глобальних об'єктів викликаються на самому початку виконання програми, тобто ще до звернення до функції **main**(). Що стосується локальних об'єктів, то їх конструктори викликаються кожного разу, коли виникає потреба створення такого об'єкта.

Доповненням до конструктора слугує *деструктор* – це функція, яка викликається під час руйнування об'єкта. У багатьох випадках під час руйнування об'єкта необхідно виконати певну дію або навіть певні послідовності дій. Локальні об'єкти створюються під час входу в блок, у якому вони визначені, і руйнуються при виході з нього. Глобальні об'єкти руйнуються внаслідок завершення програми. Існує багато чинників, які заставляють використовувати деструктори. Наприклад, об'єкт повинен звільнити раніше виділену для нього пам'ять. У мові програмування C++ саме деструкторам доручається оброблення процесу деактивізації об'єкта.

Ім'я деструктора має збігатися з іменем конструктора, але йому передує символ "~". Подібно до конструкторів, деструктори не повертають значень, а отже, в їх оголошеннях відсутній тип значення, що повертається.

Розглянемо вже знайомий нам клас myClass, але тепер він має містити конструктор і деструктор[2]:

```
class myClass {                     // Оголошення класового типу
    double a;
public:
    myClass();                      // Оголошення конструктора
    ~myClass();                     // Оголошення деструктора
    void Get(double, double);       // Введення в об'єкт значення
    double Put();                   // Виведення з об'єкта значення
};

// Визначення конструктора.
```

[1] При цьому не можна вказувати навіть тип `void`.

[2] Задля справедливості зазначимо, що класу myClass деструктор не потрібний, а його наявність тут можна виправдати тільки з ілюстративною метою.

```
myClass::myClass()
{
	a = 0; cout << "Об'єкт ініціалізовано" << endl;
}

// Визначення деструктора.
myClass::~myClass()
{
	cout << "Об'єкт зруйновано" << endl;
}
```

Ось як виглядатиме нова версія коду програми, призначеної для обчислення значення арифметичного виразу, у якій продемонстровано механізм використання конструктора і деструктора.

Код програми 2.2. Демонстрація механізму використання конструктора і деструктора

```
#include <vcl>
#include <iostream>              // Для потокового введення-виведення
#include < math>                 // Для використання математичних функцій
#include <conio>                 // Для консольного режиму роботи
using namespace std;             // Використання стандартного простору імен

class myClass {                  // Оголошення класового типу
	double a;
public:
	myClass();               // Оголошення конструктора
	~myClass();              // Оголошення деструктора
	void Get(double, double);    // Введення в об'єкт значення
	double Put();            // Виведення з об'єкта значення
};

// Визначення конструктора.
myClass::myClass()
{
	a = 0; cout << "Об'єкт ініціалізовано" << endl;
}

// Визначення деструктора.
myClass::~myClass()
{
	cout << "Об'єкт зруйновано" << endl;
}

// Введення в об'єкт значення.
void myClass::Get(double x, double y)
{
	double a1 = pow(x,1.3);
	double a2 = pow(fabs(3.2*x - y),0.4);
	double a3 = pow(pow(cos(a2),2),1./3);
```

```
        a = a1+a3;
}

// Виведення з об'єкта значення
double myClass::Put()
{
        return a;
}

int main()
{
        myClass ObjA, ObjB;      // Створення двох об'єктів класу.
        double x = 2.6, y = 7.1;

        ObjA.Get(x,y); ObjB.Get(x+y,y/x);

        cout << "Вміст об'єкта ObjA: " << ObjA.Put() << endl;
        cout << "Вміст об'єкта ObjB: " << ObjB.Put() << endl;

        getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Об'єкт ініціалізовано
Об'єкт ініціалізовано
Вміст об'єкта ObjA: 4.06663
Вміст об'єкта ObjB: 20.0294
Об'єкт зруйновано.
Об'єкт зруйновано.
```

Параметризовані конструктори. Конструктор може мати параметри, за допомогою яких при створенні об'єкта членам-даних (змінним класу) можна надати деякі початкові значення, які визначаються у основній функції **main**(). Це реалізується шляхом передачі аргументів конструктору об'єкта. У наведеному нижче коді програми спробуємо удосконалити клас myClass так, щоб він приймав аргументи, які слугуватимуть ідентифікаційними номерами (n) черги. Передусім необхідно внести зміни у визначення класу myClass:

```
class myClass {                   // Оголошення класового типу
        double a;
        int nom;                  // Містить ідентифікаційний номер об'єкта
public:
        myClass(int n);           // Оголошення параметризованого конструктора
        ~myClass();               // Оголошення деструктора
        void Get(double, double); // Введення в об'єкт значення
        double Put();             // Виведення з об'єкта значення
};
```

Член-даних nom використовують для зберігання ідентифікаційного номера об'єкта, який створюється основною функцією. Його реальне значення визначається значенням, яке передається конструктору як формальний параметр n при

створенні змінної типу myClass. Параметризований конструктор myClass() буде мати такий вигляд:

```
// Визначення параметризованого конструктора
myClass::myClass(int n)
{
    a = 0; nom = n;
    cout << "Об'єкт " << nom << " ініціалізовано" << endl;
}
```

Щоб передати аргумент конструктору, необхідно пов'язати цей аргумент з об'єктом під час його створення. Мова програмування С++ підтримує два способи реалізації такого зв'язування:

Варіант 1	**Варіант 2 (альтернативний)**
myClass ObjA = myClass(24);	myClass ObjA = 24;

У цих оголошеннях створюється об'єкт з іменем ObjA, якому передається значення (ідентифікаційний номер) 24. Але ці формати (у такому контексті) використовуються рідко, оскільки другий спосіб має коротший запис і тому є зручнішим для використання. У другому способі аргумент повинен знаходитися за іменем об'єкта і поміщатися в круглі дужки. Наприклад, така настанова еквівалентна попереднім визначенням:

```
myClass ObjA(24);
```

Це найпоширеніший спосіб визначення параметризованих об'єктів. З використанням цього способу наведемо загальний формат передачі аргументів конструктору:

тип_класу ім'я_об'єкта(*перелік_аргументів*);

У цьому записі елемент *перелік_аргументів* вказує на те, що усі аргументи, які передаються конструктору, розділені між собою комами.

Вартоа' нати! *Формально між двома наведеними вище форматами ініціалізації параметризованих конструкторів існує невелика відмінність, яку Ви зрозумієте при подальшому вивченні цього навчального посібника. Але ця відмінність не впливає на результати виконання програм, представлених у цьому розділі.*

У наведеному нижче коді програми, призначеної для обчислення значення арифметичного виразу, продемонстровано механізм використання параметризованого конструктора.

Код програми 2.3. Демонстрація механізму використання параметризованого конструктора

```
#include <vcl>
#include <iostream>             // Для потокового введення-виведення
#include < math>                // Для використання математичних функцій
#include <conio>                // Для консольного режиму роботи
using namespace std;            // Використання стандартного простору імен

class myClass {                 // Оголошення класового типу
    double a;
    int nom;                    // Містить ідентифікаційний номер об'єкта
```

```
public:
        myClass(int n);              // Оголошення параметризованого конструктора
        ~myClass();                  // Оголошення деструктора
        void Get(double, double);    // Введення в об'єкт значення
        double Put();                // Виведення з об'єкта значення
};

// Визначення параметризованого конструктора.
myClass::myClass(int n)
{
        a = 0; nom = n;
        cout << "Об'єкт " << nom << " ініціалізовано" << endl;
}

// Визначення деструктора.
myClass::~myClass()
{
        cout << "Об'єкт " << nom << " зруйновано" << endl;
}

// Введення в об'єкт значення.
void myClass::Get(double x, double y)
{
        double a1 = pow(x,1.3);
        double a2 = pow(fabs(3.2*x - y),0.4);
        double a3 = pow(pow(cos(a2),2),1./3);
        a = a1+a3;
}

// Виведення з об'єкта значення.
double myClass::Put()
{
        return a;
}

int main()
{
        // Створення та ініціалізація двох об'єктів.
        myClass ObjA(1), ObjB(2);
        double x = 2.6, y = 7.1;

        ObjA.Get(x,y); ObjB.Get(x+y,y/x);

        cout << "Вміст об'єкта ObjA: " << ObjA.Put() << endl;
        cout << "Вміст об'єкта ObjB: " << ObjB.Put() << endl;

        getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Об'єкт 1 ініціалізовано
Об'єкт 2 ініціалізовано
Вміст об'єкта ObjA: 4.06663
Вміст об'єкта ObjB: 20.0294
Об'єкт 2 зруйновано
Об'єкт 1 зруйновано
```

Як видно з коду основної функції **main**(), об'єкту під іменем ObjA присвоюється ідентифікаційний номер 1, а об'єкту з іменем ObjB – ідентифікаційний номер 2.

Хоча у прикладі з використанням класу myClass при створенні об'єкта передається тільки один аргумент, у загальному випадку можлива передача двох і більше аргументів. У наведеному нижче коді програми об'єктам типу widClass передається два параметри.

Код програми 2.4. Демонстрація механізму передачі конструктору двох параметрів

```
#include <vcl>
#include <iostream>                // Для потокового введення-виведення
#include <conio>                   // Для консольного режиму роботи
using namespace std;               // Використання стандартного простору імен

class widClass {                   // Оголошення класового типу
	int c, d;
public:
	widClass(int a, int b);        // Оголошення параметризованого конструктора
	void Show();
};

// Передаємо 2 аргументи конструктору widClass().
widClass::widClass(int a, int b)
{
	c = a; d = b;
	cout << "Об'єкт ініціалізовано" << endl;
}

void widClass::Show()
{
	cout << "c= " << c << "; d= " << d << endl;
}

int main()
{
	// Створення та ініціалізація двох об'єктів.
	widClass ObjX(10, 20), ObjY(0, 0);

	ObjX.Show();
	ObjY.Show();

	getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Об'єкт ініціалізовано
Об'єкт ініціалізовано
c= 10; d= 20
c= 0; d= 0
```

***Вартоа' нати!а**На відміну від конструкторів, деструктори не мають параметрів. Причина в тому, що не існує потреби передачі значень аргументів об'єктові, який руйнується. Якщо ж у Вас виникне ситуація, коли під час виклику деструктора Вашому об'єктові необхідно отримати доступ до деяких даних, які визначаються тільки у процесі виконання програми, то для цього створіть спеціальну змінну. Потім безпосередньо перед руйнуванням об'єкта цій змінній потрібно встановити значення, яке дорівнює потрібному.*

2.3. Особливості реалізації механізму доступу до членів класу

Усвідомлення та засвоєння механізму доступу до членів-даних класу або функцій членів класу – ось що часто бентежить програмістів-початківців. Тому зупинимося на цьому питанні детальніше. Насамперед розглянемо таку навчальну програму, в якій демонструється механізм отримання доступу до членів класу.

Код програми 2.5. Демонстрація механізму доступу до членів класу

```
#include <vcl>
#include <iostream>              // Для потокового введення-виведення
#include <conio>                 // Для консольного режиму роботи
using namespace std;             // Використання стандартного простору імен

class myClass {                  // Оголошення класового типу
    int a;                       // Закритий член-даних
public:
    int b;                       // Відкритий член-даних
    void Set(int c);             // Відкриті функції-члени класу
    double Put();
    void Fun();
};

void myClass::Set(int c)         // Надання значень членам-даних
{
    a = c;                       // Пряме звернення до змінної a
    b = c*c;                     // Пряме звернення до змінної b
}

double myClass::Put()            // Повернення значення закритого члена даних
{
    return a;                    // Пряме звернення до змінної a
}

void myClass::Fun()              // Надання значень членам-даних
{
```

```
        Set(0);                        // Прямий виклик функції Set() створеним об'єктом
}

int main()
{
        myClass Obj;                   // Створення об'єкта класу

        Obj.Set(5);                    // Надання значень членам-даних
        cout << "Об'єкт Obj після виклику функції Set(5): " << Obj.Put() << " ";
        // До члена b можна отримати прямий доступ, оскільки він є public-членом.
        cout << Obj.b << endl;

        // Члену b можна надати значення безпосередньо, оскільки він є public-членом.
        Obj.b = 20;
        cout << "Об'єкт Obj після встановлення члена Obj.b=20: ";
        cout << Obj.Put() << " " << Obj.b << endl;

        Obj.Fun();
        cout << "Об'єкт Obj після виклику функції Obj.Fun(): ";
        cout << Obj.Put() << " " << Obj.b << endl;

        getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Об'єкт Obj після виклику функції Set(5): 5 25
Об'єкт Obj після встановлення члена Obj.b=20: 5 20
Об'єкт Obj після виклику функції Obj.Fun(): 0 0
```

Тепер розглянемо, як здійснюється доступ до членів класу myClass. Передусім зверніть увагу на те, що для присвоєння значень змінним a і b у функції Set() використовуються такі рядки коду програми:

```
a = c;     // Пряме звернення до змінної a
b = c*c;   // Пряме звернення до змінної b
```

Оскільки функція Set() є членом класу, то вона може безпосередньо звертатися до членів-даних a і b цього ж класу без вказання імені об'єкта (і не використовуючи при цьому оператора "крапка"). Як уже зазначалося вище, функція-член класу завжди викликається для певного об'єкта (а коли ж виклик відбувся, то об'єкт, як наслідок, є відомим). Таким чином, у тілі функції-члена класу немає потреби вказувати об'єкт повторно. Отже, посилання на змінні a і b застосовуватимуться до копій цих змінних, що належать об'єкту, який їх викликає.

Тепер зверніть увагу на те, що змінна b – відкритий (**public**) член даних класу myClass. Це означає, що до змінної b можна отримати доступ з коду програми, визначеного поза тілом класу myClass. Наведений нижче рядок коду програми з функції **main**(), у процесі виконання якої змінній b присвоюється число 20, демонструє реалізацію такого прямого доступу:

```
Obj.b = 20;       // До члена b можна отримати прямий доступ, оскільки він є public-членом.
```

Оскільки ця настанова не належить тілу класу myClass, то до змінної b можливий доступ тільки з використанням імені конкретного об'єкта (у цьому випадку

об'єкта Obj) і оператора "крапка". Тепер зверніть увагу на те, як викликається функція-член класу Fun() з основної функції **main**():

```
Obj.Fun();
```

Оскільки функція Fun() є відкритим членом класу, то її також можна викликати з коду програми, визначеного поза тілом класу myClass, і за допомогою імені конкретного об'єкта (у цьому випадку об'єкта Obj).

Розглянемо детальніше код функції Fun(). Той факт, що вона є функцією-членом класу, дає змогу їй безпосередньо звертатися до інших членів того ж самого класу, не використовуючи оператора "крапка" або імені конкретного об'єкта. Вона викликає функцію-члена Set(). І знову ж таки, оскільки об'єкт вже відомий (він використовується для виклику функції Fun()), то немає ніякої потреби вказувати його ще раз.

Тут важливо зрозуміти таке: коли доступ до деякого члена класу відбувається ззовні цього класу, то його необхідно кваліфікувати (уточнити) за допомогою імені конкретного об'єкта. Але сама функція-член класу може безпосередньо звертатися до інших членів того ж самого класу.

Нео! хідноапам'ятати! *Не варто хвилюватися з приводу того, що Ви ще не відчули упевненості щодо механізму доступу до членів класу. Невелике Ваше занепокоєння при освоєнні цього питання – звичайне явище для програмістів-початківців. Сміливо продовжуйте вивчати навчальний посібник, розглядаючи якомога більше прикладів, і питання доступу до членів класу незабаром стане так само простим, як таблиця множення для першокласника!*

2.4. Класи і структури – споріднені типи

Як згадувалося в розд. 1, у мові програмування C++ структура також володіє об'єктно-орієнтованими можливостями. По суті, класи і структури можна назвати "близькими родичами". За одним винятком, вони взаємозамінні, оскільки структура також може містити дані та програмні коди, які маніпулюють цими даними так само, як і це може робити клас. Єдина відмінність між C++-структурою і C++-класом полягає у тому, що за замовчуванням члени класу є закритими, а члени структури – відкритими. У іншому ж структури і класи мають однакове призначення. Насправді відповідно до формального синтаксису мови програмування C++ оголошення структури створює тип класу.

Розглянемо приклад використання структури з властивостями, подібними до властивостей класу.

Код програми 2.6. Демонстрація механізму використання структури для створення класу

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен

struct myStruct {            // Оголошення структурного типу
    int Put();               // Ці члени відкриті (public)
```

```
        void Get(int d);                    // за замовчуванням.
private:
        int c;
};

int myStruct::Put()
{
        return c;
}

void myStruct::Get(int d)
{
        c = d;
}

int main()
{
        myStruct Obj;      // Створення об'єкта структури

        Obj.Get(10);
        cout << "c= " << Obj.Put() << endl;

        getch(); return 0;
}
```

У цьому коді програми визначається тип структури з іменем myStruct, у якій функції-члени Put() і Get() є відкритими (**public**), а член даних c – закритим (**private**). Зверніть увагу на те, що у структурах для оголошення закритих членів використовують ключове слово **private**.

У наведеному нижче прикладі показано еквівалентну програму, яка використовує замість типу **struct** тип **class**.

Код програми 2.7. Демонстрація механізму використання класу замість структури

```
#include <vcl>
#include <iostream>                  // Для потокового введення-виведення
#include <conio>                     // Для консольного режиму роботи
using namespace std;                 // Використання стандартного простору імен

class myClass {                      // Оголошення класового типу
        int c;                       // Закритий член за замовчуванням
public:
        int Put();
        void Get(int d);
};

int myClass::Put()
{
        return c;
}
```

```
void myClass::Get(int d)
{
	c = d;
}

int main()
{
	myClass Obj;	// Створення об'єкта класу

	Obj.Get(10);
	cout << "c= " << Obj.Put() << endl;

	getch(); return 0;
}
```

Іноді С++-програмісти до структур, які не містять функцій-членів, застосовують термін POD-**struct**. С++-програмісти тип **class** використовують в основному для визначення форми об'єкта, який містить функції-члени, а тип **struct** – для побудови об'єктів, які містять тільки члени даних. Іноді для опису структури, яка не містить функцій-членів, використовують абревіатуру POD (Plain Old Data).

Порівняння структур з класами. Той факт, що і структури, і класи мають практично однакові можливості, створює враження зайвості. Багато новачків у програмуванні мовою С++ дивуються, чому у ньому існує таке очевидне дублювання. Нерідко доводиться чути пропозиції відмовитися від непотрібного ключового слова (**class** або **struct**) і залишити тільки одне з них.

Відповідь на цю низку міркувань знаходимо, насамперед, у походженні мови програмування С++ від мови С, а також у намірі зберегти мову програмування С++ сумісною з мовою С. Відповідно до сучасного визначення С++-стандартна С-структура одночасно є абсолютно законною С++-структурою. У мові С, яка не містить ключових слів **public** або **private**, всі члени структури є відкритими. Ось чому і члени С++-структур за замовчуванням є відкритими (а не закритими). Оскільки конструкція типу **class** спеціально розроблена для підтримки інкапсуляції, то є певний сенс у тому, щоб за замовчуванням її члени були закритими. Отже, щоб уникнути несумісності з мовою С у цьому питанні, аспекти доступу, що діють за замовчуванням, змінювати було неможливо, тому і вирішено було додати нове ключове слово. Але в перспективі можна сподіватися на дещо вагомішу причину відділення структур від класів. Позаяк тип **class** синтаксично відокремлений від типу **struct**, то визначення класу цілком відкрите для еволюційних змін, які синтаксично можуть виявитися несумісними з С-подібними структурами. Оскільки ми маємо справу з двома окремими типами, то майбутній напрям розвитку мови програмування С++ не обтяжується "моральними зобов'язаннями", пов'язаними з сумісністю з С-структурами.

На завершення цієї теми зазначимо таке. Структура визначає тип класу. Отже, структура є класом. На цьому наполягав винахідник мови програмування С++ Б'єрн Страуструп. Він вважав: якщо структура і класи будуть більш-менш еквівалентними, то перехід від мови С до мови програмування С++ стане простішим. Історія розвитку мови С++ довела його правоту!

2.5. Об'єднання та класи – споріднені типи

Той факт, що структури і класи – споріднені, зазвичай нікого не дивує; проте Ви можете здивуватися, дізнавшись, що об'єднання також пов'язані з класами "близькими відносинами". Згідно з визначенням мови програмування C++, об'єднання – це, по суті, аналогічний клас, у якому всі члени даних зберігаються в одній і тій самій області[1]. Об'єднання може містити конструктор і деструктор, а також функції-члени. Звичайно ж, члени об'єднання за замовчуванням відкриті (**public**), а не закриті (**private**).

Розглянемо код програми, у якій об'єднання використовують для відображення символів, що становлять вміст старшого і молодшого байтів короткого цілочисельного значення (передбачається, що короткі цілочисельні значення займають в пам'яті комп'ютера два байти).

Код програми 2.8. Демонстрація механізму створення класу на основі об'єднання

```
#include <vcl>
#include <iostream>                    // Для потокового введення-виведення
#include <conio>                       // Для консольного режиму роботи
using namespace std;                   // Використання стандартного простору імен

union myUnion {                        // Оголошення типу об'єднання
    myUnion(short int a);              // Відкриті члени за замовчуванням
    void Show();
    short int c;
    char ch[2];
};

// Оголошення параметризованого конструктора
myUnion::myUnion(short int a)
{
    c = a;
}

// Відображення символів, які становлять значення типу short int.
void myUnion::Show()
{
    cout << ch[0] << " " << ch[1] << endl;
}

int main()
{
    myUnion Obj(1000); // Створення об'єкта об'єднання та надання йому числового знач.
    Obj.Show();

    getch(); return 0;
}
```

[1] Таким чином, об'єднання також визначає тип класу.

Подібно до структури, С++-об'єднання також виникло від свого С-попередника. Але у мові програмування С++ воно має ширші "класові" можливості. Проте тільки те, що мова програмування С++ наділяє "свої" об'єднання могутнішими засобами і більшою гнучкістю, зовсім не означає, що Ви неодмінно повинні їх використовувати. Якщо Вас цілком влаштовує об'єднання з традиційним стилем запису, то Ви можете саме таким його і використовувати. Але у випадках, коли можна інкапсулювати ці об'єднання разом з функціями, які їх обробляють, все ж таки варто скористатися С++-можливостями, що додасть Вашій програмі додаткові переваги.

2.6. Поняття про вбудовані функції

Перед тим, як продовжити засвоєння механізмів роботи з класами, дамо невелике, але важливе пояснення. Воно не належить конкретно до об'єктно-орієнтованого програмування, але є дуже корисним засобом мови програмування С++, яке достатньо часто використовують у визначеннях класів. Йдеться про *вбудовану функцію* (**inline** function), або підставну функцію. *Вбудованою* називається *функція*, програмний код якої підставляється в те місце рядка програми, з якого вона викликається, тобто виклик такої функції замінюється її кодом. Існує два способи створення вбудованої функції. Перший полягає у використанні модифікатора **inline**. Наприклад, щоби створити вбудовану функцію Fun(), яка повертає **int**-значення і не приймає жодного параметра, достатньо оголосити її так:

```
inline int Fun()
{
	//...
}
```

Модифікатор **inline** повинен передувати усім решта настанов оголошення самої функції.

Причиною існування вбудованих функцій є ефективність їх використання. Адже під час кожного виклику звичайної функції виконається деяка послідовність настанов, пов'язаних з обробленням самого виклику, що містить занесення її аргументів у стек, чи поверненням їх з функції. В деяких випадках значна кількість циклів центрального процесора використовується саме для виконання цих дій. Але, якщо функція вбудовується в рядок програми, то таких системних витрат просто немає, і загальна швидкість виконання програми зростає. Якщо ж вбудована функція виявляється великою, то загальний розмір програми може істотно збільшитися. Тому краще як вбудовані використовувати тільки маленькі функції, а ті, що є більшими, оформляти у вигляді звичайних функцій.

Продемонструємо механізм використання вбудованої функції на прикладі такої програми.

Код програми 2.9. Демонстрація механізму використання вбудованої функції

```
#include <vcl>
#include <iostream>                // Для потокового введення-виведення
#include <conio>                   // Для консольного режиму роботи
```

```
using namespace std;                // Використання стандартного простору імен

class myClass {                     // Оголошення класового типу
    int c;                          // Закритий член за замовчуванням
public:
    int Put();
    void Get(int d);
};

inline int myClass::Put()
{
    return c;
}

inline void myClass::Get(int d)
{
    c = d;
}

int main()
{
    myClass Obj;     // Створення об'єкта класу

    Obj.Get(10);
    cout << "c= " << Obj.Put() << endl;

    getch(); return 0;
}
```

У цьому коді програми замість виклику функцій Put() і Get() підставляють їх код. Так, у функції **main**() рядок

```
Obj.Get(10);
```

функціонально еквівалентний такій настанові присвоєння:

```
Obj.c = 10;
```

Оскільки змінна c за замовчуванням закрита у межах класу myClass, то цей рядок не може реально існувати в коді функції **main**(), але за рахунок вбудованої функції Get() досягнуто того ж результату, одночасно позбавившись витрат системних ресурсів, взаємопов'язаних з викликом функції.

Важливо розуміти, що насправді використання модифікатора **inline** є *запитом,* а не *командою,* за якою компілятор згенерує вбудований (**inline**-) код. Існують різні ситуації, які можуть не дати змоги компіляторові задовольнити наш запит. Ось декілька прикладів:

- деякі компілятори не генерують вбудованого коду, якщо відповідна функція містить цикл, конструкцію **switch** або настанову **goto**;
- найчастіше вбудованими не можуть бути рекурсивні функції;
- як правило, механізм вбудовування "не проходить" для функцій, які містять статичні (**static**) змінні.

Нео! хідноапам'ятати!а*Обмеження на використання вбудованих функцій залежать від конкретної реалізації компілятора системи. Тому, щоб дізнатися, які обмеження є у Вашому випадку, зверніться до документації, що додається до Вашого компілятора.*

Використання вбудованих функцій у визначенні класу. Існує ще один механізм реалізації вбудованої функції. Він полягає у визначенні коду програми для функції-члена класу в самому оголошенні класу. Будь-яка функція, що визначається в оголошенні класу, автоматично стає вбудованою. У цьому випадку необов'язково передувати її оголошенню ключовим словом **inline** не потрібно. Наприклад, наведену вище програму можна переписати у такому вигляді.

Код програми 2.10. Демонстрація механізму використання вбудованих функцій у визначенні класу

```
#include <vcl>
#include <iostream>           // Для потокового введення-виведення
#include <conio>              // Для консольного режиму роботи
using namespace std;          // Використання стандартного простору імен

class myClass {               // Оголошення класового типу
	int c;                    // Закритий член за замовчуванням
public:
	// Автоматично вбудовані функції
	int Put()          { return c; }
	void Get(int d)    { c = d; }
};

int main()
{
	myClass Obj;     // Створення об'єкта класу

	Obj.Get(10);
	cout << "c= " << Obj.Put() << endl;
	getch(); return 0;
}
```

У цьому коді програми функції Put() і Get() визначені всередині оголошення класу myClass, тобто вони автоматично є вбудованими.

Зверніть увагу на те, як виглядають коди функцій, визначених усередині класу myClass. Для невеликих за обсягом функцій таке представлення коду програми відображає звичайний стиль форматування програм мовою С++. Проте ці функції можна відформатувати ще й так:

```
// Альтернативний запис вбудованих функцій у визначенні класу
class myClass { // Оголошення класового типу
	int c;     // Закритий член за замовчуванням
public:
	// Вбудовані функції
	int Put()
	{
```

```
            return c;
        }

        void Get(int d)
        {
            c = d;
        }
};
```

У загальному випадку невеликі функції (які подано у наведеному вище прикладі) визначаються в оголошенні класу. Ця домовленість стосується і решти прикладів цього навчального посібника.

Варто знати! *Визначення невеликих функцій-членів класу в оголошенні класу – звичайна практика в програмуванні мовою C++. Тут проблема навіть не в механізмі автоматичного вбудовування, а просто в зручності запису коду функції. Навряд чи Вам трапиться в професійних програмах ситуація, коли короткі функції-члени визначалися поза їх класом.*

2.7. Особливості організації масивів об'єктів

Масиви об'єктів можна організувати так само, як і створюються масиви значень стандартних типів. Наприклад, у наведеному нижче коді програми створюється клас displayClass, який містить значення розширення для різних режимів роботи монітора. У функції **main**() створюється масив для зберігання трьох об'єктів типу displayClass, а доступ до них, які є елементами цього масиву, здійснюється за допомогою звичайної процедури індексування елементів масиву.

Код програми 2.11. Демонстрація механізму організації масиву об'єктів

```
#include <vcl>
#include <iostream>             // Для потокового введення-виведення
#include <conio>                // Для консольного режиму роботи
using namespace std;            // Використання стандартного простору імен

enum resolution {low, medium, high};

class displayClass {            // Оголошення класового типу
        int width;
        int height;
        resolution res;
public:
        void Set(int w, int h)      {width = w; height = h; }
        void Get(int &w, int &h)    {w = width; h = height; }
        void Show(resolution r)     {res = r; }
        resolution getRes()         {return res; }
};

char names[3][9] = { "Низький", "Середній", "Високий"};

int main()
```

```
{
	displayClass Monitor[3];

	Monitor[0].Show(low);
	Monitor[0].Set(640, 480);

	Monitor[1].Show(medium);
	Monitor[1].Set(800, 600);

	Monitor[2].Show(high);
	Monitor[2].Set(1600, 1200);

	cout << "Можливі режими відображення даних: " << endl;

	int w, h;
	for(int i=0; i<3; i++) {
		cout << names[Monitor[i].getRes()] << ": ";
		Monitor[i].Get(w, h);
		cout << w << " x " << h << endl;
	}

	getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Можливі режими відображення даних:
Низький: 640 x 480
Середній: 800 x 600
Високий: 1600 x 1200
```

Зверніть увагу на використання двовимірного символьного масиву names для перетворення перерахованого значення в еквівалентний символьний рядок. В усіх перерахунках, які не містять безпосередньо заданої ініціалізації, перша константа має значення 0, друга – значення 1 і т.д. Отже, значення, що повертається функцією getRes(), можна використовувати для індексації елементів масиву names, що дає змогу вивести на екран відповідну назву режиму відображення.

Багатовимірні масиви об'єктів індексуються так само, як і багатовимірні масиви значень інших типів.

Ініціалізація масивів об'єктів. Якщо клас містить параметризований конструктор, то масив об'єктів такого класу можна ініціалізувати. Наприклад, у наведеному нижче коді програми використовується клас myClass і параметризований масив об'єктів типу array цього класу, що ініціалізується конкретними значеннями.

Код програми 2.12. Демонстрація механізму ініціалізації масиву об'єктів

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен

class myClass {              // Оголошення класового типу
```

```
        int a;
public:
        myClass(int b)     { a = b; }
        double Put()       { return a; }
};

int main()
{
        myClass array[4] = { -1, -2, -3, -4 };

        for(int i=0; i<4; i++) cout << "array[" << i << "]= " << array[i].Put() << endl;
        getch(); return 0;
}
```

Результати виконання цієї програми

```
array[0]= -1
array[1]= -2
array[2]= -3
array[3]= -4
```

підтверджують, що конструктору myClass дійсно були передані значення від -1 до -4. Насправді синтаксис ініціалізації масиву, виражений рядком

```
myClass array[4] = { -1, -2, -3, -4 };,
```

є скороченим варіантом такого (довшого) формату:

```
myClass array[4] = { myClass(-1), myClass(-2), myClass(-3), myClass(-4)};
```

Формат ініціалізації, представлений у наведеній вище програмі, використовується програмістами частіше, ніж його довший еквівалент, проте необхідно пам'ятати, що він працює для масивів таких об'єктів, конструктор яких приймає тільки один аргумент. При ініціалізації масиву об'єктів, конструктор яких приймає декілька аргументів, необхідно використовувати довший формат ініціалізації. Розглянемо такий приклад.

Код програми 2.13. Демонстрація механізму ініціалізації масиву об'єктів параметризованим конструктором з декількома аргументами

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен

class myClass {              // Оголошення класового типу
        int a, b;
public:
        myClass(int c, int d) { a = c; b = d; }
        int PutA() { return a; }
        int PutB() { return b; }
};

int main()
{
```

```
    myClass array[4][2] = {
        myClass(1, 2), myClass(3, 4),
        myClass(5, 6), myClass(7, 8),
        myClass(9, 10), myClass(11, 12),
        myClass(13, 14), myClass(15, 16)
    };

    for(int i=0; i<4; i++)
        for(int j=0; j<2; j++) {
            cout << "array[" << i << "," << j << "] ==> a= ";
            cout << array[i][j].PutA() << "; b= ";
            cout << array[i][j].PutB() << endl;
        }
    getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
array[0,0] ==> a= 1; b=2
array[0,1] ==> a= 3; b=4
array[1,0] ==> a= 5; b=6
array[1,1] ==> a= 7; b=8
array[2,0] ==> a= 9; b=10
array[2,1] ==> a= 11; b=12
array[3,0] ==> a= 13; b=14
array[3,1] ==> a= 15; b=16
```

У наведеному вище прикладі параметризований конструктор класу myClass приймає два аргументи. У основній функції **main**() оголошується та ініціалізується масив array об'єктів шляхом безпосередніх викликів конструктора myClass(). Для ініціалізації масиву можна завжди використовувати довгий формат ініціалізації, навіть якщо об'єкт приймає тільки один аргумент (коротка форма просто зручніша для застосування).

2.8. Особливості використання покажчиків на об'єкти

Як було показано в попередньому розділі, доступ до структури можна отримати безпосередньо або через покажчик на цю структуру. Аналогічно можна звертатися і до об'єкта: безпосередньо (як в усіх попередніх прикладах) або за допомогою покажчика на об'єкт. Щоб отримати доступ до окремого члена об'єкта виключно "силами" самого об'єкта, використовують оператор "крапка". А якщо для цього слугує покажчик на цей об'єкт, то необхідно використовувати оператор "стрілка"[1].

Щоб оголосити покажчик на об'єкт, використовується аналогічний синтаксис як і у разі оголошення покажчиків на значення інших типів. У наведеному нижче коді програми створюється простий клас pClass, визначається об'єкт цього класу Obj і оголошується покажчик на об'єкт типу pClass з іменем p. У наведеному нижче прикладі показано, як можна безпосередньо отримати доступ до об'єкта Obj і як

---

[1] Застосування операторів "крапка" і "стрілка" для об'єктів відповідає їх застосуванню для структур і об'єднань.

використовувати для цього покажчик (у цьому випадку ми маємо справу з непрямим доступом).

Код програми 2.14. Демонстрація механізму використання покажчика на об'єкт

```
#include <vcl>
#include <iostream>            // Для потокового введення-виведення
#include <conio>               // Для консольного режиму роботи
using namespace std;           // Використання стандартного простору імен

class pClass {                 // Оголошення класового типу
	int num;
public:
	void Set(int n)	{ num = n; }
	void Show()	{ cout << "num= " << num << endl; }
};

int main()
{
	pClass Obj, *p;	// Створення об'єкта класу і покажчика на нього.

	Obj.Set(1);	// Отримуємо прямий доступ до об'єкта Obj.
	Obj.Show();

	p = &Obj;	// Присвоюємо покажчику p адресу об'єкта Obj.
	p->Show();	// Отримуємо доступ до об'єкта Obj за допомогою покажчика.
	getch(); return 0;
}
```

Зверніть увагу на те, що адреса об'єкта Obj отримується шляхом використання оператора посилання "&", що цілком відповідає механізму отриманню адреси для змінних будь-якого іншого типу.

Як було зазначено в попередніх розділах, інкрементація або декрементація покажчика відбувається так, щоб завжди вказувати на наступний або попередній елемент базового типу. Те саме відбувається і під час інкрементування або декрементування покажчика на об'єкт: він вказуватиме на наступний або попередній об'єкт. Щоби проілюструвати цей механізм, дещо модифікуємо наведену вище програму. Тепер замість одного об'єкта Obj оголосимо двоелементний масив Obj типу pClass. Зверніть увагу на те, як інкрементується та декрементується покажчик p для доступу до двох елементів цього масиву.

Код програми 2.15. Демонстрація механізму інкрементування та декрементування покажчика на об'єкт

```
#include <vcl>
#include <iostream>            // Для потокового введення-виведення
#include <conio>               // Для консольного режиму роботи
using namespace std;           // Використання стандартного простору імен

class pClass {                 // Оголошення класового типу
	int num;
public:
```

```
        void Set(int n)    { num = n; }
        void Show()        { cout << "num= " << num << endl; }
};

int main()
{
        pClass Obj[2],*p;
        Obj[0].Set(10);    // Прямий доступ до об'єктів
        Obj[1].Set(20);

        p = &Obj[0];       // Отримуємо покажчик на перший елемент.
        p->Show();         // Відображаємо значення елемента Obj[0] за допомогою покажчика.

        p++;               // Переходимо до наступного об'єкта.
        p->Show();         // Відображаємо значення елемента Obj[1] за допомогою покажчика.

        p--;               // Повертаємося до попереднього об'єкта.
        p->Show();         // Знову відображаємо значення елемента Obj[0].

        getch(); return 0;
}
```

Ось як виглядають результати виконання цієї програми:

```
num= 10
num= 20
num= 10
```

Про те, що покажчики на об'єкти мають важливе значення в реалізації одного з найважливіших принципів мови програмування C++ – поліморфізму, Ви зрозумієте з подальшого висвітлення матеріалу у цьому навчальному посібнику.

Посилання на об'єкти. На об'єкти можна посилатися так само, як і на значення будь-якого іншого типу. Для цього не існує ніяких спеціальних настанов або обмежень. Але, як буде показано в наступному розділі, використання посилань на об'єкти дає змогу справлятися з деякими специфічними проблемами, які можуть трапитися під час роботи з класами.

Розділ 3. ОРГАНІЗАЦІЯ КЛАСІВ І ОСОБЛИВОСТІ РОБОТИ З ОБ'ЄКТАМИ

У цьому розділі розглянемо деякі особливості організації класів, спробуємо познайомитися з "дружніми" функціями, перевизначенням конструкторів, а також з механізмом передачі об'єктів функціям і їх повернення ними. Окрім цього, дізнаємося про наявність *конструктора копії*, який використовується при створенні копії об'єкта. Завершує цей розділ опис ключового слова **this**.

3.1. Поняття про функції-"друзі" класу

Технологія об'єктно-орієнтованого програмування дає змогу організувати доступ до закритих членів класу функціями, які не є його членами. Для цього достатньо оголосити ці функції *дружніми* до цього класу. Щоб зробити функцію "другом" класу, потрібно помістити її прототип в **public**-розділ оголошення класу і попередити його ключовим словом **friend**. Наприклад, наведений нижче фрагмент коду функції Fun() оголошується "другом" класу myClass:

```
class myClass {           // Оголошення класового типу
      //...
public:
      friend void Fun(myClass obj);    // "Дружня" функція класу
      //...
};
```

Як бачимо, ключове слово **friend** передує решті частини прототипу функції Fun(), надає їй як не члену класу доступ до його закритих членів. Одна і та ж функція може бути "другом" декількох класів. Розглянемо приклад, у якому "дружня" функція використовується для доступу до закритих членів класу myClass.

Код програми 3.1. Демонстрація механізму використання "дружньої" функції для доступу до закритих членів класу

```
#include <vcl>
#include <iostream>     // Для потокового введення-виведення
#include <conio>        // Для консольного режиму роботи
using namespace std; // Використання стандартного простору імен

class myClass {           // Оголошення класового типу
      int a, b;
public:
      // Оголошення параметризованого конструктора
      myClass(int _a, int _b) { a = _a; b = _b; }

      friend int Sum(myClass obj); // Функція Sum() – "друг" класу myClass.
};
```

```
// Функція Sum() не є членом ні одного класу.
int Sum(myClass obj)
{
	/* Оскільки функція Sum() – "друг" класу myClass, то вона має
	право на прямий доступ до його членів-даних a і b */
	return obj.a + obj.b;
}

int main()
{
	myClass Obj(3, 4);		// Створення об'єкта класу

	cout << "Sum= " << Sum(Obj) << endl;

	getch(); return 0;
}
```

У наведеному вище прикладі функція Sum() не є членом класу myClass. Проте вона має повний доступ до **private**-членів класу myClass. Зокрема, вона може безпосередньо використовувати значення obj.a і obj.b. Зверніть також увагу на те, що функція Sum() викликається звичайним способом, тобто без прив'язування до імені об'єкта (і без використання оператора "крапка"). Оскільки вона не є функцією-членом класу, то під час виклику її не потрібно кваліфікувати з вказанням імені об'єкта[1]. Зазвичай "дружній" функції як параметр передається один або декілька об'єктів класу, для яких вона є "другом". Робиться це так само як і у випадку передачі параметра функції Sum().

Незважаючи на те, що у наведеному вище прикладі не отримуємо ніякої користі з оголошення "другом" функції Sum(), а не членом класу myClass, існують певні обставини, при яких статус "дружньої" функції класу має велике значення. По-перше, функції-"друзі" є корисними для перевизначення операторів певних типів. По-друге, функції-"друзі" спрощують створення деяких функцій введення-виведення. Усі ці питання розглядатимемо згодом у цьому навчальному посібнику.

Третя причина частого використання функцій-"друзів" полягає у тому, що в деяких випадках два (або більше) класи можуть містити члени, які перебувають у взаємному зв'язку з іншими частинами програми. Наприклад, у нас є два різні класи, які під час виникнення певних подій відображають на екрані "спливаючі" повідомлення. Інші частини програми, які призначені для виведення даних на екран, повинні знати, чи є "спливаюче" повідомлення активним, щоб випадково не перезаписати його. Для уникнення цього у кожному класі можна створити функцію-члена, що повертає значення, за якою робляться висновки про те, є повідомлення активним чи ні. Однак перевірка цієї умови вимагатиме додаткових витрат (тобто двох викликів функцій замість одного). Якщо статус "спливаючого" повідомлення необхідно перевіряти часто, то ці додаткові витрати можуть виявитися відчутними. Проте за допомогою функції, "дружньої" для обох класів, можна безпосередньо перевіряти статус кожного об'єкта, викликаючи для цього тільки одну і ту

[1] Точніше, під час її виклику *не можна* задавати ім'я об'єкта.

саму функцію, яка матиме доступ до обох класів. У таких ситуаціях "дружня" функція класу дає змогу написати більш ефективний програмний код. Реалізацію зазначеної ідеї продемонструємо на прикладі такої програми.

Код програми 3.2. Демонстрація механізму використання "дружньої" функції для перевірки статусу кожного об'єкта

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен

const int IDLE=0;
const int INUSE=1;

class bClass;                // Випереджувальне оголошення класу

class aClass {               // Оголошення класового типу
    int status;              // IDLE, якщо повідомлення неактивне
                             // INUSE, якщо повідомлення виведене на екран.
    //...
public:
    void Set(int s) { status = s; }
    friend int Put(aClass obi, bClass obj);
};

class bClass {                // Оголошення класового типу
    int status;               // IDLE, якщо повідомлення неактивне
                              // INUSE, якщо повідомлення виведене на екран.
    //...
public:
    void Set(int s) { status = s; }
    friend int Put(aClass obi, bClass obj);
};

// Функція Put() – "друг" для класів aClass і bClass.
int Put(aClass obi, bClass obj)
{
    if(obi.status || obj.status) return 0;
    else return 1;
}

int main()
{
    aClass ObjX;             // Створення об'єкта класу
    bClass ObjY;             // Створення об'єкта класу

    ObjX.Set(IDLE);          // IDLE = 0
    ObjY.Set(IDLE);

    if(Put(ObjX, ObjY)) cout << "Екран вільний" << endl;
```

```
    else cout << "Відображається повідомлення" << endl;

    ObjX.Set(INUSE); // INUSE = 1

    if(Put(ObjX, ObjY)) cout << "Екран вільний" << endl;
    else cout << "Відображається повідомлення" << endl;

    getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Екран вільний.
Відображається повідомлення.
```

Оскільки функція Put() є "другом" як для класу aClass, так і для класу bClass, то вона має доступ до закритого члена status, визначеного в обох класах. Таким чином, стан об'єкта кожного класу одночасно можна перевірити всього одним зверненням до функції Put().

Зверніть увагу на те, що у цьому коді програми використовується *випереджувальне оголошення* (часто його також називають *випереджувальним посиланням*) для класу bClass. Потреба у ньому пов'язана з тим, що оголошення функції Put() у класі aClass використовує посилання на клас bClass ще до його оголошення. Щоб створити випереджувальне оголошення для будь-якого класу, достатньо зробити це так, як показано у наведеному вище коді програми.

"Дружня" функція одного класу може бути членом іншого класу. Переробимо наведену вище програму так, щоби функція Put() стала членом класу aClass. Зверніть увагу на використання оператора дозволу області видимості (або оператора дозволу контексту) під час оголошення функції Put() як "друга" класу bClass.

Код програми 3.3. Демонстрація механізму використання функції – члена одного класу і одночасно "дружньої функції" – для іншого класу

```
#include <vcl>
#include <iostream>         // Для потокового введення-виведення
#include <conio>            // Для консольного режиму роботи
using namespace std;        // Використання стандартного простору імен

const int IDLE=0;
const int INUSE=1;

class bClass;               // Випереджувальне оголошення класу

class aClass {              // Оголошення класового типу
    int status;             // IDLE, якщо повідомлення неактивне
                            // INUSE, якщо повідомлення виведене на екран.
    //...
public:
    void Set(int s) { status = s; }
    int Put(bClass obj);    // тепер це член класу aClass
};
```

```
class bClass {                    // Оголошення класового типу
    int status;                   // IDLE, якщо повідомлення неактивне
                                  // INUSE, якщо повідомлення виведене на екран.
    //...
public:
    void Set(int s) { status = s; }
    friend int aClass::Put(bClass obj); // "Дружня" функція класу
};

// Функція Put() -- член класу aClass і "друг" класу bClass.
int aClass::Put(bClass obj)
{
    if(status || obj.status) return 0;
    else return 1;
}

int main()
{
    aClass ObjX;     // Створення об'єкта класу
    bClass ObjY;     // Створення об'єкта класу

    ObjX.Set(IDLE);  // IDLE = 0
    ObjY.Set(IDLE);

    if(ObjX.Put(ObjY)) cout << "Екран вільний" << endl;
    else cout << "Відображається повідомлення" << endl;

    ObjX.Set(INUSE); // INUSE = 1

    if(ObjX.Put(ObjY)) cout << "Екран вільний" << endl;
    else cout << "Відображається повідомлення" << endl;

    getch(); return 0;
}
```

Оскільки функція Put() є членом класу aClass, то вона має прямий доступ до змінної status об'єктів типу aClass. Отже, як параметр необхідно передавати функції Put() тільки об'єкти типу bClass.

3.2. Особливості перевизначення конструкторів

Незважаючи на те, що конструктори призначені для виконання унікальних дій, проте вони не надто відрізняються від функцій-членів класу інших типів і також можуть піддаватися перевизначенню. Щоб перевизначати конструктор класу, достатньо оголосити його в усіх потрібних форматах і визначити кожну дію, пов'язану з кожним форматом. Наприклад, у наведеному нижче коді програми оголошено клас timerClass, який діє як таймер зворотного відліку. При створенні об'єкта типу timerClass таймеру присвоюється деяке початкове значення часу. При вик-

лику функції Run() таймер здійснює відлік часу у зворотному порядку до нуля, а потім подає звуковий сигнал. У наведеному нижче коді програми конструктор перевизначається тричі, надаючи тим самим можливість задавати час як у секундах (причому або числом, або рядком), так і у хвилинах і секундах (за допомогою двох цілочисельних значень). У цьому коді програми використовується стандартна бібліотечна функція **clock**(), яка повертає кількість сигналів, прийнятих від системного годинника з моменту початку виконання програми. Прототип цієї функції має такий вигляд:

clock_t clock();

Тип **clock_t** є різновидом довгого цілочисельного типу. Операція ділення значення, що повертається функцією **clock**(), на значення **CLOCKS_PER_SEC** дає змогу перетворити отриманий результат у секунди. Як прототип для функції **clock**(), так і визначення константи **CLOCKS_PER_SEC** належать заголовку **<ctime>**.

Код програми 3.4. Демонстрація механізму використання перевизначених конструкторів

```
#include <vcl>
#include <iostream>            // Для потокового введення-виведення
#include <conio>               // Для консольного режиму роботи
#include <ctime>               // Для використання системного часу і дати
using namespace std;           // Використання стандартного простору імен

class timerClass {        // Оголошення класового типу
	int s;
public:
	// Задавання секунд у вигляді рядка
	timerClass(char *t) { s = atoi(t); }

	// Задавання секунд у вигляді цілого числа
	timerClass(int t) { s = t; }

	// Час, який задається в хвилинах і секундах
	timerClass(int xv, int sec) { s = xv*60 + sec; }

	// Час, який задається в годинах, хвилинах і секундах
	timerClass(int hod, int xv, int sec) { s = 60*(hod*60 + xv) + sec; }

	void Run(); // Таймер відліку часу
};

void timerClass::Run()
{
	clock_t t1 = clock();
	while((clock()/CLOCKS_PER_SEC – t1/CLOCKS_PER_SEC)< s);
	cout << "\a"; // Подання звукового сигналу
}

int main()
```

```
{
    timerClass ObjA(10), ObjB("20"), ObjC(1, 10), ObjD(0, 2, 8);

    ObjA.Run(); // Відлік 10 секунд
    ObjB.Run(); // Відлік 20 секунд
    ObjC.Run(); // Відлік 1 хвилини і 10 секунд
    ObjD.Run(); // відлік 0 годин 2 хвилини і 8 секунд

    getch(); return 0;
}
```

При створенні у функції **main**() об'єктів ObjA, ObjB, ObjC і ObjD класу timerClass він надає члену даних s початкові значення чотирма різними способами, що підтримуються перевизначеними функціями конструкторів. У кожному випадку викликається той конструктор, який відповідає заданому переліку параметрів, і тому правильно ініціалізує "свій" об'єкт.

На прикладі попереднього коду програми нам, можливо, не вдалося оцінили значущість перевизначення функцій конструктора, оскільки тут можна було обійтися єдиним способом задавання тимчасового інтервалу. Але якби була потреба створити бібліотеку класів на замовлення, то нам варто було б передбачити набір конструкторів, які охоплюють найширший спектр різних форматів ініціалізації, тим самим забезпечити інших програмістів найбільш придатними форматами для їх програм. Окрім цього, як буде показано далі, у мові програмування C++ існує атрибут, який робить перевизначені конструктори особливо цінним засобом ініціалізації членів-даних об'єктів.

3.3. Особливості механізму динамічної ініціалізації конструктора

У мові програмування C++ як локальні, так і глобальні змінні можна ініціалізувати у процесі виконання програми. Цей процес іноді називають *динамічною ініціалізацією.* Дотепер у більшості настанов ініціалізації, що були представлені у цьому навчальному посібнику, використовувалися константи. Проте одну і ту ж саму змінну можна також ініціалізувати у процесі виконання програми, використовуючи будь-який C++-вираз, дійсний на момент оголошення цієї змінної. Це означає, що змінну можна ініціалізувати за допомогою інших змінних і/або викликів функцій за умови, що у момент виконання настанови оголошення загальний вираз ініціалізації має дійсне значення. Наприклад, різні варіанти ініціалізації змінних абсолютно допускаються у мові програмування C++:

```
int n = strlen(str);
double arc = sin(theta);
float d = 1.02 * pm/delta;
```

Подібно до простих змінних, об'єкти можна ініціалізувати динамічно під час їх створення. Цей механізм дає змогу створювати об'єкт потрібного типу з використанням інформації, яка стає відомою тільки у процесі виконання програми. Щоб показати, як працює механізм динамічної ініціалізації конструктора, спро-

буємо модифікувати код програми реалізації таймера, який було наведено в попередньому розділі.

Згадаймо, що в першому прикладі коду програми таймера ми не отримали великої переваги від перевизначення конструктора timerClass(), оскільки всі об'єкти цього типу ініціалізувалися за допомогою констант, відомих при компілюванні програми. Але у випадках, коли об'єкт необхідно ініціалізувати у процесі виконання програми, можна отримати істотний виграш від наявності множини різних форматів ініціалізації. Це дає змогу програмісту вибрати з наявних конструкторів той, який найточніше відповідає поточному формату даних.

Наприклад, у версії коду програми таймера для створення двох об'єктів ObjB і ObjC, яку наведено нижче, використовується динамічна ініціалізація конструктора.

Код програми 3.5. Демонстрація механізму динамічної ініціалізації конструктора

```
#include <vcl>
#include <iostream>        // Для потокового введення-виведення
#include <conio>           // Для консольного режиму роботи
#include <ctime>           // Для використання системного часу і дати
using namespace std;       // Використання стандартного простору імен

class timerClass {         // Оголошення класового типу
    int s;
public:
    // Задавання секунд у вигляді рядка
    timerClass(char *t) { s = atoi(t); }

    // Задавання секунд у вигляді цілого числа
    timerClass(int t) { s = t; }

    // Час, який задається в хвилинах і секундах
    timerClass(int xv, int sec) { s = xv*60 + sec; }

    // Час, який задається в годинах, хвилинах і секундах
    timerClass(int hod, int xv, int sec) { s = 60*(hod*60 + xv) + sec; }

    void Run(); // Таймер відліку часу
};

void timerClass::Run()
{
    clock_t t1 = clock();
    while((clock()/CLOCKS_PER_SEC – t1/CLOCKS_PER_SEC)< s);
    cout << "\a"; // Подання звукового сигналу
}

int main()
{
    timerClass ObjA(10);      // Створення об'єкта класу
    ObjA.Run();
```

```
        char str[80];

        cout << "Введіть кількість секунд: ";
        cin >> str;
        timerClass ObjB(str);           // Динамічна ініціалізація конструктора
        ObjB.Run();

        int xv, sec;
        cout << "Введіть хвилини і секунди: "; cin >> xv >> sec;
        timerClass ObjC(xv, sec);       // Динамічна ініціалізація конструктора
        ObjC.Run();

        int hod;
        cout << "Введіть години, хвилини і секунди: "; cin >> hod >> xv >> sec;
        timerClass ObjD(hod, xv, sec);  // Динамічна ініціалізація конструктора
        ObjD.Run();

        getch(); return 0;
    }
```

Як бачимо, об'єкт ObjA створюється, використовуючи механізм ініціалізації цілочисельної константи. Проте основою для побудови об'єктів ObjB і ObjC слугує інформація, яка вводиться користувачем. Оскільки для об'єкта ObjB користувач вводить рядок, то є сенс перевизначити конструктор timerClass() для прийняття рядкової змінної. Об'єкт ObjC також створюється у процесі виконання програми з використанням даних, які вводяться користувачем. Оскільки у цьому випадку час вводиться у вигляді хвилин і секунд, то для побудови об'єкта ObjC логічно використовувати формат конструктора, що приймає два цілочисельні аргументи. Аналогічно створюється об'єкт ObjD, для якого час вводиться у вигляді годин, хвилин і секунд, тобто використовується формат конструктора, що приймає три цілочисельних аргументи. Важко не погодитися з тим, що наявність декількох форматів ініціалізації конструктора позбавляє програміста від виконання додаткових перетворень під час ініціалізації членів-даних об'єктів.

Механізм перевизначення конструкторів сприяє зниженню рівня складності написання коду програми, даючи змогу програмісту створювати об'єкти найбільш природно для своєї програми. Оскільки існує три найпоширеніші способи передачі об'єкту значень тимчасових інтервалів часу, то є сенс потурбуватися про те, щоб конструктор timerClass() був перевизначений для реалізації кожного з цих способів. При цьому перевизначення конструктора timerClass() для прийняття значення, вираженого, наприклад, у днях або наносекундах, навряд чи себе виправдає. Захаращення коду програми конструкторами для оброблення ситуацій, які рідко виникають, як правило, дестабілізаційно впливає на читабельність коду програми.

Вартоа' нати!а*Розробляючи перевизначені конструктори, необхідно достеменно визначитися у тих ситуаціях, які обов'язково будуть використовуватися під час виконання програми, а які можна і нехтувати.*

3.4. Особливості механізму присвоєння об'єктів

Якщо два об'єкти мають однаковий тип (тобто обидва вони – об'єкти одного класу), то значення членів-даних одного об'єкта можна присвоїти іншому. Проте, для виконання операції присвоєння недостатньо, щоб два класи були фізично подібними; імена класів, об'єкти яких беруть участь в операції присвоєння, повинні збігатися. Якщо один об'єкт присвоюється іншому, то за замовчуванням дані першого об'єкта порозрядно копіюються у другий. Механізм присвоєння об'єктів продемонстровано у наведеному нижче коді програми.

Код програми 3.6. Демонстрація механізму присвоєння об'єктів

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен

class myClass {              // Оголошення класового типу
        int a, b;
public:
        myClass()            { a = b = 0; }
        void Set(int c, int d) { a = c; b = d; }
        void Show()          { cout << "a = " << a << "; b = " << b << endl; }
};

int main()
{
        myClass ObjA, ObjB;     // Створення об'єктів класу

        ObjA.Set(10, 20);
        ObjB.Set(0, 0);
        cout << "Об'єкт ObjA до присвоєння:" << endl;
        ObjA.Show();
        cout << "Об'єкт ObjB до присвоєння:" << endl;
        ObjB.Show();
        cout << endl;

        ObjB = ObjA;       // Присвоюємо об'єкт ObjA об'єкту ObjB.

        cout << "Об'єкт ObjA після виконання операції присвоєння:" << endl;
        ObjA.Show();
        cout << "Об'єкт ObjB після виконання операції присвоєння:" << endl;
        ObjB.Show();

        getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Об'єкт ObjA до присвоєння:
a = 10; b = 20
Об'єкт ObjB до присвоєння:
```

```
a = 0; b = 0

Об'єкт ObjA після виконання операції присвоєння:
a = 10; b = 20
Об'єкт ObjB після виконання операції присвоєння:
a = 10; b = 20
```

За замовчуванням усі значення членів-даних з одного об'єкта присвоюються іншому шляхом створення порозрядної копії[1]. Але, як буде показано далі, оператор присвоєння можна перевизначати, визначивши власні операції присвоєння.

***Нео! хідноапам'ятати!** Присвоєння одного об'єкта іншому просто робить значення їх членів-даних однаковими, але ці два об'єкти залишаються абсолютно незалежними. Отже, подальше модифікування даних одного об'єкта не робить ніякого впливу на дані іншого.*

3.5. Особливості механізму передачі об'єктів функціям

Об'єкт можна передати функції так само, як і змінну будь-якого іншого типу даних. Об'єкти передаються функціям шляхом використання звичайного С++-погодження про передачу параметрів за значенням. Згідно з цим погодженням, функції передається не сам об'єкт, а його копія. Це означає, що зміни, внесені в об'єкт-копію у процесі виконання функції, не роблять ніякого впливу на вхідний об'єкт, який використовується як аргумент для функції. Цей механізм продемонстровано у наведеному нижче коді програми.

Код програми 3.7. Демонстрація механізму передачі об'єктів функціям

```
#include <vcl>
#include <iostream>         // Для потокового введення-виведення
#include <conio>            // Для консольного режиму роботи
using namespace std;        // Використання стандартного простору імен

class myClass {             // Оголошення класового типу
	int c;
public:
	myClass()             { c = 0; }
	void Set(int _c)      { c = _c; }
	void Show(char *s)    { cout << s << c << endl; }
};

void Fun(myClass obj)       // Визначення функції не члена класу
{
	obj.Show("t2= ");     // Виведення числа 10.
	obj.Set(100);         // Встановлює тільки локальну копію.
	obj.Show("t3= ");     // Виведення числа 100.
}
```

[1] Іншими словами, створюється точний дублікат об'єкта.

```
int main()
{
	myClass Obj;	// Створення об'єкта класу

	Obj.Set(10);
	Obj.Show("t1= ");	// Виведення числа 10.

	Fun(Obj);	// Передача об'єкта функції не члена класу

	Obj.Show("t4= ");	// Як і раніше, виводиться число 10, проте
		// значення змінної "і" не змінилося.

	getch(); return 0;
}
```

Ось як виглядають результати виконання цієї програми.

```
t1= 10
t2= 10
t3= 100
t4= 10
```

Як підтверджують ці результати, модифікування об'єкта obj у функції Fun() не впливає на об'єкт Obj у функції **main**().

3.5.1. Конструктори, деструктори і передача об'єктів

Хоча передача функціям нескладних об'єктів як аргументів – достатньо проста процедура, проте при цьому можуть відбуватися непередбачені події, які мають відношення до конструкторів і деструкторів. Щоб розібратися у цьому, розглянемо такий код програми.

Код програми 3.8. Демонстрація механізму використання конструкторів, деструкторів при передачі об'єктів

```
#include <vcl>
#include <iostream>	// Для потокового введення-виведення
#include <conio>	// Для консольного режиму роботи
using namespace std;	// Використання стандартного простору імен

class myClass {	// Оголошення класового типу
	int n;
public:
	myClass(int _n)	{ n = _n; cout << "Створення об'єкта" << endl; }
	~myClass()	{ cout << "Руйнування об'єкта" << endl; }
	int Put()	{ return n; }
};

void Get(myClass obj)
{
	cout << "n= " << obj.Put() << endl;
}
```

```
int main()
{
    myClass ObjA(10);       // Створення об'єкта класу

    Get(ObjA);

    getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Створення об'єкта
n= 10
Руйнування об'єкта
Руйнування об'єкта
```

Як бачимо, тут здійснюється *одне* звернення до функції конструктора (при створенні об'єкта ObjA), але чомусь відбувається *два* звернення до функції деструктора. Давайте з'ясуємо, у чому тут проблема.

При передачі об'єкта функції створюється його копія (і ця копія стає параметром у функції). Створення копії означає появу нового об'єкта. Коли виконання функції завершується, копія аргумента (тобто параметр) руйнується. Тут виникає відразу два запитання. По-перше, чи викликається конструктор об'єкта при створенні копії? По-друге, чи викликається деструктор об'єкта під час руйнування копії? Відповіді на ці запитання можуть здивувати Вас.

Оскільки під час виклику функції створюється копія аргумента, то звичайний конструктор об'єкта *не викликається*. Натомість викликається *конструктор копії* об'єкта, який визначає, як має створюватися копія об'єкта[1]. Але, якщо в класі безпосередньо не визначено конструктор копії, то мова програмування C++ надає його за замовчуванням. Конструктор копії за замовчуванням створює побітову (тобто однакову) копію об'єкта. Оскільки звичайний конструктор використовують для ініціалізації тільки деяких даних об'єкта, то він не повинен викликатися для створення копії вже наявного об'єкта. Такий виклик змінив би його вміст. При передачі об'єкта функції потрібно використовувати поточний стан об'єкта, а не його початковий стан.

Однак, коли функція завершує свою роботу, то руйнується копія об'єкта, яка використовується як аргумент, для чого викликається деструктор цього об'єкта. Потреба виклику деструктора пов'язана з виходом об'єкта з області видимості його функцією, у якій він використовується. Саме тому попередня програма виводила два звернення перед зверненням до деструктора. Перше відбулося при виході з області видимості параметра функції Put(), а друге – під час руйнування об'єкта ObjA у функції **main**() після завершення роботи коду програми.

Отже, коли об'єкт передається функції як аргумент, звичайний конструктор не викликається. Замість нього викликається конструктор копії, який за замовчуванням створює побітову (тобто, однакову) копію цього об'єкта. Але коли ця побітова копія об'єкта руйнується (зазвичай при виході за межі області видимості його після завершення роботи функції), то обов'язково викликається деструктор.

[1] Як створити конструктор копії, буде показано далі в цьому розділі.

3.5.2. Потенційні проблеми, які виникають при передачі об'єктів

Як зазначалося вище, об'єкти передаються функціям "за значенням", тобто за допомогою звичайного C++-механізму передачі параметрів, який теоретично захищає аргумент і ізолює його від параметра, що приймається. Незважаючи на ці обставини, тут все-таки можливий побічний ефект або навіть загроза для "життя" об'єкта, який використовується як аргумент. Наприклад, якщо оригінальний об'єкт, який потім використовується як аргумент, вимагає динамічного виділення області пам'яті та звільняє цю пам'ять шляхом його руйнування, то локальна копія цього об'єкта під час виклику деструктора звільнить ту ж саму область пам'яті, яка була виділена оригінальному об'єкту. Поява такої ситуації стає потенційною проблемою, оскільки оригінальний об'єкт все ще використовує цю (вже звільнену) область пам'яті. Описана ситуація робить оригінальний об'єкт "збитковим" і, по суті, непридатним для використання. Для розуміння сказаного вище розглянемо таку навчальну програму.

Код програми 3.9. Демонстрація механізму появи проблеми, яка виникає при передачі об'єктів функціям, у яких динамічно виділяється та звільняється область пам'яті

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен

class myClass {              // Оголошення класового типу
    int *p;
public:
    myClass(int c);
        { p = new int; *p = c;
        cout << "Виділення р-пам'яті звичайним конструктором" << endl; }
    ~myClass();
        { delete p; cout << "Звільнення р-пам'яті" << endl; }
    int Put() { return *p; }
};

// У процесі виконання цієї функції якраз і виникає проблема.
void Get(myClass obj) // Звичайна передача об'єкта
{
    cout << "*p= " << obj.Put() << endl;
}

int main()
{
    myClass ObjA(10);        // Створення об'єкта класу

    Get(ObjA);
    getch(); return 0;
}
```

Ось як виглядають результати виконання цієї програми.

```
Виділення p-пам'яті звичайним конструктором.
*p= 10
Звільнення p-пам'яті.
Звільнення p-пам'яті.
```

Ця програма містить принципову помилку, а саме: при створенні у функції **main**() об'єкта ObjA виділяється область пам'яті, адреса якої присвоюється покажчику ObjA.p. При передачі функції Get() об'єкт ObjA побітово копіюється в параметр obj. Це означає, що обидва об'єкти (ObjA і obj) матимуть однакове значення для покажчика p. Іншими словами, в обох об'єктах (в оригіналі та його копії) член даних p вказуватиме на одну і ту саму динамічно виділену область пам'яті. Після завершення роботи функції Get() об'єкт obj руйнується за допомогою деструктора. Деструктор звільняє область пам'яті, яка адресується покажчиком obj.p. Але ж ця (вже звільнена) область пам'яті – та ж сама область, на яку все ще вказує член даних (початкового об'єкта) ObjA.p! Тобто, як на перший погляд – виникає серйозна помилка.

Насправді справи йдуть ще гірше. Після завершення роботи коду програми руйнується об'єкт ObjA і динамічно виділена (ще під час його створення) пам'ять звільняється повторно. Йдеться про те, що звільнення однієї і тієї ж самої області динамічно виділеної пам'яті удруге вважається невизначеною операцією, яка, як правило (залежно від того, яка система динамічного розподілу пам'яті реалізована), спричиняє непоправну помилку.

Одним із шляхів вирішення проблеми, пов'язаної з руйнуванням (ще потрібних) даних деструктором об'єкта, який є параметром функції, полягає не в передачі самого об'єкта, а в передачі покажчика на нього або посилання. У цьому випадку копія об'єкта не створюється; отже, після завершення роботи функції деструктор не викликається. Ось як виглядає, наприклад, один із способів виправлення попереднього коду програми.

Код програми 3.10. Демонстрація механізму вирішення проблеми при передачі об'єктів функціям, у яких динамічно виділяється та звільняється область пам'яті

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен

class myClass {              // Оголошення класового типу
	int *p;
public:
	myClass(int c);
		{ p = new int; *p = c;
		cout << "Виділення p-пам'яті звичайним конструктором" << endl; }
	~myClass();
		{ delete p; cout << "Звільнення p-пам'яті" << endl; }
	int Put() { return *p; }
};

// Ця функція НЕ створює проблем.
```

```
void Get(myClass &obj)          // Передача об'єкта за посиланням
{
        cout << "*p= " << obj.Put() << endl;
}

int main()
{
        myClass ObjA(10);       // Створення об'єкта класу

        Get(ObjA);

        getch(); return 0;
}
```

Оскільки об'єкт obj тепер передається за посиланням, то побітова копія аргумента не створюється, а отже, об'єкт не виходить з області видимості після завершення роботи функції Get(). Результати виконання цієї версії коду програми виглядають набагато краще від попередніх:

```
Виділення p-пам'яті звичайним конструктором
*p= 10
Звільнення p-пам'яті
```

Як бачимо, тут деструктор викликається тільки один раз, оскільки при передачі за посиланням аргумента функції Get() побітова копія об'єкта не створюється.

Передача об'єкта за посиланням – типове вирішення описаної вище проблеми, але тільки у випадках, коли утворена ситуація дає змогу прийняти таке рішення, що буває далеко не завжди. На щастя, є більш загальне рішення: можна створити власну версію конструктора копії об'єкта. Це дасть змогу точно визначити, як саме потрібно створювати побітову копію об'єкта і тим самим уникнути описаних вище проблем. Але перед тим, як займемося конструктором копії, є сенс розглянути ще одну ситуацію, у вирішенні якої ми також можемо отримати певний виграш завдяки створенню конструктора копії.

3.6. Особливості механізму повернення об'єктів функціями

Якщо об'єкти можна передавати функціям, то з таким самим успіхом функції можуть повертати об'єкти. Щоби функція могла повернути об'єкт, по-перше, необхідно оголосити об'єкт, який повертається нею, типом відповідного класу. По-друге, потрібно забезпечити повернення об'єкта цього типу за допомогою звичайної настанови **return**. Розглянемо приклад функції, яка повертає об'єкт.

Код програми 3.11. Демонстрація механізму повернення об'єкта функцією

```
#include <vcl>
#include <iostream>           // Для потокового введення-виведення
#include <conio>              // Для консольного режиму роботи
using namespace std;          // Використання стандартного простору імен

class strClass {              // Оголошення класового типу
        char str[80];
```

```
public:
        void Set(char *s) { strcpy(str, s); }
        void Show()       { cout << "Рядок: " << str << endl; }
};

// Ця функція повертає об'єкт типу strClass.
strClass Init()
{
        strClass obj; char str[80];

        cout << "Введіть рядок: "; cin >> str;
        obj.Set(str);

        return obj;
}

int main()
{
        strClass Obj;      // Створення об'єкта класу

        Obj = Init();      // Присвоюємо об'єкт, повернутий функцією Init(), об'єкту Obj
        Obj.Show();

        getch(); return 0;
}
```

У наведеному прикладі функція Init() створює локальний об'єкт obj класу strClass, а потім зчитує рядок з клавіатури. Цей рядок копіюється в рядок obj.str, після чого об'єкт obj повертається функцією Init() і присвоюється об'єкту Obj у функції **main**().

Потенційна проблема при поверненні об'єктів функціями. Щодо механізму повернення об'єктів функціями, то тут важливо розуміти таке. Якщо функція повертає об'єкт класу, то вона автоматично створює тимчасовий об'єкт, який зберігає повернуте значення. Саме цей тимчасовий об'єкт реально і повертає функція. Після повернення значення об'єкт руйнується. Руйнування тимчасового об'єкта в деяких ситуаціях може викликати непередбачені побічні ефекти. Наприклад, якщо повернутий функцією об'єкт має деструктор, який звільняє динамічно виділену область пам'яті, то ця пам'ять буде звільнена навіть у тому випадку, якщо об'єкт, який отримує повернуте функцією значення, все ще цю пам'ять використовує. Розглянемо таку некоректну версію попереднього коду програми.

Код програми 3.12. Демонстрація механізму появи помилки, яка виникає при поверненні об'єкта функцією

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен

class strClass {             // Оголошення класового типу
```

```
    char *s;
public:
    strClass()           { s = 0; }
    ~strClass()          { if(s) delete[]s; cout << "Звільнення s-пам'яті" << endl; }
    void Set(char *str) { s = new char[strlen(str)+1]; strcpy(s, str); } // Завантаження рядка.
    void Show()          { cout << "s= " << s << endl; }
};

// Ця функція повертає об'єкт типу strClass.
strClass Init()
{
    strClass obj; char str[80];

    cout << "Введіть рядок: "; cin >> str;
    obj.Set(str);

    return obj;
}

int main()
{
    strClass Obj;        // Створення об'єкта класу

    // Присвоюємо об'єкт, повернутий функцією Init(), об'єкту Obj.
    Obj = Init();        // Ця настанова генерує помилку!!!!
    Obj.Show();          // Відображення "сміття".

    getch(); return 0;
}
```

Результати виконання цієї програми мають такий вигляд:

```
Введіть рядок: Привіт
Звільнення s-пам'яті.
Звільнення s-пам'яті.
s= тут появиться сміття
Звільнення s-пам'яті.
```

Зверніть увагу на те, що виклик деструктора ~strClass() відбувається тричі! Вперше він викликається при виході локального об'єкта obj з області видимості у момент його повернення з функції Init(). Другий виклик деструктора ~strClass() відбувається тоді, коли руйнується тимчасовий об'єкт, який повертається функцією Init(). Коли функція повертає об'єкт, то автоматично створюється невидимий (для Вас) тимчасовий об'єкт, який зберігає повернуте значення. У нашому випадку цей об'єкт просто є побітовою копією об'єкта obj. Отже, після повернення з функції використовується деструктор тимчасового об'єкта. Оскільки область пам'яті, що виділяється для зберігання рядка, який вводить користувач, вже була звільнена (причому двічі!), то під час виклику функції Show() на екран буде виведено "сміття"[1].

[1] Ви можете не побачити на екрані виведене "сміття". Це залежить від того, як Ваш компілятор реалізує динамічне виділення області пам'яті. Проте помилка все одно тут наявна.

Нарешті, після завершення роботи коду програми викликається деструктор об'єкта Obj, який належить функції **main**(). Ситуація тут ускладнюється ще й тим, що під час першого виклику деструктора звільняється область пам'яті, виділена для зберігання рядка, отримуваного функцією Init(). Таким чином, у цій ситуації погано не тільки те, що решта два звернення до деструктора ~strClass() спробують звільнити вже звільнену динамічно виділену область пам'яті, але вони можуть зруйнувати систему динамічного розподілу пам'яті.

Тут важливо зрозуміти, що при поверненні об'єкта з функції для тимчасового об'єкта, який зберігає побітову копію об'єкта obj, буде викликано його деструктор. Тому потрібно уникати повернення об'єктів у ситуаціях, коли це може мати згубні наслідки. Для вирішення цього питання замість повернення об'єкта з функції доцільно використати повернення покажчика або посилання на об'єкт. Але здійснити це не завжди вдається. Ще один спосіб вирішення цього питання полягає у використанні конструктора копії, механізм реалізації якого розглянемо у наступному підрозділі.

3.7. Механізми створення та використання конструктора копії

Одним з важливих форматів застосування перевизначеного конструктора є конструктор копії об'єкта. Як було показано в попередніх прикладах, при передачі об'єкта функції або при поверненні його з неї можуть виникати певні проблеми. Один із способів їх уникнення полягає у використанні *конструктора копії,* який є спеціальним типом перевизначеного конструктора.

Основна причина використання конструктора копії полягає у тому, що при передачі об'єкта функції створюється побітова (тобто точна) його копія, яка передається параметру цієї функції. Проте трапляються ситуації, коли така однакова копія об'єкта небажана. Наприклад, якщо оригінальний об'єкт містить покажчик на динамічно виділену область пам'яті, то і покажчик, який належить побітовій копії об'єкта, також посилатиметься на ту ж саму область пам'яті. Отже, якщо у копію об'єкта будуть внесені зміни у вміст цієї області пам'яті, то ці зміни стосуватимуться також оригінального об'єкта! Понад це, внаслідок завершення роботи функції, побітова копія об'єкта буде зруйнована за викликом деструктора, що негативно відобразиться на початковому об'єкті.

Аналогічна ситуація виникає при поверненні об'єкта з функції. Компілятор генерує тимчасовий об'єкт, який зберігає побітову копію об'єкта, що повертається функцією[1]. Цей тимчасовий об'єкт виходить за межі області видимості функції відразу ж, як тільки ініціатору виклику цієї функції буде повернуте "обіцяне" значення, після чого негайно викликається деструктор тимчасового об'єкта. Оскільки цей деструктор руйнує область пам'яті, потрібну для виконання далі коду програми, то наслідки її роботи будуть невтішні.

Основна причина виникнення цієї проблеми полягає у створенні побітової копії об'єкта. Щоб запобігти їй, необхідно точно визначити, що повинно відбува-

[1] Це робиться автоматично і без нашої на те згоди.

тися, коли створюється така копія об'єкта, і, тим самим, уникнути небажаних побічних ефектів. Цього можна домогтися шляхом створення конструктора копії об'єкта.

Найпоширеніший формат конструктора копії об'єкта має такий вигляд:

```
ім'я_класу (const ім'я_класу &obj)
{
    // Тіло конструктора копії
}
```

У цьому записі елемент *&obj* означає посилання на об'єкт, який використовується для ініціалізації іншого об'єкта.

У мові програмування С++ визначено дві ситуації, у яких значення одного об'єкта передається іншому: при присвоєнні та ініціалізації. Ініціалізація об'єкта може виконуватися трьома способами, тобто у випадках, коли:

- один об'єкт безпосередньо ініціалізує інший об'єкт, як, наприклад, в оголошенні;
- копія об'єкта передається як аргумент параметру функції;
- генерується тимчасовий об'єкт (найчастіше як значення, що повертається функцією).

Наприклад, нехай завчасно створено об'єкт ObjY типу myClass. Тоді у процесі виконання таких подальших настанов буде викликано конструктор копії класу myClass:

```
myClass ObjX = ObjY;    // Об'єкт ObjY безпосередньо ініціалізує об'єкт ObjX.
Fun1(ObjY);             // Об'єкт ObjY передається як аргумент
ObjY = Fun2();          // Об'єкт ObjY приймає об'єкт, що повертається функцією.
```

У перших двох випадках конструктору копії буде передано посилання на об'єкт ObjY, а в третьому – посилання на об'єкт, який повертається функцією Fun2().

Вартоа' нати! *Конструктори копії не роблять ніякого впливу на операції присвоєння.*

Щоб глибше зрозуміти призначення конструктора копії, розглянемо ґрунтовніше його значення у кожній з цих трьох ситуацій.

3.7.1. Використання конструктора копії для ініціалізації одного об'єкта іншим

Конструктор копії часто викликається тоді, коли один об'єкт використовується для ініціалізації іншого. Для розуміння сказаного розглянемо таку програму.

Код програми 3.13. Демонстрація механізму використання конструктора копії для ініціалізації одного об'єкта іншим

```
#include <vcl>
#include <iostream>         // Для потокового введення-виведення
#include <conio>            // Для консольного режиму роботи
using namespace std;        // Використання стандартного простору імен

class myClass {             // Оголошення класового типу
    int *p;
public:
    myClass(int c);         // Визначення звичайного конструктора
```

```
            {  p = new int; *p = c;
               cout << "Виділення p-пам'яті звичайним конструктором" << endl; }
        myClass(const myClass &obj);   // Визначення конструктора копії
            { p = new int; *p = *obj.p;
               cout << "Виділення p-пам'яті конструктором копії" << endl; }
        ~myClass();
            { delete p; cout << "Звільнення p-пам'яті" << endl; }
};

int main()
{
        myClass ObjA(10);          // Викликається звичайний конструктор.

        myClass ObjB = ObjA;     // Викликається конструктор копії.

        getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Виділення p-пам'яті звичайним конструктором
Виділення p-пам'яті конструктором копії
Звільнення p-пам'яті
Звільнення p-пам'яті
Звільнення p-пам'яті
```

Результати виконання цієї програми вказують на те, що при створенні об'єкта ObjA викликається звичайний конструктор. Але коли об'єкт ObjA використовують для ініціалізації об'єкта ObjB, то викликається конструктор копії. Його використання гарантує, що об'єкт ObjB виділить для своїх членів-даних власну область динамічної пам'яті. Без конструктора копії об'єкт ObjB просто був би точною копією об'єкта ObjA, а член ObjA.p вказував би на ту ж саму область динамічної пам'яті, що і член ObjB.p.

Потрібно мати на увазі, що конструктор копії викликається тільки при виконанні ініціалізації об'єкта. Наприклад, така послідовність настанов не викличе конструктора копії, визначеного у попередній програмі:

```
myClass ObjA(2), ObjB(3);
//...
ObjB = ObjA;
```

У цьому записі настанова ObjB = ObjA здійснює операцію присвоєння, а не операцію копіювання.

3.7.2. Механізм використання конструктора копії для передачі об'єкта функції

При передачі об'єкта функції як аргумента створюється побітова копія цього об'єкта. Якщо у класі визначено конструктор копії, то саме він і викликається для створення такої копії. Розглянемо код програми, у якій використовується конструктор копії для належного оброблення об'єктів типу myClass під час їх передачі функції як аргументів. Нижче наведемо коректну версію некоректної програми, представленої у розд. 3.5.2.

Код програми 3.14. Демонстрація механізму використання конструктора копії для передачі об'єкта функції

```
#include <vcl>
#include <iostream>            // Для потокового введення-виведення
#include <conio>               // Для консольного режиму роботи
using namespace std;           // Використання стандартного простору імен

class myClass {                // Оголошення класового типу
        int *p;
public:
        myClass(int c);        // Визначення звичайного конструктора
                { p = new int; *p = c;
                cout << "Виділення p-пам'яті звичайним конструктором" << endl; }
        myClass(const myClass &obj);   // Визначення конструктора копії
                { p = new int; *p = *obj.p;
                cout << "Виділення p-пам'яті конструктором копії" << endl; }
        ~myClass();
                { delete p; cout << "Звільнення p-пам'яті" << endl; }
        int Put() { return *p; }
};

// Ця функція приймає один об'єкт-параметр.
void Get(myClass obj)
{
        cout << "*p= " << obj.Put() << endl;
}

int main()
{
        myClass ObjA(10);         // Створення об'єкта класу

        Get(ObjA);

        getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Виділення p-пам'яті звичайним конструктором
Виділення p-пам'яті конструктором копії
*p= 10
Звільнення p-пам'яті
Звільнення p-пам'яті
```

У процесі виконання цієї програми тут відбувається таке: коли у функції **main**() створюється об'єкт ObjA, "стараннями" звичайного конструктора виділяється область пам'яті, адреса якої присвоюється покажчику ObjA.p. Потім об'єкт ObjA передається функції Get(), а саме – її параметру obj. У цьому випадку викликається конструктор копії, який для об'єкта ObjA створює побітову його копію з іменем obj. Конструктор копії виділяє пам'ять для цієї копії, а значення покажчика на виділену область пам'яті присвоює члену p об'єкта-копії. Потім значення, яке адресується покажчиком p початкового об'єкта, записується в область пам'яті, адреса

якої зберігається в покажчику p об'єкта-копії. Таким чином, області пам'яті, що адресуються покажчиками ObjA.p і obj.p, є різними та незалежними одна від одної, але значення (на які вказують ObjA.p і obj.p), що зберігаються в них, однакові. Якби конструктор копії у класі myClass не був завчасно визначений, то внаслідок створення за замовчуванням побітової копії члени ObjA.p і obj.p указували б на одну і ту саму область пам'яті.

Після завершення роботи функції Get() об'єкт obj виходить з області видимості. Цей вихід супроводжується викликом його деструктора, який звільняє область пам'яті, яка адресується покажчиком obj.p. Нарешті, після завершення роботи функція **main**() виходить з області видимості об'єкт ObjA, що також супроводжується викликом його деструктора і відповідним звільненням області пам'яті, яка адресується покажчиком ObjA.p. Як бачимо, використання конструктора копії усуває негативні побічні ефекти, пов'язані з передачею об'єкта функції.

3.7.3. Механізм використання конструктора копії при поверненні функцією об'єкта

Конструктор копії також викликається при створенні тимчасового об'єкта, який є результатом повернення функцією об'єкта. Для розуміння сказаного розглянемо таку навчальну програму.

Код програми 3.15. Демонстрація механізму використання конструктора копії для створення тимчасового об'єкта, що повертається функцією

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен

class myClass {              // Оголошення класового типу
public:
    myClass()                       { cout << "Звичайний конструктор" << endl; }
    myClass(const myClass &obj) { cout << "Конструктор копії" << endl; }
};

myClass Fun()
{
    myClass obj;      // Викликається звичайний конструктор.
    return obj;       // Опосередковано викликається конструктор копії.
}

int main()
{
    myClass ObjA;     // Викликається звичайний конструктор.
    ObjA = Fun();     // Викликається конструктор копії.
    getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Звичайний конструктор
Звичайний конструктор
Конструктор копії
```

Тут звичайний конструктор викликається двічі: перший раз при створенні об'єкта ObjA у функції **main**(), другий – при створенні об'єкта obj у функції Fun(). Конструктор копії викликається в ту мить, коли генерується тимчасовий об'єкт як значення, яке повертається з функції Fun().

Для багатьох програмістів-початківців "приховані" виклики конструкторів копії зазвичай є не зрозумілими. Проте практично кожен клас у професійно написаних програмах містить певний конструктор копії, без якого не можливо уникнути побічних ефектів, які виникають внаслідок створення за замовчуванням побітових копій об'єкта.

3.7.4. Конструктори копії та їх альтернативи

Як неодноразово згадувалося раніше, С++ – потужна мова програмування. Вона має багато засобів, які надають їй надзвичайно широкі можливості, але при цьому її можна назвати складною мовою. Конструктори копії – один з важливих засобів, який багато програмістів-початківців вважають основою складності мови, оскільки механізм їх роботи не сприймається на інтуїтивному рівні. Такі програмісти часто не розуміють, чому конструктори копії мають таке важливе значення для ефективної роботи коду програми. Багато з них не відразу знаходять точну відповідь на запитання: коли потрібен конструктор копії, а коли – ні? Здебільшого у них виникає наступне запитання: чи не існує йому простішої альтернативи? Відповідь також непроста: і так, і ні!

Такі мови програмування, як Java і С#, не мають конструкторів копії, оскільки в жодній з них не створюються побітові копії об'єктів. Йдеться про те, що як Java, так і С# динамічно виділяють пам'ять для всіх об'єктів, а програміст оперує цими об'єктами виключно через посилання. Тому при передачі об'єктів функції як параметрів або при поверненні їх з функцій в копіях об'єктів немає ніякої потреби.

Той факт, що ні мова Java, ні мова С# не потребують конструкторів копії, робить ці мови дещо простішими, але за простоту теж потрібно платити. Робота з об'єктами виключно за допомогою посилань (а не безпосередньо, як у мові програмування С++) накладає обмеження на типи операцій, які може виконувати програміст. Понад це, таке використання об'єктних посилань у мовах Java і С# не дає змоги точно визначити, коли об'єкт буде зруйновано. У мові програмування С++ об'єкт завжди руйнується при виході з області видимості.

Мова С++ надає програмісту повний контроль над ситуаціями, які виникають у процесі роботи коду програми, тому вона є дещо складнішою, ніж мови Java і С#. Це – ціна, яку ми платимо за потужні засоби програмування.

3.8. Поняття про ключове слово this

Під час кожного виклику функції-члена класу їй автоматично передається покажчик на об'єкт, який іменується ключовим словом **this**, для якого викликається

ця функція. Покажчик **this** – це *неявний* параметр, який приймається всіма функціями-членами класу. Отже, в будь-якій функції-члені класу покажчик **this** використовується для посилання на викликуваний об'єкт.

Як уже зазначалося вище, функція-член класу може мати прямий доступ до закритих (**private**) членів-даних свого класу. Наприклад, нехай визначено такий клас:

```
class myClass {          // Оголошення класового типу
    int c;
    void Fun() {...};
    // ...
};
```

У тілі функції Fun() можна використовувати таку настанову для присвоєння члену с значення 10:

```
c = 10;
```

Насправді попередня настанова є скороченою формою такої:

```
this->c = 10;
```

Щоби зрозуміти, як працює покажчик **this**, розглянемо таку навчальну програму.

Код програми 3.16. Демонстрація механізму застосування ключового слова this

```
#include <vcl>
#include <iostream>             // Для потокового введення-виведення
#include <conio>                // Для консольного режиму роботи
using namespace std;            // Використання стандартного простору імен

class myClass { // Оголошення класового типу
    int c;
public:
    void Get(int n)   { this->c = n; }        // те саме, що c = n
    int Put()         { return this->c; }     // те саме, що return c
};

int main()
{
    myClass Obj;      // Створення об'єкта класу
    Obj.Get(100);
    cout << "c= " << Obj.Put() << endl;
    getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані число 100.

Безумовно, цей приклад тривіальний, але у ньому показано, як можна використовувати покажчик **this**. З матеріалу наступних розділів Ви зрозумієте, чому покажчик **this** є таким важливим для написання програм мовою C++.

Вартоа' нати!а*Функції-"друзі" не мають покажчика* ***this****, оскільки вони не є членами класу. Тільки функції-члени класу мають покажчик* ***this****.*

Розділ 4. ОСОБЛИВОСТІ МЕХАНІЗМУ ПЕРЕВИЗНАЧЕННЯ ОПЕРАТОРІВ

Для покращення роботи з визначеними програмістом "класовими" типами у мові програмування C++ оператори можна перевизначати. Отриманий від цього виграш дає змогу органічно інтегрувати нові типи даних користувача у середовище програмування.

Перевизначаючи оператор, можна встановити певну його дію для конкретного класу. Наприклад, клас, який визначає зв'язний список, може використовувати оператор додавання "+" для внесення об'єкта до списку. Клас, який реалізує стек, може використовувати оператор додавання "+" для запису об'єкта в стек. У будь-якому іншому класі аналогічний оператор міг би слугувати для абсолютно іншої мети. При перевизначенні оператора жодне з оригінальних значень його об'єктів не втрачається. Перевизначений оператор (у своїй новій якості) працює як абсолютно новий оператор. Наприклад, при перевизначенні бінарного оператора додавання "+" для оброблення зв'язного списку. Ця функція не призводить до зміни операції додавання стосовно цілочисельних значень.

Механізм перевизначення операторів тісно пов'язаний з механізмом перевизначення функцій. Щоб перевизначити оператор, необхідно визначити дію нової операції для класу, до якого вона застосовуватиметься. Для цього створюється функція **operator** (операторна функція), яка визначає дію цього оператора. Її загальний формат є таким:

```
// Створення операторної функції
тип ім'я_класу::operator#(перелік_аргументів)
{
    операція_над_класом
}
```

У цьому записі перевизначена операція позначається символом "**#**", а елемент *тип* вказує на тип значення, що повертається внаслідок виконання даної операції. І хоча він у принципі може бути будь-яким, тип значення часто збігається з іменем класу, для якого перевизначається функція **operator**. Такий зв'язок полегшує використання перевизначеного оператора у складних арифметичних і логічних виразах. Як буде показано далі, конкретне значення елемента *перелік_аргументів* визначається декількома чинниками.

4.1. Механізми перевизначення операторів з використанням функцій-членів класу

Операторна функція може бути членом класу або не бути ним. Операторні функції, які не є членами класу, визначаються як його "друзі". Операторні функції-члени і не члени класу відрізняються між собою механізмом перевизначення.

Кожний з механізмів перевизначення операторів спробуємо розглянути більш детально на конкретних прикладах.

4.1.2. Перевизначення бінарних операторів додавання "+" і присвоєння "="

Почнемо з простого прикладу. У наведеному нижче коді програми створюється клас kooClass, який підтримує дії з координатами об'єкта в тривимірному просторі. Для класу kooClass перевизначаються оператори додавання "+" і присвоєння "=". Отже, розглянемо уважно код цієї програми.

Код програми 4.1. Демонстрація механізму перевизначення бінарних операторів додавання "+" та присвоєння "=" за допомогою функцій-членів класу

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен

class kooClass {             // Оголошення класового типу
	int x, y, z;             // Тривимірні координати
public:
	kooClass() {x = y = z = 0; }
	kooClass(int c, int d, int f) {x = c; y = d; z = f; }
	kooClass operator+(kooClass obj);    // Операнд obj передається неявно.
	kooClass operator=(kooClass obj);    // Операнд obj передається неявно.
	void Show(char *s);
};

// Перевизначення бінарного оператора додавання "+".
kooClass kooClass::operator+(kooClass obj)
{
	kooClass tmp;            // Створення тимчасового об'єкта
	tmp.x = x + obj.x;       // Операції додавання цілочисельних значень
	tmp.y = y + obj.y;       // зберігають початковий вміст операндів
	tmp.z = z + obj.z;

	return tmp;              // Повертає модифікований тимчасовий об'єкт
}

// Перевизначення оператора присвоєння "=".
kooClass kooClass::operator=(kooClass obj)
{
	x = obj.x;  // Операції присвоєння цілочисельних значень
	y = obj.y;  // зберігають початковий вміст операндів
	z = obj.z;

	// Повернення модифікованого об'єкта операнда, адресованого покажчиком
	return *this;
}
```

```
// Відображення тривимірних координат x, y, z.
void kooClass::Show(char *s)
{
    cout << "Координати об'єкта <" << s << ">: ";
    cout << "\t\tx= " << x << ", y= " << y << ", z= " << z << endl;
}

int main()
{
    kooClass ObjA(1, 2, 3), ObjB(10, 10, 10), ObjC;
    ObjA.Show("A");
    ObjB.Show("B");

    ObjC = ObjA + ObjB;          // Додавання об'єктів ObjA і ObjB
    ObjC.Show("C=A+B");
    ObjC = ObjA + ObjB + ObjC;   // Множинне додавання об'єктів
    ObjC.Show("C=A+B+C");
    ObjC = ObjB = ObjA;          // Множинне присвоєння об'єктів
    ObjB.Show("B=A");
    ObjC.Show("C=B");
    getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Координати об'єкта <A>:          x= 1, y= 2, z= 3
Координати об'єкта <B>:          x= 10, y= 10, z= 10
Координати об'єкта <C=A+B>:      x= 11, y= 12, z= 13
Координати об'єкта <C=A+B+C>:    x= 22, y= 24, z= 26
Координати об'єкта <B=A>:        x= 1, y= 2, z= 3
Координати об'єкта <C=B>:        x= 1, y= 2, z= 3
```

Аналізуючи код цієї програми, можна побачити, що обидві операторні функції мають тільки по одному параметру, хоча вони перевизначають бінарні операції. Цю, на перший погляд, "кричущу" суперечність можна легко пояснити. Йдеться про те, що при перевизначенні бінарного оператора з використанням функції-члена класу їй передається безпосередньо тільки один аргумент. Другий же опосередковано передається через покажчик **this**. Таким чином, у рядку

```
tmp.x = x + obj.x;
```

під членом-даних x маємо на увазі член **this**->x, тобто член x зв'язується з об'єктом, який викликає дану операторну функцію. В усіх випадках опосередковано передається об'єкт, який вказується зліва від символу операції, тобто той, який став причиною виклику операторної функції. Об'єкт, який розташовується з правого боку від символу операції, передається цій функції як аргумент. У загальному випадку під час застосування функції-члена класу для перевизначення унарного оператора параметри не використовуються взагалі, а для перевизначення бінарного оператора береться до уваги тільки один параметр[1]. У будь-якому випадку об'єкт, який викликає операторну функцію, опосередковано передається через покажчик **this**.

[1] Тернарний оператор "?" перезавантажувати не можна.

*Нео! хідноапам'ятати! Якщо для перевизначення бінарного оператора використовується функція-член класу, то об'єкт, який знаходиться зліва від операції, викликає операторну функцію і передається їй опосередковано через покажчик **this**. Об'єкт, який розташовується праворуч від операції, передається операторній функції як параметр.*

Щоб зрозуміти механізм перевизначення операторів, розглянемо уважно наведену вище програму, починаючи з перевизначеного оператора додавання "+". Під час оброблення двох об'єктів типу kooClass оператором додавання "+" виконуються операції додавання значень відповідних координат так, як це показано у функції **operator**+(). Але зауважте, що ця операторна функція не модифікує значень жодного операнда. Як результат виконання операції ця функція повертає об'єкт типу kooClass, який містить результати попарного додавання координат двох об'єктів. Щоб зрозуміти, чому операція "+" не змінює вміст жодного з об'єктів-учасників, розглянемо стандартну арифметичну операцію додавання, що застосовується, наприклад, до чисел 10 і 12. Отож, результат виконання операції 10+12 дорівнює 22, але при його отриманні ні число 10, ні 12 не були змінені. Хоча не існує правила, яке б не давало змоги перевизначеному оператору змінювати значення одного з його операндів, все ж таки краще, щоб він не суперечив загальноприйнятим нормам і залишався у згоді зі своїм оригінальним призначенням.

Зверніть увагу на те, що операторна функція **operator**+() повертає об'єкт типу kooClass, хоча вона могла б повертати значення будь-якого іншого допустимого типу, що визначається мовою програмування C++. Однак, той факт, що вона повертає об'єкт типу kooClass, дає змогу використовувати оператор додавання "+" у таких складних виразах, як ObjA + ObjB + ObjC – множинне додавання. Частина цього виразу (ObjA + ObjB) отримує результат типу kooClass, який потім додається до об'єкта ObjC. І якби ця частина виразу генерувала значення іншого типу (а не типу kooClass), то таке множинне додавання просто не відбулося б.

На відміну від оператора додавання "+", оператор присвоєння "=" модифікує один зі своїх аргументів[1]. Оскільки операторна функція **operator**=() викликається об'єктом, який розташований зліва від символу присвоєння "=", то саме цей об'єкт і модифікується внаслідок виконання операції присвоєння. Після виконання цієї операції значення, яке повертається перевизначеним оператором, містить об'єкт, який було вказано зліва від символу присвоєння[2]. Наприклад, щоб можна виконувати настанови, подібні до такої (множинне присвоєння)

ObjA = ObjB = ObjC = ObjD;,

необхідно, щоб операторна функція **operator**=() повертала об'єкт, який адресується покажчиком **this**, і щоб цей об'єкт розташовувався зліва від оператора присвоєння "=". Це дасть змогу виконати будь-який ланцюжок присвоєнь.

Операція присвоєння – це одне з найважливіших застосувань покажчика **this**.

[1] Передусім, це становить саму суть присвоєння.
[2] Такий стан речей цілком узгоджується з традиційною дією оператора "=".

4.1.2. Перевизначення унарних операторів інкремента "++" та декремента "--"

Можна перевизначати унарні оператори інкремента "++" та декремента "--", або унарні "-" і "+". Як уже зазначалося вище, при перевизначенні унарного оператора за допомогою функції-члена класу операторній функції жоден об'єкт не передається безпосередньо. Операція ж здійснюється над об'єктом, який викликає цю функцію через опосередковано переданий покажчик **this**. Наприклад, розглянемо дещо змінену версію попереднього коду програми. У наведеному нижче його варіанті для об'єктів типу kooClass визначається бінарна операція віднімання та унарна операція інкремента .

Код програми 4.2. Демонстрація механізму перевизначення префіксної форми унарного оператора інкремента "++"

```
#include <vcl>
#include <iostream>                    // Для потокового введення-виведення
#include <conio>                       // Для консольного режиму роботи
using namespace std;                   // Використання стандартного простору імен

class kooClass {                       // Оголошення класового типу
    int x, y, z;                       // Тривимірні координати
public:
    kooClass() {x = y = z = 0; }
    kooClass(int c, int d, int f) {x = c; y = d; z = f; }
    kooClass operator-(kooClass obj);      // Операнд obj передається неявно.
    kooClass operator=(kooClass obj);      // Операнд obj передається неявно.
    kooClass operator++();                 // Префіксна форма оператора інкремента "++"
    void Show(char *s);
};

// Перевизначення бінарного оператора віднімання "-".
kooClass kooClass::operator-(kooClass obj)
{
    kooClass tmp;                      // Створення тимчасового об'єкта
    tmp.x = x - obj.x;                 // Операції віднімання цілочисельних значень
    tmp.y = y - obj.y;                 // зберігають початковий вміст операндів.
    tmp.z = z - obj.z;

    return tmp;                        // Повертає модифікований тимчасовий об'єкт
}

// Перевизначення оператора присвоєння "=".
kooClass kooClass::operator=(kooClass obj)
{
    x = obj.x; // Операції присвоєння цілочисельних значень
    y = obj.y; // зберігають початковий вміст операндів.
    z = obj.z;
    // Повернення модифікованого об'єкта операнда, адресованого покажчиком
    return *this;
}
```

```
// Перевизначення префіксної форми унарного оператора інкремента "++".
kooClass kooClass::operator++()
{
	x++; // Інкремент координат x, y i z
	y++;
	z++;

	// Повернення модифікованого об'єкта операнда, адресованого покажчиком
	return *this;
}

// Відображення тривимірних координат x, y, z.
void kooClass::Show(char *s)
{
	cout << "Координати об'єкта <" << s << ">: ";
	cout << "\t\tx= " << x << ", y= " << y << ", z= " << z << endl;
}

int main()
{
	kooClass ObjA(1, 2, 3), ObjB(10, 10, 10), ObjC;

	ObjA.Show("A");
	ObjB.Show("B");

	ObjC = ObjA - ObjB;			// Віднімання об'єктів ObjA і ObjB
	ObjC.Show("C = A-B");

	ObjC = ObjA - ObjB - ObjC;		// Множинне віднімання об'єктів
	ObjC.Show("C = A-B-C");

	ObjC = ObjB = ObjA;			// Множинне присвоєння об'єктів
	ObjB.Show("B=A");
	ObjC.Show("C=B");

	++ObjC;				// Префіксний інкремент об'єкта ObjC
	ObjC.Show("++C");

	ObjA = ++ObjC;			// Префіксний інкремент об'єкта ObjC
	ObjC.Show("C");
	ObjA.Show("A = ++C");

	getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Координати об'єкта <A>:		x= 1, y= 2, z= 3
Координати об'єкта <B>:		x= 10, y= 10, z= 10
Координати об'єкта <C=A-B>:		x= -9, y= -8, z= -7
Координати об'єкта <C=A-B-C>:	x= 0, y= 0, z= 0
Координати об'єкта <B=A>:		x= 1, y= 2, z= 3
```

```
Координати об'єкта <C=B>:          x= 1, y= 2, z= 3
Координати об'єкта <++C>:          x= 2, y= 3, z= 4
Координати об'єкта <C>:            x= 3, y= 4, z= 5
Координати об'єкта <A=++C>         x= 3, y= 4, z= 5
```

Як видно з останнього рядка результату виконання програми, операторна функція **operator**++() інкрементує кожну координату об'єкта і повертає модифіковане його значення, яке повністю узгоджується з традиційною дією оператора інкремента "++".

Як уже зазначалося вище, оператори інкремента "++" та декремента "--" мають дві форми – префіксну і постфіксну. Наприклад, оператор інкремента можна використовувати у префіксній формі

```
++ObjC;
```

і у постфіксній формі

```
ObjC++;.
```

Як зазначено в коментарях до попереднього коду програми, операторна функція **operator**++() визначає префіксну форму операції інкремента "++" для класу kooClass. Але це не заважає перевизначати і його постфіксну форму. Оголошення прототипу постфіксної форми унарного оператора інкремента "++" для класу kooClass має такий вигляд:

```
kooClass kooClass::operator++(int notused);
```

Параметр notused не використовується самою функцією. Він слугує індикатором для компілятора, який дає змогу відрізнити префіксну форму оператора інкремента від постфіксної[1]. Нижче наведено один з можливих способів реалізації постфіксної форми унарного оператора інкремента "++" для класу kooClass:

```
// Перевизначення постфіксної форми унарного оператора інкремента "++".
kooClass kooClass::operator++(int notused)
{
    kooClass tmp = *this;        // Збереження початкового значення об'єкта
    x++;                         // Інкремент координат x, y і z
    y++;
    z++;
    return tmp;                  // Повернення початкового значення об'єкта
}
```

Зверніть увагу на те, що ця операторна функція зберігає початкове значення операнда шляхом виконання такої настанови:

```
kooClass tmp = *this;
```

Збережене значення операнда (у об'єкті tmp) повертається за допомогою настанови **return**. Потрібно мати на увазі, що традиційний постфіксний оператор інкремента спочатку набуває значення операнда, а потім його інкрементує. Отже, перш ніж інкрементувати поточне значення операнда, його потрібно зберегти, а потім повернути (не забувайте, що постфіксний оператор інкремента не повинен повертати модифіковане значення свого операнда). У наведеному нижче коді програмі реалізовано обидві форми унарного оператора інкремента "++".

---

[1] Цей параметр також використовується як ознака постфіксної форми і для оператора декремента.

Код програми 4.3. Демонстрація механізму перевизначення унарного оператора інкремента "++" з використанням його префіксної та постфіксної форм

```
#include <vcl>
#include <iostream>                    // Для потокового введення-виведення
#include <conio>                       // Для консольного режиму роботи
using namespace std;                   // Використання стандартного простору імен

class kooClass {                       // Оголошення класового типу
	int x, y, z;                       // Тривимірні координати
public:
	kooClass() { x = y = z = 0; }
	kooClass(int c, int d, int f) {x = c; y = d; z = f; }
	kooClass operator*(kooClass obj);      // Операнд obj передається неявно.
	kooClass operator=(kooClass obj);      // Операнд obj передається неявно.
	kooClass operator++();                 // Префіксна форма оператора інкремента "++"

	// Постфіксна форма оператора інкремента "++"
	kooClass operator++(int notused);

	// Префіксна форма унарного оператора зміни знаку "-"
	kooClass operator-();
	void Show(char *s);
};

// Перевизначення бінарного оператора множення "*".
kooClass kooClass::operator*(kooClass obj)
{
	kooClass tmp;                      // Створення тимчасового об'єкта

	tmp.x = x * obj.x;                 // Операції множення цілочисельних значень
	tmp.y = y * obj.y;                 // зберігають початковий вміст операндів
	tmp.z = z * obj.z;

	return tmp;                        // Повертає модифікований тимчасовий об'єкт
}

// Перевизначення оператора присвоєння "=".
kooClass kooClass::operator=(kooClass obj)
{
	x = obj.x; // Операції присвоєння цілочисельних значень
	y = obj.y; // зберігають початковий вміст операндів
	z = obj.z;

	// Повернення модифікованого об'єкта операнда, адресованого покажчиком
	return *this;
}

// Перевизначення префіксної форми унарного оператора інкремента "++".
kooClass kooClass::operator++()
```

```
{
        x++; // Інкремент координат x, y і z
        y++;
        z++;

        // Повернення модифікованого об'єкта операнда, адресованого покажчиком
        return *this;
}

// Перевизначення постфіксної форми унарного оператора інкремента "++".
kooClass kooClass::operator++(int notused)
{
        kooClass tmp = *this; // Збереження початкового значення об'єкта

        x++;                                    // Інкремент координат x, y і z
        y++;
        z++;
        return tmp;                             // Повернення початкового значення об'єкта
}

// Перевизначення префіксної форми унарного оператора зміни знаку "-".
kooClass kooClass::operator-()
{
        x=-x; // Зміна знаку координат x, y і z
        y=-y;
        z=-z;
        // Повернення модифікованого об'єкта операнда, адресованого покажчиком
        return *this;
}


// Відображення тривимірних координат x, y, z.
void kooClass::Show(char *s)
{
        cout << "Координати об'єкта <" << s << ">: ";
        cout << "\t\tx= " << x << ", y= " << y << ", z= " << z << endl;
}

int main()
{
        kooClass ObjA(1, 2, 3), ObjB(10, 10, 10), ObjC;

        ObjA.Show("A");
        ObjB.Show("B");

        ObjC = ObjA * ObjB;     // Множення об'єктів ObjA і ObjB
        ObjC.Show("C=A*B");

        ObjC = ObjA * ObjB * ObjC; // Множинне множення об'єктів
        ObjC.Show("C=A*B*C");
```

```
        ObjC = ObjB = ObjA;     // Множинне присвоєння об'єктів
        ObjC.Show("C=B");
        ObjB.Show("B=A");

        ++ObjC;                 // Префіксна форма операції інкремента
        ObjC.Show("++C");

        ObjC++;                 // Постфіксна форма операції інкремента
        ObjC.Show("C++");

        ObjA = ++ObjC; // Об'єкт ObjA набуває значення об'єкта ObjC після його інкрементування.
        ObjA.Show("A = ++C"); // Тепер об'єкти ObjA і ObjC мають однакові значення.
        ObjC.Show("C");

        ObjA = ObjC++; // Об'єкт ObjA набуває значення об'єкта ObjC до його інкрементування.
        ObjA.Show("A=C++");   // Тепер об'єкти ObjA і ObjC мають різні значення.
        ObjC.Show("C");

        -ObjC;                  // Префіксна форма операції зміни знаку
        ObjC.Show("-C");

        getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Координати об'єкта <A>:          x= 1, y= 2, z= 3
Координати об'єкта <B>:          x= 10, y= 10, z= 10
Координати об'єкта <C=A*B>:      x= 10, y= 20, z= 30
Координати об'єкта <C=A*B*C>:    x= 100, y= 400, z= 900
Координати об'єкта <C=B>:        x= 1, y= 2, z= 3
Координати об'єкта <B=A>:        x= 1, y= 2, z= 3
Координати об'єкта <++C>:        x= 2, y= 3, z= 4
Координати об'єкта <C++>:        x= 3, y= 4, z= 5
Координати об'єкта <A=++C>:      x= 4, y= 5, z= 6
Координати об'єкта <C>:          x= 4, y= 5, z= 6
Координати об'єкта <A=C++>:      x= 4, y= 5, z= 6
Координати об'єкта <C>:          x= 5, y= 6, z= 7
Координати об'єкта <-C>:         x= -5, y= -6, z= -7
```

Як підтверджують останні рядки результату виконання програми, при префіксному інкрементуванні об'єкта ObjC його значення збільшується до виконання операції присвоєння об'єкту ObjA, при постфіксному інкрементуванні – після виконання операції присвоєння.

Нео! хідноапам'ятати! *Якщо символ "++" знаходиться перед операндом, то викликається операторна функція* ***operator***++(), *а якщо після операнда – то операторна функція* ***operator***++(***int*** notused). *Аналогічний підхід використовується і для перевизначення префіксної та постфіксної форм оператора декремента для будь-якого класу*[1].

[1] Для домашньої вправи спробуйте визначити операторну функцію декремента для класу `kooClass`

***Вартоо' нати!а**Ранні версії мови С++ не містили відмінностей між префіксною і постфіксною формами операторів інкремента і декремента. Раніше в обох випадках викликалася префіксна форма операторної функції. Це потрібно мати на увазі тоді, коли доведеться працювати із старими С++-програмами.*

4.1.3. Особливості реалізації механізму перевизначення операторів

Дія перевизначеного оператора стосовно класу, для якого вона визначається, не обов'язково повинна співпадати з стандартними діями цього оператора стосовно вбудованих С++-типів. Наприклад, оператори "<<" і ">>", які вживаються до об'єктів **cout** і **cin**, мають мало спільного з аналогічними операторами, що застосовуються у логічних операторах для порівняння значень цілочисельного типу. Але для поліпшення структурованості та читабельності коду програми створюваний програмістом перевизначений оператор повинен за змогою відображати традиційне призначення тої або іншої операції. Наприклад, оператор додавання "+", перевизначений для класу kooClass, концептуально подібний до операції "+", визначеної для цілочисельних типів. Адже безглуздо у визначенні, наприклад, операції множення "*", яка за своєю дією більше нагадуватиме операцію ділення "/". Таким чином, основна ідея створення програмістом перевизначених операторів – наділити їх новими (потрібними для нього) можливостями, які, зазвичай, пов'язані з їх первинним призначенням.

На перевизначення операторів накладається ряд обмежень. По-перше, не можна змінювати пріоритет операцій. По-друге, не можна змінювати кількість операндів, які приймаються оператором, хоча операторна функція могла б ігнорувати будь-який операнд. Окрім цього, за винятком оператора виклику функції (про нього піде мова попереду), операторні функції не можуть мати аргументів за замовчуванням. Нарешті, деякі оператори взагалі не можна перевизначати:

```
::     .*     ?
```

Оператор ".*" – це оператор спеціального призначення, який буде розглядатися нижче у цьому навчальному посібнику.

Значення порядку слідування операндів. Перевизначаючи бінарні оператори, потрібно пам'ятати, що у багатьох випадках порядок слідування операндів має значення. Наприклад, вираз А + В комутативний, а вираз А – В – ні[1]. Отже, реалізуючи перевизначені версії не комутативних операторів, потрібно пам'ятати, який операнд знаходиться зліва від символу операції, а який – праворуч від нього. Наприклад, у наведеному нижче коді програми продемонстровано механізм перевизначення оператора ділення для класу kooClass:

```
// Перевизначення бінарного оператора ділення "/".
kooClass kooClass::operator/(kooClass obj)
{
	kooClass tmp;	// Створення тимчасового об'єкта

	tmp.x = x / obj.x;
```

[1] Іншими словами, А – В не те ж саме, що В – А!

```
        tmp.y = y / obj.y;
        tmp.z = z / obj.z;

        return tmp;        // Повертає модифікований тимчасовий об'єкт
    }
```

Нео! хідноапам'ятати! Саме лівий операнд викликає операторну функцію. Правий операнд передається безпосередньо. Ось чому для коректного виконання операції ділення "/" використовується саме такий порядок слідування операндів: x / obj.x.

4.2. Механізми перевизначення операторів з використанням функцій-не членів класу

Перевизначення бінарних і унарних операторів для класу можна реалізувати і з використанням функцій, які не є членами класу. Однак такі функції необхідно оголосити "друзями" класу. Як уже зазначалося вище, функції-не члени класу (у тому числі і функції-"друзі") не мають покажчика **this**. Отже, якщо для перевизначення бінарного оператора використовується "дружня" функція класу, то для виконання певної операції операторній функції потрібно безпосередньо передати обидва операнди. Якщо ж за допомогою "дружньої" функції класу перевизначається унарний оператор, то операторній функції передається один операнд. З використанням функцій-не членів класу не можна перевизначати такі оператори:
=, (), [], –>.

Бінарні операторні функції, які не є членами класу, мають два параметри, а унарні (теж не члени класу) − один.

4.2.1. Використання функцій-"друзів" класу для перевизначення бінарних операторів

У наведеному нижче коді програми для перевизначення бінарного оператора додавання "+" використовується "дружня" функція класу.

Код програми 4.4. Демонстрація механізму перевизначення бінарного оператора додавання "+" за допомогою "дружньої" функції класу

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен

class kooClass {             // Оголошення класового типу
        int x, y, z;         // Тривимірні координати
public:
        kooClass() {x = y = z = 0; }
        kooClass(int c, int d, int f) { x = c; y = d; z = f; }
        friend kooClass operator+(kooClass obi, kooClass obj);
        kooClass operator=(kooClass obj); // Операнд obj передається неявно.
```

```
    void Show(char *s);
};

// Операторна "дружня" функція класу.
// Перевизначення бінарного оператора додавання "+".
kooClass operator+(kooClass obi, kooClass obj)
{
    kooClass tmp;     // Створення тимчасового об'єкта
    tmp.x = obi.x + obj.x;
    tmp.y = obi.y + obj.y;
    tmp.z = obi.z + obj.z;
    return tmp; // Повертає модифікований тимчасовий об'єкт
}

// Перевизначення оператора присвоєння "=".
kooClass kooClass::operator=(kooClass obj)
{
    x = obj.x;
    y = obj.y;
    z = obj.z;
    // Повернення модифікованого об'єкта операнда, адресованого покажчиком
    return *this;
}

// Відображення тривимірних координат x, y, z.
void kooClass::Show(char *s)
{
    cout << "Координати об'єкта <" << s << ">: ";
    cout << "\t\tx= " << x << ", y= " << y << ", z= " << z << endl;
}

int main()
{
    kooClass ObjA(1, 2, 3), ObjB(10, 10, 10), ObjC;

    ObjA.Show("A");
    ObjB.Show("B");

    ObjC = ObjA + ObjB;      // Додавання об'єктів ObjA і ObjB
    ObjC.Show("C=A+B");

    ObjC = ObjA + ObjB + ObjC; // Множинне додавання об'єктів
    ObjC.Show("C=A+B+C");

    ObjC = ObjB = ObjA ;     // Множинне присвоєння об'єктів
    ObjC.Show("C=B");
    ObjB.Show("B=A");

    getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Координати об'єкта <A>:          x= 1, y= 2, z= 3
Координати об'єкта <B>:          x= 10, y= 10, z= 10
Координати об'єкта <C=A+B>:      x= 11, y= 12, z= 13
Координати об'єкта <C=A+B+C>:    x= 22, y= 24, z= 26
Координати об'єкта <C=B>:        x= 1, y= 2, z= 3
Координати об'єкта <B=A>:        x= 1, y= 2, z= 3
```

Як бачимо, операторній функції **operator**+() тепер передаються два операнди. Лівий операнд передається параметру obi, а правий – параметру obj.

У багатьох випадках при перевизначенні операторів за допомогою функцій-"друзів" класу немає ніякої переваги порівняно з використанням функцій-членів класу. Проте часто трапляються ситуації (коли потрібно, щоб зліва від бінарного оператора знаходився об'єкт вбудованого типу), у яких "дружня" функція класу виявляється надзвичайно корисною. Щоб зрозуміти це твердження, розглянемо такий випадок. Як уже зазначалося вище, покажчик на об'єкт, який викликає операторну функцію-члена класу, передається за допомогою ключового слова **this**. Під час використання бінарного оператора функцію викликає об'єкт, який розташований зліва від нього. І це чудово за умови, що лівий об'єкт визначає задану операцію. Наприклад, якщо у нас є певний об'єкт tmp.obj, для якого визначено операцію додавання з цілим числом, тоді такий запис є цілком допустимим виразом:

```
tmp.obj + 10; // працюватиме
```

Оскільки об'єкт tmp.obj знаходиться зліва від операції додавання "+", то він викликає операторну функцію, яка (імовірно) здатна виконати операцію додавання цілочисельного значення з деяким елементом об'єкта tmp.obj. Але наведений нижче вираз працювати не буде:

```
10 + tmp.obj; // не працюватиме
```

Йдеться про те, що у цьому записі константа, яка розташована зліва від оператора додавання "+", є цілим числом, тобто є значенням вбудованого типу, для якого не визначено жодної операції, операндами якої є ціле число і об'єкт класового типу.

Вирішення такого питання базується на перевизначенні оператора додавання "+" з використанням двох функцій-"друзів" класу. У цьому випадку операторній функції безпосередньо передаються обидва операнди, після чого вона виконується подібно до будь-якої іншої перевизначеної функції, тобто на основі типів її аргументів. Одна версія операторної функції **operator**+() оброблятиме аргументи *об'єкт* + ***int**-значення,* а інша – аргументи ***int**-значення* + *об'єкт.* Перевизначення бінарного оператора додавання "+" (або будь-якого іншого бінарного оператора: "–", "*", "/") з використанням функцій-"друзів" класу дає змогу розташовувати значення вбудованого типу як справа, так і зліва від операції. Механізм перевизначення такої операторної функції показано у наведеному нижче коді програми.

Код програми 4.5. Демонстрація механізму перевизначення бінарних операторів множення "*" і ділення "/" з використанням функцій-"друзів" класу

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
```

```
#include <conio>                // Для консольного режиму роботи
using namespace std;            // Використання стандартного простору імен

class kooClass {        // Оголошення класового типу
        int x, y, z;            // Тривимірні координати
public:
        kooClass() {x = y = z = 0; }
        kooClass(int c, int d, int f) { x = c; y = d; z = f; }
        friend kooClass operator*(kooClass obi, int c);
        friend kooClass operator*(int c, kooClass obi);
        friend kooClass operator/(kooClass obi, int c);
        friend kooClass operator/(int c, kooClass obi);
        void Show(char *s);
};

// Операторна "дружня" функція класу.
// Перевизначення бінарного оператора множення "*".
kooClass operator*(kooClass obi, int c)
{
        kooClass tmp;   // Створення тимчасового об'єкта

        tmp.x = obi.x * c;
        tmp.y = obi.y * c;
        tmp.z = obi.z * c;
        return tmp; // Повертає модифікований тимчасовий об'єкт
}

// Операторна "дружня" функція класу.
// Перевизначення бінарного оператора множення "*".
kooClass operator*(int c, kooClass obi)
{
        kooClass tmp;   // Створення тимчасового об'єкта

        tmp.x = c * obi.x;
        tmp.y = c * obi.y;
        tmp.z = c * obi.z;
        return tmp; // Повертає модифікований тимчасовий об'єкт
}

// Перевизначення бінарного оператора ділення "/".
kooClass operator/(kooClass obi, int c)
{
        kooClass tmp;   // Створення тимчасового об'єкта

        tmp.x = obi.x / c;
        tmp.y = obi.y / c;
        tmp.z = obi.z / c;

        return tmp; // Повертає модифікований тимчасовий об'єкт
}
```

```
// Перевизначення бінарного оператора ділення "/".
kooClass operator/(int c, kooClass obi)
{
	kooClass tmp;	// Створення тимчасового об'єкта

	tmp.x = c / obi.x;
	tmp.y = c / obi.y;
	tmp.z = c / obi.z;

	return tmp; // Повертає модифікований тимчасовий об'єкт
}


// Відображення тривимірних координат x, y, z.
void kooClass::Show(char *s)
{
	cout << "Координати об'єкта <" << s << ">: ";
	cout << "\t\tx= " << x << ", y= " << y << ", z= " << z << endl;
}

int main()
{
	kooClass ObjA(1, 2, 3), ObjB(10, 10, 10), ObjC;
	int a = 10, b = 5;

	ObjA.Show("A");
	ObjB.Show("B");

	ObjC = ObjA * a; // Множення об'єкта ObjA на число a
	ObjC.Show("C=A*a");

	ObjC = a * ObjA; // Множення числа a на об'єкт ObjA
	ObjC.Show("C=a*A");

	ObjC = ObjB / b; // Ділення об'єкта ObjB на число b
	ObjC.Show("C=B*b");

	ObjC = a / ObjB; // Ділення числа a на об'єкт ObjB
	ObjC.Show("C=a/B");

	getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Координати об'єкта <A>:		x= 1, y= 2, z= 3
Координати об'єкта <B>:		x= 10, y= 10, z= 10
Координати об'єкта <C=A*a>:	x= 10, y= 20, z= 30
Координати об'єкта <C=a*A>:	x= 10, y= 20, z= 30
Координати об'єкта <C=B/b>:	x= 2, y= 2, z= 2
Координати об'єкта <C=a/B>:	x= 1, y= 1, z= 1
```

З наведеного вище бачимо, що операторна функція **operator***() перевизначається двічі, забезпечуючи при цьому два можливі випадки участі цілого числа і об'єкта типу kooClass в операції додавання. Аналогічно перевизначається двічі операторна функція **operator**/().

4.2.2. Використання функцій-"друзів" класу для перевизначення унарних операторів

За допомогою функцій-"друзів" класу можна перевизначати й унарні оператори. Але усвідомлення механізму реалізації такого перевизначення вимагатиме від програміста деяких додаткових зусиль. Спершу подумки повернемося до початкової форми перевизначення унарного оператора інкремента "++", визначеного для класу kooClass і реалізованого у вигляді функції-члена класу. Для зручності проведення аналізу наведемо код цієї операторної функції:

```
// Перевизначення префіксної форми унарного оператора інкремента "++"
kooClass kooClass::operator++()
{
    x++;
    y++;
    z++;
    // Повернення модифікованого об'єкта операнда, адресованого покажчиком
    return *this;
}
```

Як уже зазначалося вище, кожна функція-член класу отримує (як опосередковано переданий) аргумент **this**, який є покажчиком на об'єкт, який викликав цю функцію. При перевизначенні унарного оператора за допомогою функції-члена класу аргумент безпосередньо не передаються. Єдиним аргументом, необхідним у цій ситуації, є неявний покажчик на викликуваний об'єкт. Будь-які зміни, що вносяться в члени-даних об'єкта, вплинуть на об'єкт, для якого було викликано цю операторну функцію. Отже, у процесі виконання настанови x++ (у попередній функції) буде інкрементовано член-даних x викликуваного об'єкта.

На відміну від функцій-членів класу, функції-не члени (у тому числі і "друзі") класу не отримують покажчика **this** і, як наслідок, не мають доступу до об'єкта, для якого вони були викликані. Але, як уже зазначалося вище, операторній "дружній" функції операнд передається безпосередньо. Тому спроба створити операторну "дружню" функцію **operator**++() у такому вигляді успіху не матиме:

```
// Цей варіант перевизначення операторної функції працювати не буде
kooClass operator++(kooClass obi)
{
    obi.x++;
    obi.y++;
    obi.z++;
    return obi;
}
```

Ця операторна функція не працездатна, оскільки тільки копія об'єкта, яка активізує виклик функції **operator**++(), передається функції через параметр obi. Таким

чином, зміни в тілі функції **operator++**() не вплинуть на викликуваний об'єкт, позаяк вони змінюють тільки локальний параметр.

Тим не менше, якщо все ж таки виникає бажання використовувати "дружню" функцію класу для перевизначення операторів інкремента або декремента, то необхідно передати їй об'єкт за посиланням. Оскільки посилальний параметр є неявним покажчиком на аргумент, то зміни, внесені в параметр, вплинуть і на аргумент. Застосування посилального параметра дає змогу функції успішно інкрементувати або декрементувати об'єкт, який використовується як операнд.

Таким чином, якщо для перевизначення операторів інкремента або декремента використовується "дружня" функція класу, то її префіксна форма приймає один параметр (який і є операндом), а постфіксна форма – два параметри (другим є цілочисельне значення, яке не використовується).

Нижче наведено повний код програми оброблення тривимірних координат, у якій використовується операторна "дружня" функція класу **operator++**(). Звернемо тільки увагу на те, що перевизначеними є як префіксна, так і постфіксна форми операторів інкремента.

Код програми 4.6. Демонстрація механізму використання "дружньої" функції класу для перевизначення префіксної та постфіксної форми операторів інкремента

```
#include <vcl>
#include <iostream>              // Для потокового введення-виведення
#include <conio>                 // Для консольного режиму роботи
using namespace std;             // Використання стандартного простору імен

class kooClass {         // Оголошення класового типу
        int x, y, z; // Тривимірні координати
public:
        kooClass() {x = y = z = 0; }
        kooClass(int c, int d, int f) {x = c; y = d; z = f; }
        friend kooClass operator*(kooClass obi, kooClass obj);
        kooClass operator=(kooClass obj);
        // Ці функції для перевизначення оператора інкремента "++"
        // використовують посилальні параметри.
        friend kooClass operator++(kooClass &obi);
        friend kooClass operator++(kooClass &obi, int notused);
        void Show(char *s);
};

// Операторна "дружня" функція класу.
kooClass operator*(kooClass obi, kooClass obj)
{
        kooClass tmp;     // Створення тимчасового об'єкта
        tmp.x = obi.x * obj.x;
        tmp.y = obi.y * obj.y;
        tmp.z = obi.z * obj.z;
        return tmp;          // Повертає модифікований тимчасовий об'єкт
}
```

```
// Перевизначення оператора присвоєння "=".
kooClass kooClass::operator=(kooClass obj)
{
	x = obj.x;
	y = obj.y;
	z = obj.z;
	// Повернення модифікованого об'єкта операнда, адресованого покажчиком
	return *this;
}

/* Перевизначення префіксної форми унарного оператора інкремента "++"  з використанням
"дружньої" функції класу. Для цього необхідне використання посилального параметра. */
kooClass operator++(kooClass &obi)
{
	obi.x++;
	obi.y++;
	obi.z++;
	return obi;
}

/* Перевизначення постфіксної форми унарного оператора інкремента "++" з використанням
"дружньої" функції класу. Для цього необхідне використання посилального параметра. */
kooClass operator++(kooClass &obi, int notused)
{
	kooClass tmp = obi;
	obi.x++;
	obi.y++;
	obi.z++;
	return tmp; // Повертає модифікований тимчасовий об'єкт
}

// Відображення тривимірних координат x, y, z.
void kooClass::Show(char *s)
{
	cout << "Координати об'єкта <" << s << ">: ";
	cout << "\t\tx= " << x << ", y= " << y << ", z= " << z << endl;
}

int main()
{
	kooClass ObjA(1, 2, 3), ObjB(10, 10, 10), ObjC;

	ObjA.Show("A");
	ObjB.Show("B");

	ObjC = ObjA * ObjB;		// Множення об'єктів ObjA і ObjB
	ObjC.Show("C=A*B");

	ObjC = ObjA * ObjB * ObjC;	// Множинне множення об'єктів
	ObjC.Show("c");
```

```
        ObjC = ObjB = ObjA ;      // Множинне присвоєння об'єктів
        ObjC.Show("C=B");
        ObjB.Show("B=A");

        ++ObjC;                   // Префіксна форма операції інкремента
        ObjC.Show("++C");

        ObjC++;                   // Постфіксна версія інкремента
        ObjC.Show("C++");

        ObjA = ++ObjC;  // Об'єкт ObjA набуває значення об'єкта ObjC після інкрементування.
        ObjA.Show("A = ++C");  // У цьому випадку об'єкти ObjA і ObjC
        ObjC.Show("C");           // мають однакові значення координат.

        ObjA = ObjC++;  // Об'єкт ObjA набуває значення об'єкта ObjC до інкрементування.
        ObjA.Show("A=C++");    // У цьому випадку об'єкти ObjA і ObjC
        ObjC.Show("C");           // мають різні значення координат.

        getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Координати об'єкта <A>:           x= 1, y= 2, z= 3
Координати об'єкта <B>:           x= 10, y= 10, z= 10
Координати об'єкта <C=A*B>:       x= 10, y= 20, z= 30
Координати об'єкта <C=A*B*C>:     x= 100, y= 400, z= 900
Координати об'єкта <C=B>:         x= 1, y= 2, z= 3
Координати об'єкта <B=A>:         x= 1, y= 2, z= 3
Координати об'єкта <++C>:         x= 2, y= 3, z= 4
Координати об'єкта <C++>:         x= 3, y= 4, z= 5
Координати об'єкта <A=++C>:       x= 4, y= 5, z= 6
Координати об'єкта <C>:           x= 4, y= 5, z= 6
Координати об'єкта <A=C++>:       x= 4, y= 5, z= 6
Координати об'єкта <C>:           x= 5, y= 6, z= 7
```

Нео! хідноапам'ятати!а*Для перевизначення бінарних і унарних операторів потрібно використовувати безпосередньо функції-члени класу. Функції-"друзі" класу використовуються мовою програмування С++ в основному для оброблення спеціальних ситуацій.*

4.2.3. Перевизначення операторів відношення та логічних операторів

Оператори відношення (наприклад, "==", "<", ">", "<=", ">=", "!=") і логічні оператори (наприклад, "&&" або "||") також можна перевизначати, причому механізм їх реалізації не представляє жодних труднощів. Як правило, перевизначена операторна функція відношення повертає об'єкт того класу, для якого вона перевизначається. А будь-який перевизначений оператор відношення або логічний оператор повертає одне з двох можливих значень: **true** або **false**. Це відповідає звичайному застосуванню цих операторів і дає змогу використовувати їх в умовних виразах.

Розглянемо приклад перевизначення операторної функції дорівнює "==" для вже розглянутого вище класу kooClass:

```
// Перевизначення операторної функції дорівнює "=="
bool kooClass::operator==(kooClass obj)
{
	if((x == obj.x) && (y == obj.y) && (z == obj.z))
		return true;
	else
		return false;
}
```

Якщо вважати, що операторна функція **operator**==() вже реалізована, то такий код програми є абсолютно коректним:

```
kooClass ObjA, ObjB;
//...
if(ObjA == ObjB) cout << "ObjA = ObjB" << endl;
else cout << "ObjA не дорівнює ObjB" << endl;
```

Оскільки операторна функція **operator**==() повертає результат типу **bool**, то її можна використовувати для керування настановою **if**. Як вправу рекомендуємо самостійно реалізувати й інші оператори відношення та логічні оператори для класу kooClass.

4.3. Особливості реалізації оператора присвоєння

У попередньому розділі ми розглянули потенційну проблему, пов'язану з передачею об'єктів функціям і поверненням їх з них. У обох випадках проблема виникла під час використання конструктора за замовчуванням, який створює побітову копію об'єкта. Згадаймо (див. роз. 3.7), раніше це питання ми вирішували шляхом створення власного конструктора копії, який точно визначає, як повинна бути створена копія об'єкта.

Подібна проблема може виникати і під час присвоєння одного об'єкта іншому. За замовчуванням об'єкт, який знаходиться з лівого боку від операції присвоєння "=", отримує побітову копію об'єкта, який знаходиться справа. До негативних наслідків це може призвести у випадках, коли при створенні об'єкт виділяє певний ресурс (наприклад, пам'ять), а потім змінює його або звільняє. Якщо після виконання операції присвоєння об'єкт змінює або звільняє цей ресурс, то другий об'єкт також змінюється, оскільки він все ще використовує його ресурс. Вирішувати це питання можна також шляхом перевизначення оператора присвоєння.

Щоб до кінця зрозуміти суть описаної вище проблеми, розглянемо таку (некоректну) програму.

Код програми 4.7. Демонстрація механізму появи помилки, яка виникає при поверненні об'єкта з функції

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен
```

```
class strClass {              // Оголошення класового типу
        char *s;
public:
        strClass() { s = 0; }
        strClass(const strClass &obj);    // Оголошення конструктора копії
        ~strClass() {if(s) delete[]s; cout << "Звільнення s-пам'яті" << endl; }
        void Show(char *c) { cout << c << s << endl; }
        void Set(char *str);
};

// Визначення конструктора копії.
strClass::strClass(const strClass &obj)
{
        s = new char[strlen(obj.s)+1];
        strcpy(s, obj.s);
}

// Завантаження рядка.
void strClass::Set(char *str)
{
        s = new char[strlen(str)+1];
        strcpy(s, str);
}

// Ця функція повертає об'єкт типу strClass.
strClass Init()
{
        char str[80];
        strClass obj;

        cout << "Введіть рядок: "; cin >> str;
        obj.Set(str);
        return obj;
}

int main()
{
        strClass Obj;         // Створення об'єкта класу
        // Присвоюємо об'єкт, повернутий функцією Init(), об'єкту Obj.
        Obj = Init();         // Ця настанова генерує помилку!!!!
        Obj.Show("s= ");
        getch(); return 0;
}
```

Можливі результати виконання цієї програми мають такий вигляд:

Введіть рядок: Привіт
Звільнення s-пам'яті.
Звільнення s-пам'яті.
s= *тут "сміття"*
Звільнення s-пам'яті.

Залежно від використовуваного компілятора, на екрані монітора Ви можете побачити переважно "сміття" або й ні. Програма може також згенерувати помилку тривалості її виконання. У будь-якому випадку помилки не минути. І ось чому.

У цьому коді програми конструктор копії коректно обробляє повернення об'єкта функцією Init(). Згадаймо, у разі, коли функція повертає об'єкт, то для зберігання повернутого нею значення створюється тимчасовий об'єкт. Оскільки при створенні об'єкта-копії конструктор копії виділяє нову область пам'яті, то член-даних s початкового об'єкта і член-даних s об'єкта-копії вказуватимуть на різні області пам'яті, які, як наслідок, не стануть псувати один одного.

Проте помилки не минути, якщо повернутий функцією об'єкт присвоюється об'єкту Obj, оскільки у процесі виконання операції присвоєння за замовчуванням створюється побітова його копія. У цьому випадку тимчасовий об'єкт, який повертається функцією Init(), копіюється в об'єкт Obj. Як наслідок, член obj.s вказує на ту ж саму область пам'яті, що і член s тимчасового об'єкта. Але після виконання операції присвоєння в процесі руйнування тимчасового об'єкта ця пам'ять звільняється. Отже, член obj.s тепер вказуватиме на вже звільнену пам'ять! Окрім цього, пам'ять, яка адресується членом obj.s, повинна бути звільнена і після завершення роботи коду програми, тобто удруге. Щоб запобігти цьому, необхідно перевизначити оператор присвоєння так, щоб об'єкт, який розташовується зліва від оператора присвоєння, виділяв власну область пам'яті.

Реалізацію цього рішення покажемо у такій відкоректованій програмі.

Код програми 4.8. Демонстрація механізму появи помилки, яка може виникнути при поверненні об'єкта з функції

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен

class strClass {             // Оголошення класового типу
    char *s;
public:
    strClass();              // Оголошення звичайного конструктора
    strClass(const strClass &obj);   // Оголошення конструктора копії
    ~strClass() { if(s) delete[]s; cout << "Звільнення s-пам'яті" << endl; }
    void Show(char *c) { cout << c << s << endl; }
    void Set(char *str);
    // Перевизначений оператор присвоєння
    strClass operator=(const strClass &obj);
};

// Визначення звичайного конструктора.
strClass::strClass()
{
    s = new char ('\0'); // Член s вказує на NULL-рядок.
}
// Визначення конструктора копії.
strClass::strClass(const strClass &obj)
```

```
{
	s = new char[strlen(obj.s)+1];
	strcpy(s, obj.s);
}

// Завантаження рядка.
void strClass::Set(char *str)
{
	s = new char[strlen(str)+1];
	strcpy(s, str);
}

// Перевизначення оператора присвоєння "=".
strClass strClass::operator=(const strClass &obj)
{
	/* Якщо виділена область пам'яті має недостатній
	розмір, виділяється нова область пам'яті. */
	if(strlen(obj.s) > strlen(s)) {
		delete[]s;
		s = new char[strlen(obj.s)+1];
	}
	strcpy(s, obj.s);

	// Повернення модифікованого об'єкта операнда, адресованого покажчиком
	return *this;
}

// Ця функція повертає об'єкт типу strClass.
strClass Init()
{
	strClass obj; char str[80];

	cout << "Введіть рядок: "; cin >> str;
	obj.Set(str);

	return obj;
}

int main()
{
	strClass Obj;		// Створення об'єкта класу

	// Присвоюємо об'єкт, повернутий функцією Init(), об'єкту Obj
	Obj = Init();		// Тепер тут все гаразд!
	Obj.Show("s= ");

	getch(); return 0;
}
```

Ця програма тепер відображає такі результати (у припущенні, що на пропозицію "Введіть рядок: " Ви введете "Привіт").

```
Введіть рядок: Привіт
Звільнення s-пам'яті.
Звільнення s-пам'яті.
Звільнення s-пам'яті.
Привіт
Звільнення s-пам'яті.
```

Як бачимо, ця програма тепер працює коректно. Спробуйте детально проаналізувати програму і зрозуміти, чому виводиться кожне з повідомлень "Звільнення s-пам'яті.".[1]

4.4. Механізми перевизначення оператора індексації елементів масиву "[]"

На додаток до традиційних перевизначених операторів мова програмування С++ дає змогу перевизначати і більш "екзотичні", наприклад, оператор індексації елементів масиву "[]". У мові програмування С++ (з погляду механізму перевизначення) оператор "[]" вважається бінарним. Його можна перевизначати тільки для класу і тільки з використанням функції-члена класу. Ось як виглядає загальний формат операторної функції-члена класу **operator**[]().

```
тип ім'я_класу::operator[](int індекс)
{
    //...
}
```

Формально параметр *індекс* необов'язково повинен мати тип **int**, але операторна функція **operator**[]() зазвичай використовують для забезпечення індексації елементів масивів, тому в загальному випадку як аргумент цієї функції передається цілочисельне значення.

Оператор індексації елементів масиву "[]" перевизначається як бінарний оператор.

Припустимо, нехай створено об'єкт ObjA, тоді вираз ObjA[3] перетвориться в такий виклик операторної функції **operator**[]():

```
ObjA.operator[](3);
```

Іншими словами, значення виразу, що задається в операторі індексації елементів масиву "[]", передається операторній функції **operator**[]() як безпосередньо заданий аргумент. При цьому покажчик **this** вказуватиме на об'єкт ObjA, тобто об'єкт, який здійснює виклик цієї функції.

У наведеному нижче коді програми в класі aClass оголошується масив для зберігання трьох **int**-значень. Його конструктор ініціалізує кожного члена цього масиву. Перевизначена операторна функція **operator**[]() повертає значення елемента, що задається його параметром.

[1] Підказка: одне з них викликане настановою **delete** в тілі операторної функції **operator**=().

Код програми 4.9. Демонстрація механізму перевизначення оператора індексації елементів масиву "[]"

```
#include <vcl>
#include <iostream>            // Для потокового введення-виведення
#include <conio>               // Для консольного режиму роботи
using namespace std;           // Використання стандартного простору імен

const int size = 3;

class aClass {            // Оголошення класового типу
    int aMas[size];
public:
    aClass() { for(int i=0; i<size; i++) aMas[i] = i*i; }
    int operator[](int i) {return aMas[i]; }
};

int main()
{
    aClass ObjA;

    cout << "aMas[2]= " << ObjA[2] << endl; // Відображає число 4

    cout << "Значення елементів масиву <A>:" << endl;
    for(int i=0; i<3; i++)
        cout << "aMas[" << i << "]= " << ObjA[i] << endl;

    getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
a[2]= 4
Значення елементів масиву <A>:
a[0]= 0
a[1]= 1
a[2]= 4
```

Ініціалізація масиву aMas за допомогою конструктора (у цій і наступній програмах) здійснюється тільки з ілюстративною метою. У цьому коді програми функція **operator**[]() спочатку повертає значення 3-го елемента масиву aMas. Таким чином, вираз ObjA[2] повертає число 4, яке відображається настановою **cout**. Потім у циклі виводяться усі елементи масиву.

Нео! хідноапам'ятати! *Щоб оператор індексації елементів масиву "[]" можна було використовувати як зліва, так і праворуч від оператора присвоєння, достатньо вказати значення, що повертається операторною функцією **operator[]()**, як посилання.*

Можна розробити операторну функцію **operator**[]() так, щоб оператор індексації елементів масиву "[]" можна було використовувати як зліва, так і праворуч від оператора присвоєння. Для цього достатньо вказати, що значення, що повертається операторною функцією **operator**[](), є посиланням. Цю можливість продемонстровано у наведеному нижче коді програми.

Код програми 4.10. Демонстрація механізму перевизначення оператора індексації елементів масиву "[]" як зліва, так і праворуч від оператора присвоєння

```
#include <vcl>
#include <iostream>              // Для потокового введення-виведення
#include <conio>                 // Для консольного режиму роботи
using namespace std;             // Використання стандартного простору імен

const int size = 3;

class aClass {             // Оголошення класового типу
	int aMas[size];
public:
	aClass() { for(int i=0; i<size; i++) aMas[i] = i*i; }
	int &operator[](int i) {return aMas[i]; }
};

int main()
{
	aClass ObjA;

	cout << "Значення елементів масиву <A>:" << endl;
	for(int i=0; i<3; i++)
		cout << "aMas[" << i << "]= " << ObjA[i] << endl;

	// Оператор "[]" знаходиться зліва від оператора присвоєння "=".
	ObjA[2] = 25;
	cout << endl << "aMas[2]= " << ObjA[2]; // Тепер відображається число 25.

	getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Значення елементів масиву <A>:
a[0]= 0
a[1]= 1
a[2]= 4

a[2]= 25
```

Оскільки операторна функція **operator**[]() тепер повертає посилання на елемент масиву, що індексується параметром i, то оператор індексації елементів масиву "[]" можна використовувати зліва від оператора присвоєння, що дасть змогу модифікувати будь-який елемент масиву[1].

Одна з наявних переваг перевизначення оператора індексації елементів масиву "[]" полягає у тому, що за допомогою нього ми можемо забезпечити реалізацію безпечної індексації елементів масиву. Як уже зазначалося вище, у мові програмування C++ можливий вихід за межі масиву у процесі виконання програми без від-

[1] Звичайно ж, його, як і раніше, можна використовувати і праворуч від оператора присвоєння.

повідного повідомлення (тобто без генерування повідомлення про динамічну помилку). Але, якщо створити клас, який містить масив, і надати доступ до цього масиву тільки через перевизначений оператор індексації елементів масиву "[]", то в процесі виконання програми можливе перехоплення індексу, значення якого вийшло за дозволені межі. Наприклад, наведений нижче код програми (в основу якої покладений програмний код попередньої) оснащена засобом контролю потрапляння індексу масиву в допустимий інтервал його перебування.

Код програми 4.11. Демонстрація прикладу організації безпечного масиву

```
#include <vcl>
#include <iostream>             // Для потокового введення-виведення
#include <conio>                // Для консольного режиму роботи
using namespace std;            // Використання стандартного простору імен

const int size = 3;

class aClass {              // Оголошення класового типу
        int aMas[size];
public:
        aClass() { for(int i=0; i<size; i++) aMas[i] = i*i; }
        int &operator[](int i);
};

// Забезпечення контролю потрапляння індексу масиву
// в допустимий інтервал його перебування.
int &aClass::operator[](int i)
{
        if(i<0 || i> size-1) {
                cout << endl << "Значення індексу " << i <<
                " виходить за межі допустимого інтервалу" << endl;
                getch(); exit(1);
        }
        return aMas[i];
}

int main()
{
        aClass ObjA;
        cout << "Значення елементів масиву <A>:" << endl;
        for(int i=0; i<3; i++)
                cout << "aMas[" << i << "]= " << ObjA[i] << endl;

        ObjA[2] = 25;       // Оператор "[]" знаходиться в лівій частині.
        cout << endl << "aMas[2]= " << ObjA[2];         // Відображається число 25.

        ObjA[3] = 44;       // Виникає помилка тривалості виконання, оскільки
                            // значення індексу 3 виходить за межі допустимого інтервалу.
        getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Значення елементів масиву <A>:
a[0]= 0
a[1]= 1
a[2]= 4

a[2]= 25
Значення індексу 3 виходить за межі масиву.
```

У процесі виконання настанови

```
ObjA[3] = 44;
```

операторною функцією **operator**[]() перехоплюється помилка порушення меж допустимого інтервалу перебування індексу масиву, після чого програма відразу завершується, щоб не допустити потім ніяких потенційно можливих руйнувань.

4.5. Механізми перевизначення оператора виклику функцій "()"

Можливо, найбільш інтригуючим оператором, якого можна перевизначати, є оператор виклику функції "()". Під час його перевизначення створюється не новий спосіб виклику функцій, а операторна функція, якій можна передати довільну кількість параметрів. Почнемо з такого прикладу. Припустимо, що певний клас містить наведене нижче оголошення перевизначеної операторної функції:

```
int operator()(float f, char *p);
```

І якщо у програмі створюється об'єкт obj цього класу, то настанова

```
obj(99.57, "перевизначення");
```

перетвориться в такий виклик операторної функції **operator**():

```
operator()(99.57, "перевизначення");
```

У загальному випадку при перевизначенні оператора виклику функцій "()" визначаються параметри, які необхідно передати функції **operator**(). Під час використання оператора "()" у програмі задані аргументи копіюються в ці параметри. Як завжди, об'єкт, який здійснює виклик операторної функції (obj у наведеному прикладі), адресується покажчиком **this**.

Розглянемо приклад перевизначення оператора виклику функцій "()" для класу kooClass. Тут створюється новий об'єкт класу kooClass, координати якого є результатом підсумовування відповідних значень координат об'єкта і значень, що передаються як аргументи.

Код програми 4.12. Демонстрація механізму перевизначення оператора виклику функцій "()"

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен

class kooClass {        // Оголошення класового типу
        int x, y, z;    // Тривимірні координати
public:
```

```
        kooClass() { x = y = z = 0; }
        kooClass(int c, int d, int f) {x = c; y = d; z = f; }
        kooClass operator()(int a, int b, int c);
        void Show(char *s);
};

// Перевизначення оператора виклику функцій "()".
kooClass kooClass::operator()(int a, int b, int c)
{
        kooClass tmp;     // Створення тимчасового об'єкта
        tmp.x = x + a;
        tmp.y = y + b;
        tmp.z = z + c;
        return tmp;       // Повертає модифікований тимчасовий об'єкт
}

// Відображення тривимірних координат x, y, z.
void kooClass::Show(char *s)
{
        cout << "Координати об'єкта <" << s << ">: ";
        cout << "\t\tx= " << x << ", y= " << y << ", z= " << z << endl;
}

int main()
{
        kooClass ObjA(1, 2, 3), ObjB;

        ObjB = ObjA(10, 11, 12); // Виклик функції operator()

        ObjA.Show("A");
        ObjB.Show("B");
        getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Координати об'єкта <A>:        x= 1, y= 2, z= 3
Координати об'єкта <B>:        x= 11, y= 13, z= 15
```

Не забувайте, що при перевизначенні оператора виклику функцій "()" можна використовувати параметри будь-якого типу, та і сама операторна функція **operator**() може повертати значення будь-якого типу. Вибір типу повинен диктуватися потребами конкретних програм.

4.6. Механізми перевизначення рядкових операторів

За винятком таких операторів, як **new**, **delete**, ->, ->* і "кома", решту С++-оператори можна перевизначати таким самим способом, як це було показано в попередніх прикладах. Перевизначення операторів **new** і **delete** вимагає застосування спеціальних методів, повний опис яких наведено в розд. 8 (він присвячений обробленню виняткових ситуацій). Оператори ->, ->* і "кома" – це спеціальні оператори, де-

тальний перегляд яких виходить за рамки цього навчального посібника. Читачі, яких цікавлять інші приклади перевизначення операторів, можуть звернутися до такої книги [27]. У цьому ж підрозділі розглядатимемо механізм перевизначення тільки рядкових операторів.

4.6.1. Конкатенація та присвоєння класу рядків з рядками класу

Завершуючи тему перевизначення операторів, розглянемо приклад, який часто називають квінтесенцією прикладів, присвячених вивченню механізму перевизначення операторів класу рядків. Незважаючи на те, що С++-підхід до рядків (які реалізуються у вигляді символьних масивів, що завершуються нулем, а не як окремий тип) є дуже ефективним і гнучким, проте початківці С++-програмування часто стикаються з недоліком у понятійній ясності реалізації рядків, яка наявна в таких мовах, як BASIC. Звичайно ж, цю ситуацію неважко змінити, оскільки у мові програмування С++ існує можливість визначити клас рядків, який забезпечуватиме їх реалізацію подібно до того, як це зроблено в інших мовах програмування. Правду кажучи, на початкових етапах розвитку мови програмування С++ реалізація класу рядків була забавою для програмістів. І хоча стандарт мови програмування С++ тепер визначає рядковий клас, який описано далі у цьому навчальному посібнику, проте спробуйте самостійно реалізувати простий варіант такого класу. Цей приклад наочно ілюструє потужність механізму перевизначення операторів класу рядків.

Код програми 4.13. Демонстрація механізму конкатенації та присвоєння класу рядків

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен

class strClass {  // Оголошення класового типу
	char string[80];
public:
	strClass(char *str = "") { strcpy(string, str); }
	strClass operator+(strClass obj); // Конкатенація рядків
	strClass operator=(strClass obj); // Присвоєння рядків
	// Виведення рядка
	void Show(char *s) { cout << s << string << endl; }
};
```

Як бачимо, в класі strClass оголошується закритий символьний масив string, призначений для зберігання рядка. У наведеному прикладі домовимося, що розмір рядків не перевищуватиме 79 байтів. У реальному ж класі рядків пам'ять для їх зберігання повинна виділятися динамічно, однак це обмеження зараз діяти не буде. Окрім цього, щоби не захаращувати логіку цього прикладу, ми вирішили звільнити цей клас (і його функції-члени) від контролю виходу за межі масиву. Безумовно, в будь-якій справжній реалізації подібного класу повинен бути забезпечений повний контроль за помилками.

Клас strClass має один конструктор, який можна використовувати для ініціалізації масиву string з використанням заданого значення або для присвоєння йому порожнього рядка у разі відсутності ініціалізації. У цьому класі також оголошуються два перевизначені оператори, які виконують операції конкатенації та присвоєння. Нарешті, клас strClass містить функцію Show(), яка виводить рядок на екран. Ось як виглядають коди операторних функцій **operator**+() і **operator**=():

```
// Конкатенація двох рядків
strClass strClass::operator+(strClass obj)
{
	strClass tmp;	// Створення тимчасового об'єкта
	strcpy(tmp.string, string);
	strcat(tmp.string, obj.string);
	return tmp;	// Повертає модифікований тимчасовий об'єкт
}

// Присвоєння одного рядка іншому
strClass strClass::operator=(strClass obj)
{
	strcpy(string, obj.string);
	// Повернення модифікованого об'єкта операнда, адресованого покажчиком
	return *this;
}
```

Маючи визначення цих операторних функцій, продемонструємо, як їх можна використовувати на прикладі наведеної нижче основної функції **main**():

```
int main()
{
	strClass ObjA("Всім "), ObjB("привіт"), ObjC;
	ObjA.Show("A: ");
	ObjB.Show("B: ");

	ObjC = ObjA + ObjB;
	ObjC.Show("C=A+B: ");
	getch(); return 0;
}
```

Спочатку вона конкатенує рядки (об'єкти класу strClass) ObjA і ObjB, а потім присвоює результат конкатенації рядку ObjC.

Внаслідок виконання ця програма відображає на екрані такі результати:

```
A: Привіт
B: усім
C=A+B: Привіт усім
```

4.6.2. Конкатенація та присвоєння класу рядків з рядками, що закінчуються нульовим символом

Потрібно мати на увазі, що оператори присвоєння "=" і конкатенації "+" визначено тільки для об'єктів типу strClass. Наприклад, наведена нижче настанова не працездатна, оскільки вона є спробою присвоїти об'єкту ObjA рядок, який завершується нульовим символом:

```
ObjA = "Цього поки що робити не можна.";
```

Але клас strClass, як буде показано далі, можна удосконалити і дати йому змогу виконувати такі настанови.

Для розширення переліку операцій, підтримуваних класом strClass(наприклад, щоб можна було об'єктам типу strClass присвоювати рядки з завершальним нуль-символом, або конкатенувати рядок, який завершується нульовим символом, з об'єктом типу strClass), необхідно перевизначити оператори "=" і "+" ще раз. Спочатку змінимо оголошення класу:

```
// Перевизначення рядкового класу: остаточний варіант
class strClass {  // Оголошення класового типу
	char string[80];
public:
class strClass {  // Оголошення класового типу
	char string[80];
public:
	strClass(char *str = "") { strcpy(string, str); }
	// Конкатенація об'єктів типу strClass
	strClass operator+(strClass obj);
	// Конкатенація об'єкта з рядком, що завершується нулем
	strClass operator+(char *str);
	// Присвоєння одного об'єкта типу strClass іншому
	strClass operator=(strClass obj);
	// Присвоєння рядка, що завершується нулем, об'єкту типу strClass
	strClass operator=(char *str);
	void Show(char *s) { cout << s << string << endl; }
};
```

Потім реалізуємо перевизначення операторних функцій operator+() і operator=():

```
// Присвоєння рядка об'єкту типу strClass, що завершується нулем
strClass strClass::operator=(char *str)
{
	strClass tmp;	// Створення тимчасового об'єкта

	strcpy(string, str);
	strcpy(tmp.string, string);

	return tmp;	// Повертає модифікований тимчасовий об'єкт
}

// Конкатенація рядка з об'єктом типу strClass, що завершується нулем
strClass strClass::operator+(char *str)
{
	strClass tmp;	// Створення тимчасового об'єкта

	strcpy(tmp.string, string);
	strcat(tmp.string, str);

	return tmp;	// Повертає модифікований тимчасовий об'єкт
}
```

Уважно проаналізуйте коди цих функцій. Зверніть увагу на те, що правий аргумент є не об'єктом типу strClass, а покажчиком на символьний масив, який завершується нулем, тобто звичайним C++-рядком. Але обидві ці функції повертають об'єкт типу strClass. І хоча теоретично вони могли б повертати об'єкт будь-якого іншого типу, весь сенс їх існування і полягає у тому, щоб повертати об'єкт типу strClass, оскільки результати цих операцій приймаються також об'єктами типу strClass. Перевага визначення рядкової операції, у якій як правий операнд бере участь рядок, який завершується нульовим символом, полягає у тому, що воно дає змогу писати деякі настанови в природній формі. Наприклад, наведені нижче настанови є цілком законними:

```
strClass a, b, c;
a = "Привіт усім";          // Присвоєння рядка, який завершує нулем, об'єкту
c = a + "Георгій";          // Конкатенація об'єкта з рядком, що завершується нулем
```

Наведений нижче код програми містить додаткові визначення операторів присвоєння "=" і конкатенації "+".

Код програми 4.14. Демонстрація механізму конкатенації та присвоєння класу рядків з рядками, що закінчуються нульовим символом

```
#include <vcl>
#include <iostream>             // Для потокового введення-виведення
#include <conio>                // Для консольного режиму роботи
using namespace std;            // Використання стандартного простору імен

class strClass {  // Оголошення класового типу
        char string[80];
public:
        strClass(char *str = "") { strcpy(string, str); }
        // Конкатенація об'єктів типу strClass
        strClass operator+(strClass obj);
        // Конкатенація об'єкта з рядком, що завершується нулем
        strClass operator+(char *str);
        // Присвоєння одного об'єкта типу strClass іншому
        strClass operator=(strClass obj);
        // Присвоєння рядка об'єкту типу strClass, що завершується нулем
        strClass operator=(char *str);
        void Show(char *s) { cout << s << string << endl; }
};

strClass strClass::operator+(strClass obj)
{
        strClass tmp;       // Створення тимчасового об'єкта

        strcpy(tmp.string, string);
        strcat(tmp.string, obj.string);

        return tmp;         // Повертає модифікований тимчасовий об'єкт
}

strClass strClass::operator=(strClass obj)
```

```
{
	strcpy(string, obj.string);
	// Повернення модифікованого об'єкта операнда, адресованого покажчиком
	return *this;
}

strClass strClass::operator=(char *str)
{
	strClass tmp;	// Створення тимчасового об'єкта

	strcpy(string, str);
	strcpy(tmp.string, string);

	return tmp;	// Повертає модифікований тимчасовий об'єкт
}

strClass strClass::operator+(char *str)
{
	strClass tmp;	// Створення тимчасового об'єкта

	strcpy(tmp.string, string);
	strcat(tmp.string, str);
	return tmp;	// Повертає модифікований тимчасовий об'єкт
}

int main()
{
	strClass ObjA("Привіт "), ObjB("всім"), ObjC;

	ObjA.Show("A: ");
	ObjB.Show("B: ");

	ObjC = ObjA + ObjB;
	ObjC.Show("C=A+B: ");

	ObjA = "для програмування, тому що";
	ObjA.Show("A: ");

	ObjB = ObjC = "C++ це супер";
	ObjC = ObjC + " " + ObjA + " " + ObjB;
	ObjC.Show("C: ");

	getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
A: Привіт
B: усім
C=A+B: Привіт усім
A: для програмування, тому що
C: C++ це супер для програмування, тому що C++ це супер
```

Перш ніж переходити до наступного розділу, спробуйте переконатися у тому, що до кінця розумієте, як отримано ці результати. Тепер постарайтеся також самостійно визначати ще деякі інші операції над рядками. Наприклад, спробуйте визначити операцію видалення підрядка на основі оператора вилучення "-". Зокрема, якщо рядок об'єкта A містить фразу "Це важкий-важкий тест", а рядок об'єкта B – фразу "важкий", то обчислення виразу A-B дасть у підсумку "Це – тест". У цьому випадку з початкового рядка були видалені всі входження підрядка "важкий". Визначте також "дружню" функцію, яка б давала змогу рядку, що завершується нулем, знаходитися зліва від оператора конкатенації "+". Нарешті, додайте у програму код, який забезпечує контроль за помилками – виходу індексу за межі масиву.

Вартоа' нати!а*Для створюваних власних класів завжди є сенс експериментувати з перевизначенням операторів. Як показують приклади цього розділу, механізм перевизначення операторів можна використовувати для залучення нових типів даних у середовище програмування. Це один з найпотужніших засобів мови програмування C++.*

Розділ 5. ОРГАНІЗАЦІЯ МЕХАНІЗМІВ УСПАДКУВАННЯ В КЛАСАХ

Успадкування – один з трьох фундаментальних механізмів об'єктно-орієнтованого програмування, оскільки саме завдяки йому уможливлюється створення ієрархічних класифікацій. Використовуючи механізми успадкування, можна розробити загальний клас, який визначає характеристики, що є властиві множині взаємопов'язаним між собою елементам. Цей клас потім може успадковуватися іншими, вузькоспеціалізованими класами з додаванням у кожен з них своїх, властивих тільки їм унікальних особливостей.

У стандартній термінології мови програмування С++ початковий клас називається *базовим*. Клас, який успадковує базовий клас, називається *похідним*. Похідний клас можна використовувати як базовий для іншого похідного класу. За таким механізмом якраз і будується багаторівнева ієрархія класів.

5.1. Поняття про успадкування в класах

Мова програмування С++ підтримує механізм успадкування, який дає змогу в оголошенні класу вбудовувати інший клас. Для цього базовий клас задається під час оголошення похідного класу. Щоб зрозуміти сказане, почнемо з конкретного прикладу. Розглянемо клас dorZasib, який загалом визначає дорожній транспортний засіб. Його члени даних дають змогу зберігати наявну кількість коліс і можливу кількість пасажирів, яких може перевозити транспортний засіб:

```
// Оголошення класу, що визначає дорожній транспортний засіб
class dorZasib {
	int kolesa;		// Кількість коліс
	int pasagyr;		// Кількість пасажирів
public:
	void setKolesa(int f)	{ kolesa = f; }
	int getKolesa()		{ return kolesa; }
	void setPasagyr(int t)	{ pasagyr = t; }
	int getPasagyr()		{ return pasagyr; }
};
```

Таке загальне визначення дорожнього транспортного засобу є частиною визначення будь-якого конкретного типу автотранспорту. Наприклад, у наведеному нижче оголошенні класу шляхом успадкування класу dorZasib створюється клас vanAuto – вантажних автомобілів:

```
class vanAuto : public dorZasib {
	int mistkist;		// Вантажомісткість у м куб.
public:
	void setMistkist(int h)	{ mistkist = h; }
	int getMistkist()		{ return mistkist; }
```

```
        void Show(char *s);
};
```

Той факт, що клас vanAuto успадковує клас dorZasib, означає, що клас vanAuto успадковує весь вміст класу dorZasib. До вмісту класу dorZasib клас vanAuto додає свого члена даних mistkist, а також функції-члени, необхідні для його підтримки.

Зверніть увагу на те, як успадковується клас dorZasib. Загальний формат для забезпечення механізму успадкування має такий вигляд:

```
class ім'я_похідного_класу : доступ ім'я_базового_класу
{
        тіло нового класу
}
```

У такому оголошенні похідного класу елемент *доступ* є необов'язковим. У разі потреби він може бути виражений одним із специфікаторів доступу: **public**, **private** або **protected**. Ґрунтовніше про них буде сказано нижче у цьому розділі. А поки що у визначеннях усіх успадкованих класів будемо використовувати специфікатор доступу **public**. Це означає, що всі **public**-члени базового класу також будуть **public**-членами похідного класу. Отже, у наведеному вище прикладі члени класу vanAuto мають доступ до відкритих функцій-членів класу dorZasib, неначе вони (ці функції) були оголошені в тілі класу vanAuto. Проте клас vanAuto не має доступу до **private**-членів класу dorZasib. Наприклад, для класу vanAuto закритий доступ до членів даних kolesa і pasagyr.

Розглянемо код програми, яка демонструє механізм успадкування двох підкласів класу dorZasib: vanAuto і lehAuto.

Код програми 5.1. Демонстрація механізму успадкування двох підкласів

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен

// Оголошення базового класу транспортних засобів
class dorZasib {
        int kolesa;          // Кількість коліс
        int pasagyr;         // Кількість пасажирів
public:
        void setKolesa(int f)     { kolesa = f; }
        int getKolesa()           { return kolesa; }
        void setPasagyr(int t)    { pasagyr = t; }
        int getPasagyr()          { return pasagyr; }
};

// Оголошення похідного класу вантажівок.
class vanAuto : public dorZasib {
        int mistkist;        // вантажомісткість у м куб.
public:
        void setMistkist(int h)   { mistkist = h; }
        int getMistkist()         { return mistkist; }
        void Show(char *s);
};
```

```
enum type {car, van, wagon};    // Перерахунковий тип даних

// Оголошення похідного класу автомобілів.
class lehAuto : public dorZasib {
    enum type carType;
public:
    void setType(type t)        { carType = t; }
    enum type getType()         { return carType; }
    void Show(char *s);
};

void vanAuto::Show(char *s)
{
    cout << "Транспортний засіб: " << s << endl;
    cout << "коліс: " << getKolesa() << " шт" << endl;
    cout << "пасажирів: " << getPasagyr() << " осіб" << endl;
    cout << "вантажомісткість: " << mistkist << " м куб" << endl;
    cout << endl;
}

void lehAuto::Show(char *s)
{
    cout << "Транспортний засіб: " << s << endl;
    cout << "коліс: " << getKolesa() << " шт" << endl;
    cout << "пасажирів: " << getPasagyr() << " осіб" << endl;
    cout << "тип: ";
    switch(getType()) {
        case van: cout << "автофургон" << endl;
            break;
        case car: cout << "легковий" << endl;
            break;
        case wagon: cout << "фура" << endl;
    }
    cout << endl;
}

int main()
{
    vanAuto ObjT, ObjF;
    lehAuto ObjG;

    // Ініціалізація об'єкта типу вантажівка
    ObjT.setKolesa(18);
    ObjT.setPasagyr(2);
    ObjT.setMistkist(160);

    // Ініціалізація об'єкта типу вантажівка
    ObjF.setKolesa(6);
    ObjF.setPasagyr(3);
    ObjF.setMistkist(80);
```

```
        // Виведення інформації про об'єкт типу вантажівка
        ObjT.Show("Вантажівка 1");
        ObjF.Show("Вантажівка 2");

        // Ініціалізація об'єкта типу автомобіль
        ObjG.setKolesa(4);
        ObjG.setPasagyr(6);
        ObjG.setType(van);

        // Виведення інформації про об'єкт типу автомобіль
        ObjG.Show("Автомобіль");
        getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Транспортний засіб: Вантажівка 1
коліс: 18 шт.
пасажирів: 2 осіб
вантажомісткість: 160 м куб.

Транспортний засіб: Вантажівка 2
коліс: 6 шт.
пасажирів: 3 осіб
вантажомісткість: 80 м куб.

Транспортний засіб: Автомобіль
коліс: 4 шт.
пасажирів: 6 осіб
тип: автофургон
```

Як видно з результатів виконання цієї програми, основна перевага успадкування полягає у тому, що вона дає змогу створити базовий клас, який потім можна використовувати у складі похідних, більш спеціалізованих класів. Таким чином, кожен похідний клас може слугувати певній меті та, водночас, залишатися частиною загальної класифікації.

Вартоа' нати! *Зверніть увагу на те, що обидва класи vanAuto і lehAuto містять функцію-члена Show(), яка відображає інформацію про об'єкт. Ця функція демонструє ще один аспект об'єктно-орієнтованого програмування – поліморфізм. Оскільки кожна функція Show() пов'язана з власним класом, компілятор може легко "зрозуміти", яку саме функцію потрібно викликати для цього об'єкта.*

Після поверхневого ознайомлення із загальним механізмом успадкування одним класом іншого можна перейти і до вивчення конкретних його деталей.

5.2. Управління механізмом доступу до членів базового класу

Якщо один клас успадковує інший, то члени базового класу стають членами похідного. Статус доступу до членів базового класу у похідному класі визначається специфікатором доступу, який використовують для успадкування базово-

го класу. Специфікатор доступу до членів базового класу виражається одним з ключових слів: **public**, **private** або **protected**. Якщо специфікатор доступу не вказано, то за замовчуванням використовується специфікатор **private**, коли йдеться про успадкування типу **class**. Якщо ж успадковується тип **struct**, то за відсутності безпосередньо заданого специфікатора доступу використовується специфікатор **public**.

---

*Якщо базовий клас успадковується як **public**-клас, та всі його **public**-члени стають **public**-членами похідного класу.*

Розглянемо поки що раміфікацію (розгалуження) використання специфікаторів доступу до членів класу **public** або **private**[1] (рис. 5.1).

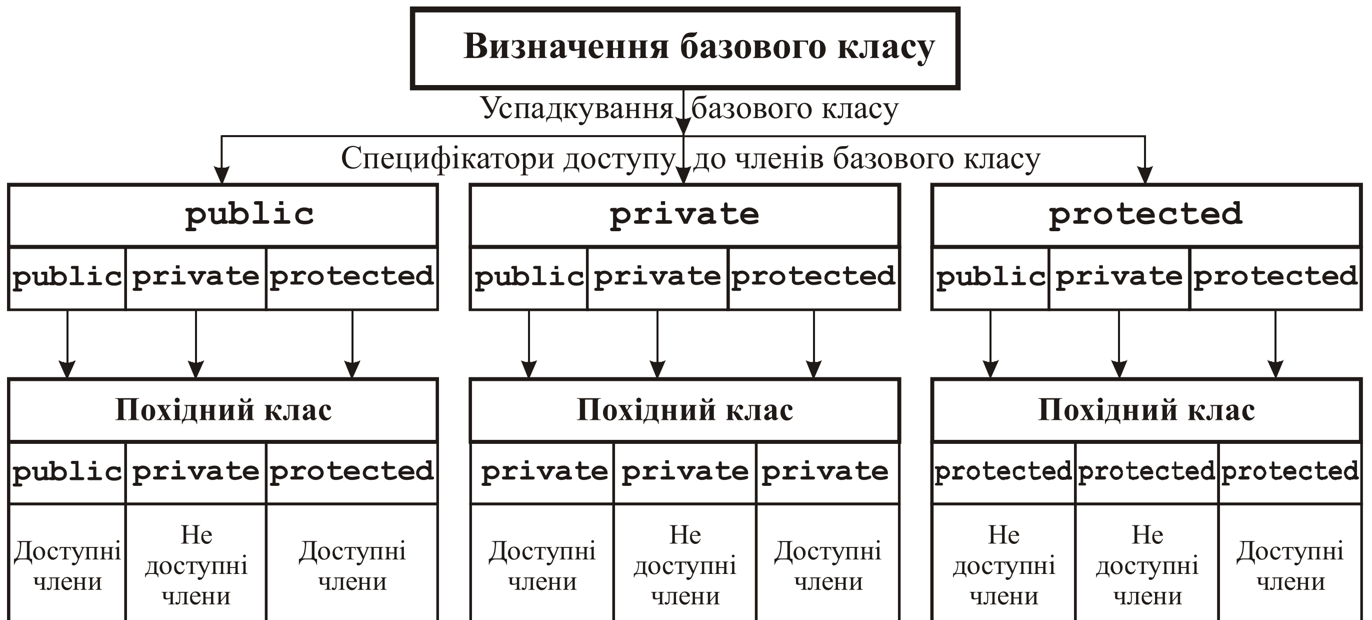

Рис. 5.1. Управління механізмом доступу до членів базового класу

В усіх випадках **private**-члени базового класу залишаються закритими у межах цього класу і не доступні для членів похідного. Наприклад, у наведеному нижче коді програми **public**-члени класу baseClass стають **public**-членами класу derived. Отже, вони будуть доступними і для інших частин програми.

Код програми 5.2. Демонстрація механізму доступу до членів базового класу після їх успадковується як public-клас

```
#include <vcl>
#include <iostream>            // Для потокового введення-виведення
#include <conio>               // Для консольного режиму роботи
using namespace std;           // Використання стандартного простору імен

class baseClass {        // Оголошення базового класу
	int c, d;
public:
	void setB(int a, int b) { c = a; d = b; }
	void showB(char *s)
		{ cout << s << "c= " << c << "; d= " << d << endl; }
};
```

---

[1] Специфікатор **protected** описано у наступному підрозділі.

```
// Оголошення похідного класу
class derived : public baseClass {
        int f;
public:
        derived(int x) { f = x; }
        void showF(char *b, char *p)
                { showB(b); cout << p << "f= " << f << endl; }
};

int main()
{
        derived ObjD(3);

        // Доступ до членів класу baseClass
        ObjD.setB(1, 2);
        // Доступ до членів класу baseClass
        ObjD.showB("Базовий клас: "); cout << endl;

        // Доступ до члена класу derived
        ObjD.showF("Базовий клас: ", "Похідний клас: ");

        getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Базовий клас: c= 1; d= 2

Базовий клас: c= 1; d= 2
Похідний клас: f= 3
```

Оскільки функції setB() і showB() (члени класу baseClass) успадковані класом derived як **public**-члени, то їх можна викликати для об'єкта типу derived у функції **main**(). Позаяк члени даних c та d визначені як **private**-члени, то вони залишаються закритими у межах свого класу baseClass.

Протилежністю відкритому (**public**) успадкуванню є закрите (**private**).

*Якщо базовий клас успадковується як **private**-клас, то всі його **public**-члени стають **private**-членами похідного класу.*

Наприклад, наведений нижче код програми не відкомпілюється, оскільки обидві функції setB() і showB() тепер стали **private**-членами класу derived (тобто, доступними тільки для функцій-членів похідного класу), і тому їх не можна викликати з функції **main**().

Код програми 5.3. Демонстрація механізму доступу до членів базового класу після їх успадковується як private-клас

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен
```

```
class baseClass {          // Оголошення базового класу
    int c, d;
public:
    void setB(int a, int b) { c = a; d = b; }
    void showB(char *s) { cout << s << "c= " << c << "; d= " << d << endl; }
};

// Відкриті члени класу baseClass тепер стають закритими членами класу derived.
class derived : private baseClass {
    int f;
public:
    derived(int x) { f = x; }
    void showF(char *b, char *p) { showB(b); cout << p << "f= " << f << endl; }
};

int main()
{
    derived ObjD(3);

    // Помилка, доступу до функції setB() немає.
    ObjD.setB(1, 2);
    // Помилка, доступу до функції showB() немає.
    ObjD.showB("Базовий клас: "); cout << endl;

    // Доступ до члена класу derived
    ObjD.showF("Базовий клас: ", "Похідний клас: ");

    getch(); return 0;
}
```

Вартоа' нати! *У випадку, коли базовий клас успадковується як* ***private****-клас, то його відкриті члени стають закритими (****private****) членами похідного класу. Це означає, що вони доступні для функцій-членів похідного класу, але не доступні для інших частин програми.*

5.3. Механізми використання захищених членів класу

Член класу може бути оголошений не тільки відкритим (**public**) або закритим (**private**), але і захищеним (**protected**). Окрім цього, базовий клас у цілому може успадковуватися з використанням специфікатора доступу **protected**. Ключове слово **protected** додане мові програмування C++ для надання механізму успадкування більшої гнучкості.

Специфікатор доступу ***protected*** *оголошує захищені члени або забезпечує механізм успадкування захищеного класу.*

Якщо член класу оголошений з використанням специфікатора доступу **protected**, то він не буде доступним для інших елементів програми, які не є членами цьо-

го класу. Доступ до захищеного члена є ідентичним механізму доступу до закритого члена, тобто до нього можуть звертатися тільки інші члени того ж самого класу. Механізм успадкування захищеного члена класу істотно відрізняється від закритого члена класу.

5.3.1. Використання специфікатора доступу **protected** для надання членам класу статусу захищеності

Як уже зазначалося вище, закриті члени базового класу не доступні ніяким іншим частинам програми, окрім функцій-членів свого класу, в т.ч. йдеться і про похідні класи. Проте із захищеними членами класу все відбувається інакше. Якщо базовий клас успадковується як **public**-клас, то захищені члени базового класу стають захищеними членами похідного класу, тобто доступними для нього. Отже, використовуючи специфікатор доступу **protected**, можна створити члени класу, які закриті у межах свого класу, але можуть успадковуватися похідним класом з отриманням доступу до себе. Для розуміння сказаного розглянемо такий приклад програми.

Код програми 5.4. Демонстрація механізму доступу до захищених членів базового класу

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен

class baseClass {         // Оголошення базового класу
protected:
    int c, d;    // Ці члени закриті у класі baseClass але доступні для класу derived.
public:
    void setB(int a, int b) { c = a; d = b; }
    void showB(char *s) { cout << s << "c= " << c << "; d= " << d << endl; }
};

// Оголошення похідного класу
class derived : public baseClass {
    int f;
public:
    // Клас derived має доступ до членів класу baseClass c та d.
    void setF()                  { f = c*d; }
    void showF(char *b, char *p) { showB(b); cout << p << "f= " << f << endl; }
};

int main()
{
    derived ObjD;      // Створення об'єкта класу

    // OK, класу derived це робити дозволено.
    ObjD.setB(2, 3);
```

```
        // OK, класу derived це робити дозволено.
        ObjD.showB("Базовий клас: "); cout << endl;

        // Функція setF() належить класу derived.
        ObjD.setF();
        // Функція showF() належить класу derived.
        ObjD.showF("Базовий клас: ", "Похідний клас: ");
        getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Базовий клас: c= 2; d= 3

Базовий клас: c= 2; d= 3
Похідний клас: f= 6
```

Оскільки клас baseClass успадкований класом derived відкритим способом (тобто як **public**-клас), а члени-даних c та d оголошені захищеними у класі baseClass, то функція setF() (член класу derived) може отримувати до них доступ. Якби члени-даних c та d були оголошені у класі baseClass закритими, то клас derived не міг би звертатися до них і ця програма не скомпілювалася б.

Нео! хідноапам'ятати!а*Специфікатор доступу* **protected** *дає змогу створити член класу, який буде доступним у рамках даної ієрархії класів, але є закритим для решти елементів програми.*

Якщо деякий похідний клас використовується як базовий для іншого похідного класу, то будь-який захищений член початкового базового класу, який успадковується (відкритим способом) першим похідним класом, може успадковуватися ще раз (як захищений член) другим похідним класом. Наприклад, у наступній (цілком коректній) програмі клас derivedB має повноправний доступ до членів-даних c та d початкового базового класу.

Код програми 5.5. Демонстрація механізму доступу до захищених членів базового класу похідними класами

```
#include <vcl>
#include <iostream>             // Для потокового введення-виведення
#include <conio>                // Для консольного режиму роботи
using namespace std;            // Використання стандартного простору імен

class baseClass {         // Оголошення базового класу
protected:
        int c, d;
public:
        void setB(int a, int b) { c = a; d = b; }
        void showB(char *s) { cout << s << "c= " << c << "; d= " << d << endl; }
};

// Члени c та d успадковуються як protected-члени.
class derivedA : public baseClass {
        int f;
```

```
public:
        void setF() { f = c*d; }      // Правомірний доступ
        void showF(char *b, char *p) { showB(b); cout << p << "f= " << f << endl; }
};

// Члени c та d успадковуються опосередковано через клас derivedA.
class derivedB : public derivedA {
        int h;
public:
        void setM() { h = c + d; } // Правомірний доступ
        void showM(char *b, char *p1, char *p2)
                { showF(b, p1); cout << p2 << "h= " << h << endl; }
};

int main()
{
        derivedA ObjA;    // Створення об'єкта класу
        derivedB ObjB;    // Створення об'єкта класу

        ObjA.setB(2, 3);
        ObjA.showB("Базовий клас: "); cout << endl;
        ObjA.setF();
        ObjA.showF("Базовий клас: ", "Похідний клас: "); cout << endl;

        ObjB.setB(3, 4);
        ObjB.showB("Базовий клас: "); cout << endl;

        ObjB.setF();
        ObjB.setM();
        ObjB.showF("Базовий клас: ", "Похідний клас 1: "); cout << endl;
        ObjB.showM("Базовий клас: ", "Похідний клас 1: " , "Похідний клас 2: ");

        getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Базовий клас: c= 2; d= 3

Базовий клас: c= 2; d= 3
Похідний клас: f= 6

Базовий клас: c= 3; d= 4
Базовий клас: c= 3; d= 4
Похідний клас 1: f= 12

Базовий клас: c= 3; d= 4
Похідний клас 1: f= 12
Похідний клас 2: h= 7
```

Якщо базовий клас успадковується як закритий (тобто з використанням специфікатора доступу **private**), то захищені (derived) члени базового класу стають зак-

ритими (**private**) членами похідного класу. Отже, якби у наведеному вище прикладі клас baseClass успадковувався закритим способом, то всі його захищені члени стали б **private**-членами класу derivedA, і у цьому випадку вони не були б доступні для класу derivedB[1]. Цю ситуацію продемонстровано у наведеному нижче коді програми, яка через це є некоректною і не відкомпілюється. Всі помилки відзначені у коментарях.

Код програми 5.6. Демонстрація некоректного використання закритих членів

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен

class baseClass {       // Оголошення базового класу
protected:
    int c, d;
public:
    void setB(int a, int b) { c = a; d = b; }
    void showB(char *s) { cout << s << "c= " << c << "; d= " << d << endl; }
};

// Елементи класу baseClass будуть закриті у межах класу derivedA.
class derivedA : private baseClass {
    int f;
public:
    // Виклики цих функцій цілком законні, оскільки змінні c та d
    // є одночасно private-членами класу derivedA.
    void setF() { f = c*d; } // OK
    void showF(char *b, char *p)
        { showB(b); cout << p << "f= " << f << endl; }
};

// Доступ до членів c, d, setB() і showB() не успадковується.
class derivedB : public derivedA {
    int h;
public:
    // Неправильно, оскільки члени-даних c та d закриті у межах класу derivedA.
    void setM()        { h = c + d; } // Помилка
    void showM(char *b, char *p1, char *p2)
        { showF(b, p1); cout << p2 << "h= " << h << endl; }
};

int main()
{
    derivedA ObjA;   // Створення об'єкта класу
    derivedB ObjB;   // Створення об'єкта класу
```

[1] Проте члени c та d, як і раніше, залишаються доступними для класу derivedA.

```
        // Помилка: не можна викликати функцію setB()
        ObjA.setB(1, 2);
        // Помилка: не можна викликати функцію showB()
        ObjA.showB("Базовий клас: "); cout << endl;

        // Помилка: не можна викликати функцію setB()
        ObjB.setB(3, 4);
        // Помилка: не можна викликати функцію showB()
        ObjB.showB("Базовий клас: ");

        getch(); return 0;
}
```

Незважаючи на те, що клас baseClass успадковується класом derivedA закритим способом, однак клас derivedA має доступ до **public**- і **protected**-членів класу baseClass. Однак цей привілей він не може передати далі, тобто вниз за ієрархією класів. Ключове слово **protected** – це частина мови С++. Воно забезпечує захист певних елементів класу від модифікації функціями, які не є членами цього класу, але дає змогу передавати їх "за успадкуванням".

Специфікатор доступу **protected** можна також використовувати стосовно структур. Але його не можна застосовувати до об'єднань, оскільки об'єднання не успадковується іншим класом. Деякі компілятори допускають використання специфікатора доступу **protected** в оголошенні об'єднання, але, оскільки об'єднання не можуть брати участь в успадкуванні, то у цьому контексті ключове слово **protected** буде рівносильним ключовому слову **private**.

Специфікатор захищеного доступу може знаходитися в будь-якому місці оголошення класу, але, як правило, **protected**-члени оголошуються після (оголошуваних за замовчуванням) **private**-членів і перед **public**-членами. Таким чином, найзагальніший формат оголошення класу зазвичай має такий вигляд:

```
class ім'я_класу {
        private-члени
protected:
        protected-члени
public:
        public-члени
};
```

Нагадаємо, що розділ захищених членів є необов'язковим.

5.3.2. Використання специфікатора доступу protected для успадкування базового класу

Специфікатор доступу **protected** можна використовувати не тільки для надання членам класу статусу "захищеності", але і для успадкування базового класу. Якщо базовий клас успадковується як захищений, то всі його відкриті та закриті члени стають захищеними членами похідного класу. Для розуміння сказаного розглянемо такий приклад.

Код програми 5.7. Демонстрація механізму використання специфікатора доступу protected для успадкування базового класу

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен

class baseClass {      // Оголошення базового класу
	int c;      // Закритий
protected:
	int d;      // Захищений
public:
	int f;      // Відкритий
	void Set(int a)    { c = a; }
	int Put()          { return c; }
};

// Успадковуємо клас baseClass як protected-клас.
class derived : protected baseClass {
public:
	void setJ(int a) { d = a; } // d – тут protected-член
	void setF(int a) { f = a; }  // f – тут protected-член
	int getJ() { return d; }
	int getK() { return f; }
};

int main()
{
	derived ObjD;      // Створення об'єкта класу
	/* Наступний рядок неправомірний, оскільки функція Set() є
	protected-членом класу derived, що робить його недоступним за його межами. */
	ObjD.Set(10);

	cout << ObjD.Put();      // Неправильно, оскільки функція Put() -- protected-член.

	ObjD.f = 10;             // Неправильно, оскільки змінна f - protected-член

	// Наступні настанови правомірні.
	ObjD.setF(10);
	cout << ObjD.getK() << " ";
	ObjD.setJ(12);
	cout << ObjD.getJ() << " ";

	getch(); return 0;
}
```

Як зазначено у коментарях до цієї програми, члени (класу baseClass) f, d, Set() і Put() стають **protected**-членами класу derived. Це означає, що до них не можна отримати доступу з коду програми, "прописаного" поза класом derived. Тому посилання на ці члени у функції **main**() (через об'єкт ObjD) неправомірні.

5.3.3. Узагальнення інформації про використання специфікаторів доступу public, protected і private

Оскільки права доступу, що визначаються специфікаторами доступу **public**, **protected** і **private**, є принциповими для створення програм мовою C++, то є сенс узагальнити все те, що було сказано про ці ключові слова.

Оголошення членів класу:

- у випадку оголошення члена класу відкритим (з використанням ключового слова **public**), то до нього можна отримати доступ з будь-якої іншої частини програми;
- якщо член класу оголошується закритим (за допомогою специфікатора доступу **private**), то до нього можуть отримувати доступ тільки члени того ж самого класу. Понад це, до закритих членів базового класу не мають доступу навіть похідні класи;
- якщо член класу оголошується захищеним (тобто **protected**-членом), то до нього можуть отримувати доступ тільки члени того ж самого класу або похідних від нього. Специфікатор доступу **protected** дає змогу успадковувати члени, але залишає їх закритими у межах ієрархії класів.

Успадкування базових класів:

- якщо базовий клас успадковується з використанням ключового слова **public**, то його **public**-члени стають **public**-членами похідного класу, а його **protected**-члени – **protected**-членами похідного класу;
- якщо базовий клас успадковується з використанням ключового слова **private**, то його **public**- і **protected**-члени стають **private**-членами похідного класу;
- якщо базовий клас успадковується з використанням специфікатора доступу **protected**, то його **public**- і **protected**-члени стають **protected**-членами похідного класу;
- в усіх випадках **private**-члени базового класу залишаються закритими у межах цього класу і не успадковуються.

У міру збільшення досвіду створення програм мовою C++ застосування специфікаторів доступу **public**, **protected** і **private** не завдаватиме Вам клопоту. А поки що, коли Вам ще не вистачає упевненості у правильності використання того або іншого специфікатора доступу, спробуйте написати просту експериментальну програму і проаналізуйте отримані результати.

5.4. Механізми успадкування декількох базових класів

Похідний клас може успадковувати два або більше базових класів. Наприклад, у цій короткій програмі клас derived успадковує обидва класи baseA і baseB.

Код програми 5.8. Демонстрація механізму успадкування декількох базових класів

```
#include <vcl>
#include <iostream>        // Для потокового введення-виведення
#include <conio>           // Для консольного режиму роботи
using namespace std;       // Використання стандартного простору імен

class baseA {   // Оголошення базового класу
protected:
    int x;
public:
```

```
    void showX() { cout << x << endl; }
};

class baseB {    // Оголошення базового класу
protected:
    int y;
public:
    void showY() { cout << y << endl; }
};

// Успадкування двох базових класів.
// Оголошення похідного класу
class derived : public baseA, public baseB {    // Оголошення базового класу
public:
    void setXY(int c, int d) {x = c; y = d; }
};

int main()
{
    derived ObjD;       // Створення об'єкта класу

    ObjD.setXY(10, 20);        // Член класу derived
    ObjD.showX();              // Функція з класу baseA
    ObjD.showY();              // Функція з класу baseB

    getch(); return 0;
}
```

Як видно з цього коду програми, щоб забезпечити успадкування декількох базових класів, необхідно через кому перерахувати їх імена у вигляді списку. При цьому потрібно вказати специфікатор доступу для кожного успадкованого базового класу.

5.5. Особливості використання конструкторів і деструкторів при реалізації механізму успадкування

Під час реалізації механізму успадкування зазвичай виникають два важливі запитання, пов'язані з використання конструкторів і деструкторів:

- коли викликаються конструктори і деструктори базового і похідного класів?
- як можна передати параметри конструктору базового класу?

Відповіді на ці запитання викладено нижче у цьому підрозділі.

5.5.1. Послідовність виконання конструкторів і деструкторів

Базовий і/або похідний клас може містити конструктор і/або деструктор. Важливо розуміти послідовність, у якому виконуються ці функції при створенні об'єкта похідного класу і його (об'єкта) руйнування. Для розуміння сказаного розглянемо таку навчальну програму.

Код програми 5.9. Демонстрація послідовності виконання конструкторів і деструкторів

```
#include <vcl>
#include <iostream>            // Для потокового введення-виведення
#include <conio>               // Для консольного режиму роботи
using namespace std;           // Використання стандартного простору імен

class baseClass {       // Оголошення базового класу
public:
	baseClass()     { cout << "Створення baseClass-об'єкта" << endl; }
	~baseClass()    { cout << "Руйнування baseClass-об'єкта" << endl; }
};

// Оголошення похідного класу
class derived : public baseClass {
public:
	derived() { cout << "Створення derived-об'єкта" << endl; }
	~derived() { cout << "Руйнування derived-об'єкта" << endl; }
};

int main()
{
	derived ObjD;     // Створення об'єкта класу

	// Ніяких дій, окрім створення і руйнування об'єкта ObjD.

	getch(); return 0;
}
```

Як зазначено у коментарях для функції **main**(), ця програма тільки створює і відразу руйнує об'єкт ObjD, який має тип derived. Внаслідок виконання ця програма відображає на екрані такі результати:

```
Створення baseClass-об'єкта.
Створення derived-об'єкта.
Руйнування derived-об'єкта.
Руйнування baseClass-об'єкта.
```

Аналізуючи отримані результати, бачимо, що спочатку виконується конструктор класу baseClass, а за ним – конструктор класу derived. Потім (через негайне руйнування об'єкта ObjD у цьому коді програми) викликається деструктор класу derived, а за ним – деструктор класу baseClass.

Результати описаного вище експерименту можна узагальнити. При створенні об'єкта похідного класу спочатку викликається конструктор базового класу, а за ним – конструктор похідного класу. Під час руйнування об'єкта похідного класу спочатку викликається його "рідний" конструктор, а за ним – конструктор базового класу.

Конструктори викликаються у порядку походження класів, а деструктори – у зворотному порядку.

Цілком логічно, що функції конструкторів виконуються у порядку походження їх класів. Оскільки базовий клас "нічого не знає" ні про який похідний клас, операції з ініціалізації, які йому потрібно виконати, не залежать від операцій ініціалізації, що виконуються похідним класом, але, можливо, створюють попередні умови для подальшої роботи. Тому конструктор базового класу повинен виконуватися першим.

Така ж логіка простежується і у тому, що деструктори виконуються у порядку, зворотному порядку походження класів. Оскільки базовий клас знаходиться в основі похідного класу, то руйнування першого передбачає руйнування другого. Отже, деструктор похідного класу є сенс викликати до того, як об'єкт буде повністю зруйнований.

При розширеній ієрархії класів (тобто за ситуації, коли похідний клас стає базовим класом для ще одного похідного) застосовується таке загальне правило: *конструктори викликаються у порядку походження класів, а деструктори – у зворотному порядку*. Для розуміння сказаного розглянемо таку навчальну програму.

Код програми 5.10. Демонстрація послідовності виконання конструкторів і деструкторів при розширеній ієрархії класів

```
#include <vcl>
#include <iostream>              // Для потокового введення-виведення
#include <conio>                 // Для консольного режиму роботи
using namespace std;             // Використання стандартного простору імен

class baseClass {       // Оголошення базового класу
public:
	baseClass()       { cout << "Створення baseClass-об'єкта" << endl; }
	~baseClass()      { cout << "Руйнування baseClass-об'єкта" << endl; }
};

// Оголошення похідного класу
class derivedA : public baseClass {
public:
	derivedA()        { cout << "Створення derivedA-об'єкта" << endl; }
	~derivedA()       { cout << "Руйнування derivedA-об'єкта" << endl; }
};

// Оголошення похідного класу
class derivedB : public derivedA {
public:
	derivedB()        { cout << "Створення derivedB-об'єкта" << endl; }
	~derivedB()       { cout << "Руйнування derivedB-об'єкта" << endl; }
};

int main()
{
	derivedB ObjB;    // Створення об'єкта класу
	// Створення і руйнування об'єкта ObjB.
	getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

Створення baseClass-об'єкта.
Створення derivedA-об'єкта.
Створення derivedB-об'єкта.
Руйнування derivedB-об'єкта.
Руйнування derivedA-об'єкта.
Руйнування baseClass-об'єкта.

Те саме загальне правило застосовується і у ситуаціях, коли похідний клас успадковує декілька базових класів.

Код програми 5.11. Демонстрація послідовності виконання конструкторів і деструкторів під час успадкування декількох базових класів

```
#include <vcl>
#include <iostream>             // Для потокового введення-виведення
#include <conio>                // Для консольного режиму роботи
using namespace std;            // Використання стандартного простору імен

class baseA {   // Оголошення базового класу
public:
    baseA()   { cout << "Створення baseA-об'єкта" << endl; }
    ~baseA()  { cout << "Руйнування baseA-об'єкта" << endl; }
};

class baseB {   // Оголошення базового класу
public:
    baseB()         { cout << "Створення baseB-об'єкта" << endl; }
    ~baseB()        { cout << "Руйнування baseB-об'єкта" << endl; }
};

// Оголошення похідного класу
class derived : public baseA, public baseB {
public:
    derived()       { cout << "Створення derived-об'єкта" << endl; }
    ~derived()      { cout << "Руйнування derived-об'єкта" << endl; }
};

int main()
{
    derived ObjD;     // Створення об'єкта класу
    // Створення і руйнування об'єкта ObjD.
    getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

Створення baseA-об'єкта.
Створення baseB-об'єкта.
Створення derived-об'єкта.
Руйнування derived-об'єкта.
Руйнування baseB-об'єкта.
Руйнування baseA-об'єкта.

Як бачите, конструктори викликаються у порядку походження їх класів, зліва направо, у порядку їх задавання у переліку успадкування для класу derived. Деструктори викликаються у зворотному порядку, справа наліво. Це означає, що якби клас baseB знаходився перед класом baseA у переліку класу derived, тобто відповідно до такої настанови:

```
class derived : public baseB, public baseA {,
```

то результати виконання попереднього коду програми були б такими:

```
Створення baseB-об'єкта.
Створення baseA-об'єкта.
Створення derived-об'єкта.
Руйнування derived-об'єкта.
Руйнування baseA-об'єкта.
Руйнування baseB-об'єкта.
```

5.5.2. Передача параметрів конструкторам базового класу

Дотепер жоден з попередніх прикладів не містив конструкторів, для яких потрібно було б передавати аргументи. У випадках, коли конструктор тільки похідного класу вимагає передачі одного або декількох аргументів, достатньо використовувати стандартний синтаксис параметризованого конструктора. Але як передати аргументи конструктору базового класу? У цьому випадку необхідно використовувати розширену форму оголошення конструктора похідного класу, у якому передбачено можливість передачі аргументів одному або декільком конструкторам базового класу.

Загальний формат розширеного оголошення конструктора похідного класу має такий вигляд:

```
конструктор_похідного_класу (перелік_аргументів):
	baseA(перелік_аргументів),
	baseB(перелік_аргументів),
	. . . . . . . . . . . .
	baseN(перелік_аргументів);
{
	тіло конструктора похідного класу
}
```

Тут елементи baseA, ..., baseN означають імена базових класів, що успадковуються похідним класом. Зверніть увагу на те, що оголошення конструктора похідного класу відділяється від переліку базових класів двокрапкою, а імена базових класів розділяються між собою комами (у разі успадкування декількох базових класів). Для розуміння сказаного розглянемо таку навчальну програму.

Код програми 5.12. Демонстрація механізму передачі параметрів конструкторам базового класу

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен
```

```
class baseClass {          // Оголошення базового класу
protected:
    int c;
public:
    baseClass(int x)  { c = x; cout << "Створення baseClass-об'єкта" << endl; }
    ~baseClass()      { cout << "Руйнування baseClass-об'єкта" << endl; }
};

// Оголошення похідного класу
class derived : public baseClass {
    int d;
public:
    // Клас derived використовує параметр x, а параметр y
    // передається конструктору класу baseClass.
    derived(int x, int y) : baseClass(y)
        { d = x; cout << "Створення derived-об'єкта" << endl; }
    ~derived() { cout << "Руйнування derived-об'єкта" << endl; }
    void showB(char *s)
        { cout << s << "c= " << c << "; d= " << d << endl; }
};

int main()
{
    derived ObjD(3, 4);

    ObjD.showB("Базовий клас: "); // Відображає числа 4 3

    getch(); return 0;
}
```

У цьому коді програми конструктор класу derived оголошується з двома параметрами x і y. Проте конструктор derived() використовує тільки параметр x, а параметр y передається конструктору baseClass(). У загальному випадку конструктор похідного класу повинен оголошувати параметри, які приймає його клас, а також ті, які потрібні базовому класу. Як це показано у наведеному вище прикладі, будь-які параметри, що потрібні базовому класу, передаються йому у переліку аргументів базового класу, які вказуються після двокрапки. Розглянемо приклад програми, у якій продемонстровано механізм передачі параметрів конструкторам декількох базових класів.

Код програми 5.13. Демонстрація механізму передачі параметрів конструкторам декількох базових класів

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен

class baseA {                // Оголошення базового класу
protected:
    int c;
```

```
public:
        baseA(int x)        { c = x; cout << "Створення baseA-об'єкта" << endl; }
        ~baseA()            { cout << "Руйнування baseA-об'єкта" << endl; }
};

// Оголошення базового класу
class baseB {    // Оголошення базового класу
protected:
        int f;
public:
        baseB(int x)        { f = x; cout << "Створення baseB-об'єкта" << endl; }
        ~baseB()            { cout << "Руйнування baseB-об'єкта" << endl; }
};

// Оголошення похідного класу
class derived : public baseA, public baseB {
        int d;
public:
        derived(int x, int y, int z): baseA(y), baseB(z)
                            { d = x; cout << "Створення derived-об'єкта" << endl; }
        ~derived() { cout << "Руйнування derived-об'єкта" << endl; }
        void showB(char *s) {cout << s << "c= " << c << "; d= " << d << "; f= " << f << endl; }
};

int main()
{
        derived ObjD(3, 4, 5);

        ObjD.showB("Базовий клас: "); // Відображає числа c= 4; d= 3; f= 5

        getch(); return 0;
}
```

Важливо розуміти, що аргументи для конструктора базового класу передаються через аргументи, які приймаються конструктором похідного класу. Навіть якщо конструктор похідного класу не використовує ніяких аргументів, то він повинен оголосити один або декілька аргументів, якщо базовий клас приймає один або декілька аргументів. У цій ситуації аргументи, що передаються похідному класу, "транзитом" передаються базовому. Наприклад, у наведеному нижче коді програми конструктори baseA() і baseB(), на відміну від конструктора класу derived, приймають аргументи.

Код програми 5.14. Демонстрація механізму передачі аргументів конструкторам базового класу через конструктори похідного класу

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен
```

```
class baseA {              // Оголошення базового класу
protected:
    int c;
public:
    baseA(int x)       { c = x; cout << "Створення baseA-об'єкта" << endl; }
    ~baseA()           { cout << "Руйнування baseA-об'єкта" << endl; }
};

class baseB {   // Оголошення базового класу
protected:
    int f;
public:
    baseB(int x)       { f = x; cout << "Створення baseB-об'єкта" << endl; }
    ~baseB()           { cout << "Руйнування baseB-об'єкта" << endl; }
};

// Оголошення похідного класу
class derived : public baseA, public baseB {
public:
    /* Конструктор класу derived не використовує параметрів, але повинен
    оголосити їх, щоб передати конструкторам базових класів. */

    derived(int x, int y) : baseA(x), baseB(y)
        { cout << "Створення derived-об'єкта" << endl; }
    ~derived() { cout << "Руйнування derived-об'єкта" << endl; }
    void showB(char *s)
        { cout << s << "c =" << c << "; f= " << f << endl; }
};

int main()
{
    derived ObjD(3, 4);

    ObjD.showB("Базовий клас: "); // Відображає числа c= 3; f= 4

    getch(); return 0;
}
```

Конструктор похідного класу може використовувати будь-які (або всі) параметри, які ним оголошені для прийняття, незалежно від того, чи передаються вони (один або декілька) базовому класу. Іншими словами, той факт, що деякий аргумент передається базовому класу, не заважає його використанню і самим похідним класом. Наприклад, наведений нижче фрагмент коду програми є абсолютно допустимим:

```
class derived : public baseClass {
    int d;
public:
    /* Клас derived використовує обидва параметри x і y,
    а також передає їх класу baseClass */
```

```
        derived(int x, int y) : baseClass(x, y)
                { d = x*y; cout << "Створення derived-об'єкта" << endl; }
        //. . .
}
```

При передачі аргументів конструкторам базового класу необхідно мати на увазі, що аргумент, який передається, може містити будь-який (дійсний на момент передачі) вираз, що містить виклики функцій і змінних. Це можливо завдяки тому, що мова С++ дає змогу виконувати динамічну ініціалізацію даних.

5.6. Повернення успадкованим членам класу початкової специфікації доступу

Коли базовий клас успадковується як закритий (як **private**-клас), то всі його члени (відкриті, захищені та закриті) стають **private**-членами похідного класу. Але за певних обставин один або декілька успадкованих членів необхідно повернути до їх початкової специфікації доступу. Наприклад, незважаючи на те, що базовий клас успадковується як **private**-клас, деяким визначеним його **public**-членам потрібно надати **public**-статус у похідному класі. Це можна зробити двома способами:

- по-перше, у похідному класі можна використовувати оголошення **using** (цей спосіб рекомендується стандартом мови С++ для використання в новому коді програми). Але особливості використання директиви **using** відкладемо до розгляду теми простору імен[1];
- по-друге, можна налагодити доступ до успадкованого члена за допомогою *оголошень доступу.*

Зауважмо, що оголошення доступу все ще підтримуються стандартом мови С++, але останнім часом активізувалися заперечення проти його застосування, а це означає, що його не варто використовувати в новому коді програми. Позаяк воно все ще використовуються у мові програмування С++, то дещо приділимо увагу цій темі.

Оголошення доступу має такий формат:

ім'я_базового_класу::член;

Оголошення доступу відновлює рівень доступу до успадкованого члена, внаслідок чого він отримує той статус, який був у нього в базовому класі.

Оголошення доступу поміщається у похідному класі під відповідним специфікатором доступу. Зверніть увагу на те, що оголошення типу у цьому випадку вказувати не потрібно. Щоб зрозуміти, як працює оголошення доступу, розглянемо спочатку такий короткий фрагмент коду програми:

```
class baseClass {
public:
        int d; // public-доступ у класі baseClass
};
```

---

[1] Основне призначення директиви `using` – забезпечити підтримку просторів імен.

```
// Клас baseClass успадковується як private-клас.
class derived : private baseClass {        // Оголошення похідного класу
public:
        // Ось механізм використання оголошення доступу:
        baseClass::d; // Тепер член d знову став відкритим.
        //. . .
};
```

Оскільки клас baseClass успадковується класом derived закритим способом, то його **public**-змінна d стає **private**-змінною класу derived. Проте внесення цього оголошення доступу

```
baseClass::d;
```

в класі derived під специфікатором доступу **public** відновлює public-статус члена d.

Оголошення доступу можна використовувати для відновлення прав доступу **public**- і **protected**-членів. Проте для зміни (підвищення або пониження) статусу доступу його використовувати не можна. Наприклад, член, оголошений закритим у базовому класі, не можна зробити відкритим у похідному[1]. Механізм використання оголошення доступу продемонстровано в такому коді програми.

Код програми 5.15. Демонстрація механізму використання оголошення доступу

```
#include <vcl>
#include <iostream>             // Для потокового введення-виведення
#include <conio>                // Для консольного режиму роботи
using namespace std;            // Використання стандартного простору імен

class baseClass {          // Оголошення базового класу
        int c; // private-член у класі baseClass
public:
        int d, f;
        void Set(int x)     { c = x; }
        int Put()           { return c; }
};

// Клас baseClass успадковується як private-клас.
class derived : private baseClass {
public:
        /* Наступні три настанови перевизначають private-успадкування класу
        baseClass і відновлюють public-статус доступу для членів d, Set() і Put(). */

        // Змінна d стає знову public-членом а змінна f залишається закритим членом.
        baseClass::d;
        baseClass::setL;  // Функція Set() стає public-членом.
        baseClass::getC; // Функція Put() стає public-членом.

        // baseClass::c;    // Неправильно: не можна підвищувати рівень доступу
        int a;              // public-член
};
```

[1] Дозвіл подібних речей зруйнував би інкапсуляцію!

```
int main()
{
	derived ObjD;	// Створення об'єкта класу

	// ObjD.c = 10;	// Неправильно, оскільки член с закритий у класі derived.
	ObjD.d = 20;	// Допустимо, оскільки член d став відкритим у класі derived
	// ObjD.f = 30;	// Неправильно, оскільки член f закритий у класі derived.
	ObjD.a = 40;	// Допустимо, оскільки член а відкрито у класі derived.
	ObjD.Set(10);
	cout << ObjD.Put() << " " << ObjD.d << " " << ObjD.a;

	getch(); return 0;
}
```

Зверніть увагу на те, як у цьому коді програми використовуються оголошення доступу для відновлення статусу **public** у членів d, Set() і Put(). У коментарях відзначені й інші обмеження, пов'язані із статусом доступу.

Мова програмування C++ забезпечує можливість відновлення рівня доступу для успадкованих членів, щоб фахівець міг успішно програмувати такі спеціальні ситуації, коли велика частина успадкованого класу повинна стати закритою, а колишній **public**- або **protected**-статус потрібно повернути тільки декільком членам. Однак до цього засобу програмування краще вдаватися тільки у крайніх випадках.

5.7. Поняття про віртуальні базові класи

Під час успадкування декількох базових класів у C++-програму може бути внесений елемент невизначеності. Для розуміння сказаного розглянемо таку некоректну програму, яка містить помилку і не відкомпілюється.

Код програми 5.16. Демонстрація невизначеності під час успадкування декількох базових класів

```
#include <vcl>
#include <iostream>		// Для потокового введення-виведення
#include <conio>		// Для консольного режиму роботи
using namespace std;	// Використання стандартного простору імен

class baseClass {	// Оголошення базового класу
public:
	int c;
};

// Клас derivedA успадковує клас baseClass.
class derivedA : public baseClass {
public:
	int d;
};

// Клас derivedB успадковує клас baseClass.
class derivedB : public baseClass {
```

```
public:
    int f;
};

/* Клас derivedC успадковує обидва класи derivedA і derivedB.
 Це означає, що у класі derivedC існує дві копії класу baseClass! */
class derivedC : public derivedA, public derivedB {
public:
    int sum;
};

int main()
{
    derivedC ObjC;

    ObjC.c = 10; // Це і є неоднозначність: який саме член c маємо на увазі???
    ObjC.d = 20;
    ObjC.f = 30;

    // І тут теж спостерігається неоднозначність з членом c.
    ObjC.sum = ObjC.c + ObjC.d + ObjC.f;

    // І тут теж неоднозначність з членом c.
    cout << ObjC.c << " ";

    cout << ObjC.d << " " << ObjC.f << " ";
    cout << ObjC.sum;

    getch(); return 0;
}
```

Як зазначено у коментарях до цієї програми, обидва класи derivedA і derivedB успадковують клас baseClass. Але клас derivedC успадковує як клас derivedA, так і клас derivedB. У результаті в об'єкті типу derivedC присутні дві копії класу baseClass, тому у такому виразі

```
ObjC.c = 20;
```

не зрозуміло, на яку саме копію члена c тут дане посилання: на член, успадкований від класу derivedA або від класу derivedB? Оскільки в об'єкті ObjC є обидві копії класу baseClass, то у ньому існують і два члени ObjC.c! Через це настанова c є успадкуванням неоднозначним (істотно невизначеним).

Є два способи виправити наведену вище програму. Перший полягає у застосуванні оператора дозволу контексту (дозволи області видимості), за допомогою якого можна "вручну" вказати потрібного члена c. Наприклад, наступна версія цієї програми успішно відкомпілюється та виконається так, як очікувалося.

Код програми 5.17. Демонстрація механізму використання оператора дозволу контексту для вибору потрібного члена

```
#include <vcl>
#include <iostream>            // Для потокового введення-виведення
```

```
#include <conio>                  // Для консольного режиму роботи
using namespace std;              // Використання стандартного простору імен

class baseClass {        // Оголошення базового класу
public:
        int c;
};

// Клас derivedA успадковує клас baseClass.
class derivedA : public baseClass {
public:
        int d;
};

// Клас derivedB успадковує клас baseClass.
class derivedB : public baseClass {
public:
        int f;
};

/* Клас derivedC успадковує обидва класи derivedA і derivedB.
 Це означає, що у класі derivedC існує дві копії класу baseClass! */
class derivedC : public derivedA, public derivedB {
public:
        int sum;
};

int main()
{
        derivedC ObjC;

        ObjC.derivedA::c = 10; // Контекст дозволений, використовується член с класу derivedA.
        ObjC.d = 20;
        ObjC.f = 30;

        // Контекст дозволений і тут.
        ObjC.sum = ObjC.derivedA::c + ObjC.d + ObjC.f;

        // Виникнення неоднозначності ліквідована і тут.
        cout << ObjC.derivedA::_x << " ";
        cout << ObjC.d << " " << ObjC.f << " ";
        cout << ObjC.sum;

        getch(); return 0;
}
```

Застосування оператора "::" дає змогу програмі "ручним способом" вибрати версію класу baseClass (успадковану класом derivedA). Але після такого рішення виникає цікаве запитання: а що, коли насправді потрібна тільки одна копія класу baseClass? Чи можна якимсь чином запобігти включенню двох копій у клас derivedC?

Відповідь, як, напевно, Ви здогадалися, позитивна. Це рішення досягається за допомогою *віртуальних базових класів.*

Якщо два (або більше) класи виведено із загального базового класу, то ми можемо запобігти включенню декількох його копій в об'єкті, виведеному з цих класів, що реалізується шляхом оголошення базового класу під час його успадкування віртуальним. Для цього достатньо, щоби імені успадкованого базового класу передувало ключове слово **virtual**.

Віртуальне успадкування базового класу гарантує, що в будь-якому похідному класі наявна тільки одна його копія.

Для ілюстрації цього процесу наведемо ще одну версію попереднього коду програми. Цього разу клас derivedC містить тільки одну копію класу baseClass.

Код програми 5.18. Демонстрація механізму застосування віртуальних базових класів

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен

class baseClass {       // Оголошення базового класу
public:
	int c;
};

// Клас derivedA успадковує клас baseClass як віртуальний.
class derivedA : virtual public baseClass {
public:
	int d;
};

// Клас derivedB успадковує клас baseClass як віртуальний.
class derivedB : virtual public baseClass {
public:
	int f;
};

/* Клас derivedC успадковує обидва класи derivedA і derivedB.
 Цього разу він містить тільки одну копію класу baseClass. */
class derivedC : public derivedA, public derivedB {
public:
	int sum;
};

int main()
{
	derivedC ObjC;

	ObjC.c = 10; // Тепер неоднозначності немає.
```

```
        ObjC.d = 20;
        ObjC.f = 30;

        // Тепер неоднозначності немає.
        ObjC.sum = ObjC.c + ObjC.d + ObjC.f;

        // Тепер неоднозначності немає.
        cout << ObjC.c << " ";

        cout << ObjC.d << " " << ObjC.f << " ";
        cout << ObjC.sum;

        getch(); return 0;
}
```

Як бачите, ключове слово **virtual** передує решті частини специфікації успадкованого класу. Тепер обидва класи derivedA і derivedB успадковують клас baseClass як віртуальний, і тому при будь-якому багаторазовому їх успадкуванні у похідний клас буде включено тільки одну його копію. Отже, у класі derivedC є тільки одна копія класу baseClass, а настанова ObjC.c = 10 тепер абсолютно допускається і не містить ніякої неоднозначності.

Необхідно пам'ятати! *Навіть якщо обидва класи derivedA і derivedB задають клас baseClass як **virtual**-клас, він, як і раніше, присутній в об'єкті будь-якого типу.*

Наприклад, така послідовність настанов цілком допускається:

```
// Оголошення похідного класу типу derivedA.
derivedA myClass;

myClass.c = 88;
```

Різниця між звичайним базовим і віртуальним класами стає очевидною тільки тоді, коли цей базовий клас успадковується більше одного разу. Якщо базовий клас оголошується віртуальним, то тільки один його примірник буде включено в об'єкт класу, що успадковується. Інакше у цьому об'єкті будуть міститися декілька його копій.

Розділ 6. ПОНЯТТЯ ПРО ВІРТУАЛЬНІ ФУНКЦІЇ ТА ПОЛІМОРФІЗМ

Однією з трьох основних особливостей об'єктно-орієнтованого програмування є поліморфізм. Стосовно мови програмування C++, під терміном *поліморфізм* розуміють механізм реалізації функції, у якому різні результати можна отримати за допомогою одного її імені. З цієї причини поліморфізм іноді характеризується фразою "один інтерфейс, багато методів". Це означає, що до всіх функцій-членів класу можна отримати доступ одним і тим самим способом, незважаючи на можливу відмінність у конкретних діях, пов'язаних з кожною окремою операцією.

У мові програмування C++ поліморфізм підтримується як у процесі виконання програми, так у період її компілювання. Перевизначення операторів і функцій – це приклади поліморфізму, що належить до моменту компілювання. Але, попри потужність механізму перевизначення операторів і функцій, він не у змозі вирішити всі завдання, які виникають в реальних додатках, розроблених з використанням об'єктно-орієнтованого програмування. Тому у мові C++ також реалізовано поліморфізм періоду виконання, який базується на використанні *похідних класів* і *віртуальних функцій*, що і становить основні теми цього розділу.

Почнемо розгляд матеріалу з короткого опису покажчиків на похідні типи, оскільки саме вони забезпечують підтримку динамічного поліморфізму.

6.1. Покажчики на похідні типи – підтримка динамічного поліморфізму

Основою для динамічного поліморфізму слугує покажчик на базовий клас. Покажчики на базові та похідні класи пов'язані такими відносинами, які не властиві покажчикам інших типів. Як було зазначено вище у цьому навчальному посібнику, покажчик одного типу, як правило, не може вказувати на об'єкт іншого типу. Проте покажчики на об'єкти базових класів і об'єкти похідних класів – винятки з цього правила.

У мові програмування C++ покажчик на базовий клас також можна використовувати для посилання на об'єкт будь-якого класу, виведеного з базового. Наприклад, припустимо, що у нас є базовий клас baseClass і похідний клас derivClass, який виведено з класу baseClass. У мові програмування C++ будь-який покажчик, оголошений як покажчик на базовий клас baseClass, може бути також покажчиком на похідний клас derivClass. Отже, після цих оголошень

```
baseClass *p;       // Створення покажчика на об'єкт базового типу
baseClass ObjB;     // Створення об'єкта базового типу
derivClass ObjD;    // Створення об'єкта похідного типу
```

обидві такі настанови є абсолютно законними:

```
p = &ObjB;          // Покажчик p вказує на об'єкт типу baseClass
p = &ObjD;          // Присвоєння покажчику адреси об'єкта похідного класу derivClass
```

```
/* Покажчик p вказує на об'єкт типу derivClass, який є об'єктом,
що був виведений з класу baseClass */
```

У наведеному прикладі покажчик p можна використовувати для доступу до всіх елементів об'єкта ObjD, що є виведеним з об'єкта ObjB. Проте до елементів, які становлять специфічну "надбудову" (над базою, тобто над базовим класом baseClass) об'єкта ObjD, доступ за допомогою покажчика p отримати не можна.

Як конкретний приклад розглянемо навчальну програму, яка визначає базовий клас baseClass і похідний клас derivClass. У цьому коді програми проста ієрархія класів використовується для зберігання імен авторів і назв їх книг.

Код програми 6.1. Демонстрація механізму використання покажчиків на базовий клас для доступу до об'єктів похідних класів

```
#include <vcl>
#include <conio>
#include <iostream>          // Для потокового введення-виведення
#include <cstring>           // Для роботи з рядковими типами даних
using namespace std;         // Використання стандартного простору імен

#include <windows>
char bufCyr[256];
char *Cyr(const char *text)
{
	CharToOem(text, bufCyr);
	return bufCyr;
}

class baseClass {            // Оголошення класового типу
	char author[80];
public:
	void putAuthor(char *s) { strcpy(author, s); }
	void showAuthor() { cout << Cyr("Аутор: ") << author << endl; }
};

class derivClass : public baseClass {    // Оголошення класового типу
	char title[80];
public:
	void putTitle(char *n) { strcpy(title, n); }
	void showTitle() { cout << Cyr("Назва: ") << title << endl; }
};

int main()
{
	baseClass *bp;          // Створення покажчика на об'єкт базового типу
	baseClass ObjB;         // Створення об'єкта базового типу
	derivClass *dp;         // Створення покажчика на об'єкт похідного типу
	derivClass ObjD;        // Створення об'єкта похідного типу
	                        // Доступ до класу baseClass через покажчик.
	bp = &ObjB;             // Присвоєння покажчику адреси об'єкта базового класу
	bp->putAuthor(Cyr("Еміль Золя"));
```

```
                            // Доступ до класу derivClass через "базовий" покажчик.
    bp = &ObjD;             // Присвоєння покажчику адреси об'єкта похідного класу
    bp->putAuthor(Cyr("Вільям Шекспір"));

    ObjB.showAuthor();      // Покажемо, що кожен автор належить до відповідного об'єкта.
    ObjD.showAuthor();
    cout << endl;

    /* Оскільки функції putTitle() і showTitle() не є частиною базового класу, то вони
    недоступні через "базовий" покажчик bp, і тому до них потрібно звертатися
    або безпосередньо, або, як показано тут, через покажчик на похідний тип. */

    dp = &ObjD;             // Присвоєння покажчику адреси об'єкта похідного класу
    dp->putTitle(Cyr("Буря"));
    dp->showAuthor();       // Тут можна використовувати або покажчик bp, або покажчик dp.
    dp->showTitle();

    getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Еміль Золя
Вільям Шекспір

Вільям Шекспір
Назва: Буря
```

У наведеному прикладі покажчик bp визначається як покажчик на базовий клас baseClass. Але він може також посилатися на об'єкт похідного класу derivClass, причому його можна використовувати для доступу тільки до тих елементів похідного класу, які успадковані від базового. Проте необхідно пам'ятати, що через "базовий" покажчик неможливо отримати доступ до тих членів, які належать похідному класу. Ось чому до функції showTitle() звернення реалізується за допомогою покажчика dp, який є покажчиком на похідний клас.

Якщо виникає потреба за допомогою покажчика на базовий клас отримати доступ до елементів, що визначені певним похідним класом, то необхідно привести цей покажчик до типу покажчика на похідний тип. Наприклад, у процесі виконання цього рядка коду програми дійсно буде викликано функцію showTitle() об'єкта ObjD:

```
((derivClass *)bp)->showTitle();
```

Зовнішній набір круглих дужок використовують для зв'язку операції приведення типу з покажчиком p, а не з типом, що повертається функцією showTitle(). Незважаючи на те, що у використанні такої операції формально немає нічого некоректного, такого запису, за змогою, необхідно уникати, оскільки усі ці прийоми просто вносять у код програми плутанину[1].

Крім цього, необхідно розуміти: хоча "базовий" покажчик можна використовувати для доступу до об'єктів будь-якого похідного типу, зворотне ж твердження

[1] Насправді більшість C++ програмістів вважає такий стиль програмування невдалим

є неправильним. Іншими словами, використовуючи покажчик на похідний клас, не можна отримати доступ до об'єкта базового типу.

Покажчик можна інкрементувати та декрементувати щодо свого базового типу. Отже, якщо покажчик на базовий клас використовують для доступу до об'єкта похідного типу, то механізм інкрементування або декрементування не примусить його посилатися на наступний об'єкт похідного класу. Натомість він вказуватиме на наступний об'єкт базового класу. Таким чином, інкрементування або декрементування покажчика на базовий клас необхідно розцінювати як некоректну операцію, якщо цей покажчик використовують для посилання на об'єкт похідного класу.

Той факт, що покажчик на базовий тип можна використовувати для посилання на будь-який об'єкт, який є виведеним з базового типу, надзвичайно важливий і принциповий для мови С++. Як буде показано нижче у цьому посібнику, ця гнучкість є ключовим моментом для механізму реалізації динамічного поліморфізму у мові програмування С++.

Посилання на похідні типи. Подібно до покажчиків, посилання на базовий клас також можна використовувати для доступу до об'єкта похідного типу. Ця можливість особливо часто застосовується при передачі аргументів функціям. Параметр, який має тип посилання на базовий клас, може приймати об'єкти базового класу, а також об'єкти будь-якого іншого типу, виведеного з нього.

6.2. Механізми реалізації віртуальних функцій

Динамічний поліморфізм можливий завдяки поєднанню двох засобів програмування: механізму успадкування і застосування віртуальних функцій. Про механізм успадкування було сказано у попередньому розділі. Тут ми познайомимося із застосуванням віртуальних функцій.

6.2.1. Поняття про віртуальні функції

Віртуальна функція – це така функція, яка оголошується в базовому класі з використанням ключового слова **virtual** і перевизначається в одному або декількох похідних класах. Таким чином, кожен похідний клас може мати власну версію віртуальної функції.

Під час застосування віртуальних функцій часто трапляються ситуації, коли віртуальна функція викликається через покажчик (або посилання) на базовий клас. У цьому випадку мова програмування С++ визначає, яку саме версію віртуальної функції необхідно викликати за типом об'єкта, який адресується цим покажчиком. Причому необхідно мати на увазі, що це рішення приймається *у процесі виконання* програми. Отже, при вказанні на різні об'єкти викликатимуться і різні версії віртуальної функції. Іншими словами, саме за типом об'єкта (а не за типом самого покажчика), який адресується, визначається, яку версію віртуальної функції буде виконано. Таким чином, якщо базовий клас містить віртуальну функцію і якщо з цього базового класу виведено два (або більше) інші класи, то при адресації різних типів об'єктів через покажчик на базовий клас виконуватимуться

і різні версії віртуальної функції. Аналогічний механізм працює і при використанні посилання на базовий клас.

Функція оголошується віртуальною в базовому класі за допомогою ключового слова **virtual**. При перевизначенні віртуальної функції у похідному класі ключове слово **virtual** повторювати не потрібно (хоча це не буде помилкою).

Цей термін також застосовується до класу, який успадковує базовий клас, що містить віртуальну функцію. Розглянемо наведену нижче навчальну програму, у якій продемонстровано механізм застосування віртуальних функцій.

Код програми 6.2. Демонстрація механізму застосування віртуальних функцій

```
#include <vcl>
#include <conio>
#include <iostream>           // Для потокового введення-виведення
using namespace std;          // Використання стандартного простору імен

class baseClass {             // Оголошення базового класу
public:
                              // Оголошення віртуальної функції
    virtual void Show() { cout << Cyr("Базовий клас.") << endl; }
};

class firstD : public baseClass {
public:
                              // Перевизначення функції Show() для класу firstD.
    void Show() { cout << Cyr("Перший похідний клас.") << endl; }
};

class secondD : public baseClass {
public:
                              // Перевизначення функції Show() для класу secondD.
    void Show() { cout << Cyr("Другий похідний клас.") << endl; }
};

int main()
{
    baseClass ObjB;           // Створення об'єкта базового типу
    baseClass *bp;            // Створення покажчика на об'єкт базового типу
    firstD ObjF;              // Створення об'єкта похідного типу
    secondD ObjS;             // Створення об'єкта похідного типу

    bp = &ObjB;               // Присвоєння покажчику адреси об'єкта базового класу
    bp->Show();               // Доступ до функції Show() класу baseClass

    bp = &ObjF;               // Присвоєння покажчику адреси об'єкта похідного класу
    bp->Show();               // Доступ до функції Show() класу firstD

    bp = &ObjS;               // Присвоєння покажчику адреси об'єкта похідного класу
    bp->Show();               // Доступ до функції Show() класу secondD
    getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Базовий клас.
Перший похідний клас.
Другий похідний клас.
```

Розглянемо детально код цієї програми, щоб зрозуміти, як вона працює.

У класі baseClass функція Show() оголошена віртуальною. Це означає, що її можна перевизначити у похідному класі (у класі, виведеному з baseClass). І вона дійсно перевизначається в обох похідних класах firstD і secondD. У основній функції **main**() оголошуються чотири змінні: ObjB – об'єкт типу baseClass, bp – покажчик на об'єкт класу baseClass, а також два об'єкти ObjF і ObjS двох похідних класів firstD і secondD відповідно. Потім покажчику bp присвоюється адреса об'єкта ObjB і викликається функція Show(). Оскільки функцію Show() оголошено віртуальною, то мова С++ у процесі виконання програми визначає, до якої саме версії функції Show() тут потрібно звернутися, причому рішення приймається шляхом аналізу типу об'єкта, який адресується покажчиком bp. У цьому випадку покажчик bp вказує на об'єкт типу baseClass, тому спочатку виконується та версія функції Show(), яку оголошено у базовому класі baseClass. Потім покажчику bp присвоюється адреса об'єкта ObjF. Як зазначалося вище, за допомогою покажчика на базовий клас можна звертатися до об'єкта будь-якого його похідного класу. Тому, коли функція Show() викликається удруге, мова С++ знову з'ясовує тип об'єкта, який адресує покажчик bp, і, виходячи з цього типу, визначає, яку версію функції Show() потрібно викликати. Оскільки покажчик bp тут вказує на об'єкт типу firstD, то виконується версія функції Show(), яку визначено у похідному класі firstD. Аналогічно після присвоєння покажчику bp адреси об'єкта ObjS викликається версія функції Show(), яку оголошено у похідному класі secondD.

Нео! хідноапам'ятати! *Те, яка версія віртуальної функції дійсно буде викликана, визначається у процесі виконання програми. Відповідне рішення ґрунтується виключно на аналізі типу об'єкта, який адресується покажчиком на базовий клас.*

Віртуальну функцію можна викликати звичайним способом (не через покажчик), використовуючи оператор "крапка" і задаючи ім'я об'єкта, який викликається. Це означає, що у наведеному вище прикладі було б синтаксично коректно звернутися до функції Show() за допомогою такої настанови:

```
ObjF.Show();
```

Проте під час виклику віртуальної функції у такий спосіб ігноруються її поліморфні атрибути. І тільки під час звернення до віртуальної функції через покажчик на базовий клас досягається динамічний поліморфізм.

Якщо віртуальна функція перевизначається у похідному класі, то її називають ***переви' наченою****.*

Спочатку може видатися дивним, що перевизначення віртуальної функції у похідному класі є спеціальною формою перевизначення функцій. Але це не так. Насправді ми маємо справу з двома принципово різними процесами. Передусім, різні версії перевизначеної функції мають відрізнятися одна від іншої типом і/або

кількістю параметрів, тоді як тип і кількість параметрів у різних версіях віртуальної функції мають точно збігатися. І справді, прототипи віртуальної та перевизначеної функцій мають бути абсолютно однаковими. Якщо прототипи будуть різними, то такі функції просто вважатимуться перевизначеними, а їх "віртуальна суть" втратиться. Окрім цього, віртуальна функція повинна бути членом класу, для якого вона визначається, а не його "другом". Водночас віртуальна функція може бути "другом" іншого класу.

Вартоа' нати! *Функціям деструкторів дозволено бути віртуальними, а функціям конструкторів – ні.*

6.2.2. Успадкування віртуальних функцій

Якщо функція оголошується віртуальною, то вона залишається такою незалежно від того, через скільки рівнів похідних класів вона може пройти. Наприклад, якби похідний клас secondD був виведений з похідного класу firstD, а не з базового класу baseClass, як це показано в наведеному вище прикладі, то функція Show(), як і раніше, залишалася б віртуальною, і механізм вибору відповідної версії теж працював би коректно. Тобто, наведений нижче приклад є коректним.

```
// Цей клас виведено з класу firstD, а не з baseClass.
class secondD : public firstD {
public:
    // Перевизначення функції Show() для класу secondD.
    void Show() { cout << Cyr("Другий похідний клас.") << endl; }
};
```

Якщо похідний клас не перевизначає віртуальну функцію, то використовується функція, яку було визначено в базовому класі. Наприклад, перевіримо, як поведеться версія попереднього коду програми, якщо у похідному класі secondD не буде перевизначено функції Show().

Код програми 6.3. Демонстрація механізму успадкування віртуальних функцій

```
#include <vcl>
#include <conio>
#include <iostream>           // Для потокового введення-виведення
using namespace std;          // Використання стандартного простору імен

class baseClass {             // Оголошення базового класу
public:
    virtual void Show() { cout << Cyr("Базовий клас.") << endl; }
};

class firstD : public baseClass {        // Клас firstD виведений з класу baseClass
public:
    void Show() { cout << Cyr("Перший похідний клас.") << endl; }
};

class secondD : public baseClass {       // Клас secondD виведений з класу baseClass
                              // Функція Show() тут взагалі не визначена.
};
```

```
int main()
{
    baseClass ObjB;       // Створення об'єкта базового типу
    baseClass *bp;        // Створення покажчика на об'єкт базового типу
    firstD ObjF;          // Створення об'єкта похідного типу
    secondD ObjS;         // Створення об'єкта похідного типу

    bp = &ObjB;           // Присвоєння покажчику адреси об'єкта базового класу
    bp->Show();           // Доступ до функції Show() класу baseClass

    bp = &ObjF;           // Присвоєння покажчику адреси об'єкта похідного класу
    bp->Show();           // Доступ до функції Show() класу firstD

    bp = &ObjS;           // Присвоєння покажчику адреси об'єкта похідного класу
    bp->Show();           /* Тут здійснюється звернення до функції Show() класу
                          baseClass, оскільки у класі secondD вона не перевизначена. */

    getch(); return 0;
}
```

Тепер у процесі виконання цієї програми на екран виводиться таке:

```
Базовий клас.
Перший похідний клас.
Базовий клас.
```

Як підтверджують отримані результати виконання цієї програми, оскільки функція Show() не перевизначена у похідному класі secondD, то під час її виклику за допомогою настанови bp->Show() (коли член p вказує на об'єкт ObjS) виконується та версія функції Show(), яку було визначено у базовому класі baseClass.

Вартоа' нати! *Успадковані властивості специфікатора **virtual** є ієрархічними. Тому, якщо попередній приклад змінити так, щоб похідний клас secondD був виведений з похідного класу firstD, а не з базового класу baseClass, то під час звернення до функції Show() через об'єкт типу secondD буде викликано ту її версію, яка оголошена у похідному класі firstD, оскільки цей клас є "найближчим" (за ієрархічними "мірками") до похідного класу secondD, а не функцію Show() з тіла базового класу baseClass. Ці ієрархічні залежності демонструються на прикладі наведеної нижче програми.*

Код програми 6.4. Демонстрація механізму ієрархії успадкування віртуальних функцій

```
#include <vcl>
#include <conio>
#include <iostream>        // Для потокового введення-виведення
using namespace std;       // Використання стандартного простору імен

class baseClass {          // Оголошення базового класу
public:
    virtual void Show() { cout << Cyr("Базовий клас.") << endl; }
};

// Клас firstD виведений з класу baseClass
```

```
class firstD : public baseClass {
public:
	void Show() { cout << Cyr("Перший похідний клас.") << endl; }
};

// Клас secondD тепер виведений з класу firstD, а не з класу baseClass.
class secondD : public firstD {
				// Функція Show() не визначена.
};

int main()
{
	baseClass ObjB;		// Створення об'єкта базового типу
	baseClass *bp;		// Створення покажчика на об'єкт базового типу
	firstD ObjF;		// Створення об'єкта похідного типу
	secondD ObjS;		// Створення об'єкта похідного типу

	bp = &ObjB;		// Присвоєння покажчику адреси об'єкта базового класу
	bp->Show();		// Доступ до функції Show() класу baseClass

	bp = &ObjF;		// Присвоєння покажчику адреси об'єкта похідного класу
	bp->Show();		// Доступ до функції Show() класу firstD

	bp = &ObjS;		// Присвоєння покажчику адреси об'єкта похідного класу
	bp->Show();		/* Тут здійснюється звернення до функції Show()
				класу firstD, оскільки у класі secondD її не перевизначено. */
	getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Базовий клас.
Перший похідний клас.
Перший похідний клас.
```

Як бачите, похідний клас secondD тепер використовує версію функції Show(), яку визначено у похідному класі firstD, оскільки вона знаходиться найближче в ієрархічному ланцюжку класів.

6.2.3. Потреба у застосуванні віртуальних функцій

Як наголошувалося на початку цього розділу, віртуальні функції у поєднанні з похідними типами дають змогу мові програмування С++ підтримувати динамічний поліморфізм. Він для об'єктно-орієнтованого програмування є важливим з однієї причини: дає змогу деякому узагальненому класу визначати функції, які використовуватимуть усі похідні від нього класи, причому похідний клас може визначити власну реалізацію усіх або деяких цих функцій. Іноді ця ідея виражається так: базовий клас диктує загальний інтерфейс, який матиме будь-який об'єкт, виведений з цього класу, але при цьому дає змогу похідному класу визначити метод, який використовують для реалізації цього інтерфейсу. Ось чому для опису поліморфізму часто використовують фраза – один інтерфейс, багато методів.

Для успішного застосування поліморфізму необхідно розуміти, що базовий і похідний класи утворюють ієрархію, розвиток якої спрямований від більшого до меншого ступеня узагальнення (тобто від базового класу до похідного). У разі коректної реалізації базовий клас забезпечує всі елементи, які похідний клас може використовувати безпосередньо. Він також визначає функції, які похідний клас повинен реалізувати самостійно. Це дає похідному класу гнучкість у визначенні власних методів, але водночас зобов'язує використовувати загальний інтерфейс. Іншими словами, оскільки формат інтерфейсу визначається базовим класом, то будь-який похідний клас повинен розділяти цей загальний інтерфейс. Таким чином, застосування віртуальних функцій дає змогу базовому класу визначати узагальнений інтерфейс, який використовуватиметься всіма похідними класами.

Тепер у Вас може виникнути запитання: чому ж такий важливий загальний інтерфейс з множиною реалізацій? Відповідь знову повертає нас до основної спонукальної причини виникнення об'єктно-орієнтованого програмування: такий інтерфейс дає змогу програмісту справлятися із наростаючою складністю програм. Наприклад, якщо коректно розробити програму, то можна упевнитися в тому, що до всіх об'єктів, виведених з базового класу, можна буде отримати доступ єдиним (загальним для всіх) способом, незважаючи на те, що конкретні дії одного похідного класу можуть відрізнятися від дій іншого. Це означає, що програмісту доведеться пам'ятати тільки один інтерфейс, а не велику їх кількість. Окрім цього, похідний клас має можливість використовувати будь-які або всі функції, надані базовим класом. Іншими словами, розробнику похідного класу не потрібно наново винаходити елементи, які вже є в базовому класі. Понад це, від'єднання інтерфейсу від реалізації дає змогу створювати бібліотеки класів, написанням яких можуть займатися сторонні організації. Коректно реалізовані бібліотеки повинні надавати загальний інтерфейс, який програміст може використовувати для виведення похідних класів відповідно до своїх конкретних потреб. Наприклад, як бібліотека базових класів Microsoft (Microsoft Foundation Classes – MFC), так і новіша бібліотека класів .NET Framework Windows Forms підтримують Windows-програмування. Використання цих класів дає змогу розробляти програми, які можуть успадкувати багато функцій, потрібних будь-якій Windows-програмі. Вам знадобиться тільки додати в неї засоби, унікальні для Вашої програми. Це – велика допомога при програмуванні складних ієрархічних систем.

6.2.4. Приклад застосування віртуальних функцій

Щоб Ви могли отримати уявлення про потужність механізму "один інтерфейс, багато методів", розглянемо таку навчальну програму. Вона створює базовий клас figure, призначений для зберігання розмірів різних двовимірних об'єктів і обчислення їх площ. Функція Set() є стандартною функцією-членом класу, оскільки вона підходить для всіх похідних класів. Проте функція Show() оголошена як віртуальна, оскільки методики обчислення площі різних об'єктів будуть різними. Програма використовує базовий клас figure для виведення двох спеціальних класів rectangle і triangle.

Код програми 6.5. Демонстрація механізму застосування віртуальних функцій

```
#include <vcl>
#include <conio>
#include <iostream>               // Для потокового введення-виведення
using namespace std;              // Використання стандартного простору імен

class figure {
protected:
        double x, y;
public:
        void Set(double _x, double _y) { x = _x; y = _y; }
        virtual void Show()
        {
                cout << Cyr("У цьому класі виразу для обчислення площі не визначено.") << endl;
        }
};

class triangle : public figure {
public:
        void Show()
        {
                cout << Cyr("Трикутник з висотою ") << x; cout << Cyr(" і основою ") << y;
                cout << Cyr(" має площу ") << x * 0.5 * y ; cout << Cyr(" кв. од.") << endl;
        }
};

class rectangle : public figure {
public:
        void Show()
        {
                cout << Cyr("Прямокутник розмірами ") << x << " x " << y;
                cout << Cyr(" має площу ") << x * y ; cout << Cyr(" кв. од.") << endl;
        }
};

int main()
{
        figure *p;                // Створення покажчика на об'єкт базового типу
        triangle ObjT;            // Створення об'єктів похідних типів
        rectangle ObjR;           // Створення об'єкта похідного типу

        p = &ObjT;                // Присвоєння покажчику адреси об'єкта похідного класу
        p->Set(10.3, 5.5);
        p->Show();

        p = &ObjR;                // Присвоєння покажчику адреси об'єкта похідного класу
        p->Set(10.3, 5.5);
        p->Show();
        getch(); return 0;
}
```

Ось як виглядають результати виконання цієї програми:

Трикутник з висотою 10.3 і основою 5.5 має площу 28.325 кв. од.
Прямокутник розмірами 10.3 x 5.5 має площу 56.65 кв. од.

У цьому коді програми зверніть увагу на те, що під час роботи з класами rectangle і triangle використано однаковий інтерфейс, хоча у кожному з них реалізовано власні методики обчислення площі відповідних об'єктів.

Як Ви думаєте, використовуючи оголошення базового класу figure, можна вивести похідний клас circle для обчислення площі круга за заданим значенням радіуса? Відповідь: так. Для цього достатньо створити новий похідний тип, який би обчислював площу круга. Потужність віртуальних функцій опирається на той факт, що програміст може легко вивести новий тип, який розділятиме загальний інтерфейс з іншими "спорідненими" об'єктами. Ось, наприклад, як це можна зробити в нашому випадку:

```
class circle : public figure {
public:
	void Show()
	{
		cout << Cyr("Круг з радіусом ") << x;
		cout << Cyr(" має площу ") << 3.14 * x * x << endl;
	}
};
```

Перш ніж випробувати клас circle в роботі, розглянемо уважно визначення функції Show(). Зверніть увагу на те, що в ній використовується тільки одне значення змінної x, яка повинна містити радіус круга[1]. Проте, згідно з визначенням функції Set(), у базовому класі figure їй передається два значення, а не одне. Оскільки похідному класу circle не потрібне друге значення, то що ми можемо зробити? Є два способи вирішити цю проблему. Перший (і одночасно найгірший) полягає у тому, що ми могли б, працюючи з об'єктом класу circle, просто викликати функцію Set(), передаючи їй як другий параметр фіктивне значення. Основний недолік цього методу – відсутність чіткості у задаванні параметрів і потрібно пам'ятати про спеціальні винятки, які порушують дію механізму – один інтерфейс, багато методів.

Є вдаліший спосіб вирішення цього питання, який полягає у наданні параметру функції Set() значення, яке діє за замовчуванням. У цьому випадку під час виклику функції Set() для круга потрібно задавати тільки радіус. Під час виклику ж функції Set() для трикутника або прямокутника задають обидва значення. Нижче показано програму, у якій реалізовано цей підхід.

Код програми 6.6. Демонстрація механізму надання параметру віртуальної функції значення, що діє за замовчуванням

```
#include <vcl>
#include <conio>
#include <iostream>          // Для потокового введення-виведення
using namespace std;         // Використання стандартного простору імен
```

[1] Пригадайте, що площа круга обчислюється за формулою πR^2.

```
class figure {
protected:
        double x, y;
public:
        void Set(double _x, double _y=0) { x = _x; y = _y; }
        virtual void Show()
        {
                cout << Cyr("У цьому класі вираз для обчислення площі не визначено.") << endl;
        }
};

class triangle : public figure {
public:
        void Show()
        {
                cout << Cyr("Трикутник з висотою ") << x; cout << Cyr(" і основою ") << y;
                cout << Cyr(" має площу ") << x * 0.5 * y ; cout << Cyr(" кв. од.") << endl;
        }
};

class rectangle : public figure {
public:
        void Show()
        {
                cout << Cyr("Прямокутник розмірами ") << x << " x " << y;
                cout << Cyr(" має площу ") << x * y ; cout << Cyr(" кв. од.") << endl;
        }
};

class circle : public figure {
public:
        void Show()
        {
                cout << Cyr("Круг з радіусом ") << x;
                cout << Cyr(" має площу ") << 3.14159 * x * x ; cout << Cyr(" кв. од.") << endl;
        }
};

int main()
{
        figure *p;              // Створення покажчика на об'єкт базового типу
        triangle ObjT;          // Створення об'єкта похідного типу
        rectangle ObjR;         // Створення об'єкта похідного типу
        circle ObjC;            // Створення об'єкта похідного типу

        p = &ObjT;              // Присвоєння покажчику адреси об'єкта похідного класу
        p->Set(10.3, 5.5);
        p->Show();

        p = &ObjR;              // Присвоєння покажчику адреси об'єкта похідного класу
```

```
        p->Set(10.3, 5.5);
        p->Show();

        p = &ObjC;              // Присвоєння покажчику адреси об'єкта похідного класу
        p->Set(9.67);
        p->Show();

        getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

Трикутник з висотою 10.3 і основою 5.5 має площу 28.325 кв. од.
Прямокутник розмірами 10.3 x 5.5 має площу 56.65 кв. од.
Круг з радіусом 9.67 має площу 293.767 кв. од.

Вартоа' нати!а*Хоча віртуальні функції синтаксично прості для розуміння, їх справжні можливості неможливо продемонструвати на навчальних прикладах. Як правило, потужність поліморфізму виявляється у великих складних ієрархічних системах. У міру засвоєння мови програмування С++ Вам ще не раз буде надано можливість переконатися в їх користі.*

6.2.5. Поняття про суто віртуальні функції та абстрактні класи

Як було зазначено вище, якщо віртуальна функція, яка не перевизначена у похідному класі, викликається об'єктом цього похідного класу, то використовується та її версія, яку було визначено в базовому класі. Але у багатьох випадках взагалі немає сенсу давати визначення віртуальної функції в базовому класі. Наприклад, в базовому класі figure (з попереднього прикладу) визначення функції Show() – це просто заглушка. Вона не обчислює і не відображає площу жодного з об'єктів. Як буде показано згодом, при створенні власних бібліотек класів, у тому, що віртуальна функція не має конкретного визначення у базовому класі, немає нічого незвичайного.

Існує два способи оброблення таких ситуацій. Перший (він показаний у наведеному вище прикладі коду програми) полягає у виведенні функцією застережного повідомлення. Можливо, такий підхід і буде корисний у певних ситуаціях, але здебільшого він просто неприйнятний. Наприклад, можна уявити собі віртуальні функції, без визначення яких у існуванні похідного класу взагалі немає ніякого сенсу. Розглянемо клас triangle. Він абсолютно зайвий, якщо у ньому не визначити функцію Show(). У цьому випадку варто створити метод, який би гарантував, що похідний клас дійсно містить усі необхідні функції. У мові програмування С++ для вирішення цього питання і передбачено суто віртуальні функції.

Суто віртуальна функція – це віртуальна функція, яку оголошено в базовому класі, але вона не має у ньому ніякого визначення. Тому будь-який похідний тип повинен визначити власну версію цієї функції, адже у нього просто немає ніякої можливості використовувати версію з базового класу (через її відсутність). Щоб оголосити суто віртуальну функцію, використовують такий загальний формат:

virtual *тип ім'я_функції* (*перелік_параметрів*) = 0;

У цьому записі під елементом *тип* маємо на увазі тип значення, що повертається функцією, а елемент *ім'я_функції* – використовуване у програмі її ім'я. Позначення = 0 є ознакою того, що функція тут оголошується як *суто віртуальна*. Наприклад, в наступній версії визначення базового класу figure функція Show() вже представлена як суто віртуальна:

```
class figure {
        double x, y;
public:
        void Set(double _x, double _y=0) { x = _x; y = _y; }
        virtual void Show() = 0;  // Суто віртуальна функція
};
```

Оголосивши функцію суто віртуальною, програміст створює умови, при яких похідний клас просто вимушений мати визначення власної її реалізації. Без цього компілятор видасть повідомлення про помилку. Наприклад, спробуйте скомпілювати цю модифіковану версію програми обчислення площ геометричних фігур, у якій з класу circle видалено визначення функції Show().

Код програми 6.7. Демонстрація не коректної програми, яка у класі circle немає перевизначення функції Show()

```
#include <vcl>
#include <conio>
#include <iostream>              // Для потокового введення-виведення
using namespace std;             // Використання стандартного простору імен

class figure {
protected:
        double x, y;
public:
        void Set(double _x, double _y) { x = _x; y = _y; }
        virtual void Show() = 0;         // Суто віртуальна функція
};

class triangle : public figure {
public:
        void Show()
        {
                cout << Cyr("Трикутник з висотою ") << x; cout << Cyr(" і основою ") << y;
                cout << Cyr(" має площу ") << x * 0.5 * y ; cout << Cyr(" кв. од.") << endl;
        }
};

class rectangle : public figure {
public:
        void Show()
        {
                cout << Cyr("Прямокутник розмірами ") << x << " x " << y;
                cout << Cyr(" має площу ") << x * y ; cout << Cyr(" кв. од.") << endl;
        }
};
```

```
class circle : public figure {
        // Відсутність визначення функції Show()
        // Викличе повідомлення про помилку.
};

int main()
{
        figure *p;                  // Створення покажчика на об'єкт базового типу
        triangle ObjT;              // Створення об'єкта похідного типу
        rectangle ObjR;             // Створення об'єкта похідного типу
        circle ObjC;                // Помилка: створення цього об'єкта є неможливим!
        p = &ObjT;                  // Присвоєння покажчику адреси об'єкта похідного класу
        p->Set(10.3, 5.5);
        p->Show();
        p = &ObjR;                  // Присвоєння покажчику адреси об'єкта похідного класу
        p->Set(10.3, 5.5);
        p->Show();
        getch(); return 0;
}
```

*Якщо клас містить хоч би одну суто віртуальну функцію, то він називається **а! страктним**.*

Абстрактний клас характеризує одна важлива особливість: у такого класу не може бути об'єктів. Абстрактний клас можна використовувати тільки як базовий, з якого виводитимуться інші похідні класи. Причина того, що абстрактний клас не можна використовувати для побудови об'єктів, полягає, безумовно, у тому, що його одна або декілька функцій не мають визначення. Але навіть якщо базовий клас є абстрактним, то його все одно можна використовувати для оголошення покажчиків і посилань, які необхідні для підтримки динамічного поліморфізму.

6.2.6. Порівняння механізму раннього зв'язування з пізнім

Під час обговорення об'єктно-орієнтованих мов програмування зазвичай використовують два терміни: *раннє зв'язування* (early binding) і пізнє зв'язування (late binding). У мові програмування С++ ці терміни пов'язують з подіями, які відбуваються при компілюванні та у період виконання програми відповідно.

При ранньому зв'язуванні виклик функції готується при компілюванні програми, а при пізньому − у процесі виконання програми.

Раннє зв'язування означає, що вся інформація, необхідна для виклику функції, відома при компілюванні програми. Прикладами раннього зв'язування можуть слугувати виклики стандартних функцій і виклики перевизначених функцій (звичайних і операторних). З принципових переваг раннього зв'язування можна назвати ефективність: воно працює швидше від пізнього і часто вимагає менших витрат пам'яті. Його основний недолік – відсутність гнучкості.

Пізнє зв'язування означає, що точне рішення про виклик функції буде ухвалено у процесі виконання програми. Пізнє зв'язування у мові програмування С++

досягається за рахунок застосування віртуальних функцій і похідних типів. Перевага пізнього зв'язування полягає у тому, що воно забезпечує великий ступінь гнучкості. Його можна застосовувати для підтримки загального інтерфейсу і давати змогу при цьому різним об'єктам, які використовують цей інтерфейс, визначати їх власні реалізації. Понад це, пізнє зв'язування може допомогти програмісту у створенні бібліотек класів, що характеризуються багатократним використанням і можливістю розширюватися. Але до його недоліків можна віднести, хоч і незначне, але все таки пониження швидкості виконання програм.

Відповідь на запитання: чому віддати перевагу – ранньому або пізньому зв'язуванню, залежить від призначення Вашої програми[1]. Пізнє зв'язування (його ще називають *динамічним*) – це один з найпотужніших засобів мови програмування C++. Проте за цю потужність доводиться розплачуватися втратами у швидкості виконання програм. Тому пізнє зв'язування найкраще використовувати тільки у разі, коли воно істотно покращує структуру і керованість програмою. Як і всі практичні засоби, пізнє зв'язування, зазвичай, варто використовувати, але не зловживати ним. Викликані ним втрати у продуктивності є дуже незначні, тому, коли ситуація вимагає пізнього зв'язування, сміливо беріть його на озброєння.

6.2.7. Поняття про поліморфізм і пуризм

Впродовж всього навчального посібника (і зокрема, у цьому розділі) ми відзначаємо відмінності між динамічним і статичним поліморфізмом. *Статичний поліморфізм* (поліморфізм часу компілювання) реалізується у перевизначенні функцій і операторів. *Динамічний поліморфізм* (поліморфізм тривалості виконання програми) досягається за рахунок віртуальних функцій. Найзагальніше визначення поліморфізму поміщене у фразі "один інтерфейс, багато методів", і всі згадані вище "знаряддя" поліморфізму відповідають цьому визначенню. Проте під час використання самого терміну *поліморфізм* все ж таки існують деякі розбіжності.

Деякі пуристи (у цьому випадку – борці за чистоту термінології) об'єктно-орієнтованого програмування наполягають на тому, щоб цей термін використовувався тільки для подій, які відбуваються у процесі виконання програм. Вони стверджують, що поліморфізм підтримується тільки віртуальними функціями. Частково ця точка зору ґрунтується на тому факті, що найпершими поліморфічними мовами програмування були інтерпретатори (для них характерним є те, що всі події належать тривалості виконання програми). Поява трансльованих поліморфічних мов програмування розширила концепцію реалізації поліморфізму. Однак все ще не утихають заяви про те, що термін *поліморфізм* повинен застосовуватися виключно до подій періоду виконання програми. Більшість C++-програмістів не погоджуються з цією точкою зору і вважають, що цей термін можна застосовувати до обох видів засобів. Тому Ви не повинні дивуватися, якщо хтось раптом стане сперечатися з Вами на предмет використання цього терміна!

---

[1] Насправді в більшості крупних програм використовуються обидва види зв'язування.

Розділ 7. РОБОТА З ШАБЛОННИМИ ФУНКЦІЯМИ ТА КЛАСАМИ

Шаблон – це один з складних і потужних засобів мови програмування С++. Він не увійшов до початкової специфікації мови С++, і тільки в кінці 90-х років став невід'ємною частиною програмування нею. Шаблони дають змогу виконати одне з найважчих завдань у програмуванні – створювати програмний код, який можна використовувати для оброблення різних типів даних.

Використовуючи шаблони, можна створювати *узагальнені функції* та *узагальнені класи*. В узагальненій функції (або класі) оброблюваний нею (ним) тип даних задається як параметр. Таким чином, одну і ту саму функцію або клас можна використовувати для роботи з різними типами даних, не вказуючи безпосередньо конкретні її (його) версії для оброблення кожного з типів. Детальний аналіз механізму реалізації узагальнених функцій і класів якраз і проведено у цьому розділі.

7.1. Поняття про узагальнені функції

Узагальнена функція визначає загальний набір операцій, які згодом використовуватимуться для оброблення даних різних типів. Тип даних, який обробляється функцією, передається їй як параметр. Використовуючи узагальнену функцію для оброблення широкого діапазону даних, можна застосувати єдину загальну процедуру. На сьогодні відомо багато алгоритмів, які мають однакову логіку оброблення різних типів даних. Наприклад, один і той самий алгоритм сортування Quicksort застосовується і для впорядкування елементів масиву цілих чисел, і до масиву чисел з плинною крапкою. Відмінність тут полягає тільки в типі сортованих даних. Створюючи узагальнену функцію, можна запрограмувати роботу алгоритму незалежно від типу оброблюваних даних. Після цього компілятор автоматично згенерує коректний програмний код для типу даних, який насправді встановлюється у процесі виконання цієї функції. Загалом, створюючи узагальнену функцію, створюється функція, яка автоматично перевизначає себе саму.

Узагальнена функція – це функція, яка перевизначає сама себе.

Узагальнена функція створюється за допомогою ключового слова **template**. Звичайне значення слова "**template**" точно відображає мету його застосування у мові програмування С++. Це ключове слово використовують для створення шаблону (або оболонки), який описує дії, виконувані функцією. Компіляторові ж залишається "доповнити відсутні деталі" відповідно до заданого значення параметра. Загальний формат визначення шаблонної функції має такий вигляд:

```
template <class tType> тип ім'я_функції (перелік_параметрів)
{
    // тіло функції
}
```

У цьому записі елемент *tType* є "заповнювачем" для типу даних, які обробляються функцією. Це ім'я використовується в тілі самої функції. Але воно означає всього тільки заповнювач, замість якого компілятор автоматично підставить реальний тип даних при створенні конкретної версії функції. І хоча для задавання узагальненого типу в **template**-оголошенні за традицією застосовується ключове слово **class**, однак можна також використовувати ключове слово **typename**.

7.1.1. Механізм реалізації шаблонної функції з одним узагальненим типом

У наведеному нижче прикладі створюється шаблонна функція з одним узагальненим типом, яка міняє місцями значення двох змінних, що використовується під час її виклику. Оскільки загальний процес обміну значеннями змінних не залежить від їх типу, він є типовим претендентом для створення узагальненої функції.

Код програми 7.1. Демонстрація механізму застосування шаблонної функції з одним узагальненим типом

```
#include <vcl>
#include <iostream>            // Для потокового введення-виведення
#include <conio>               // Для консольного режиму роботи
using namespace std;           // Використання стандартного простору імен

// Визначення шаблонної функції.
template <class aType> void swapAB(aType &a, aType &b)
{
	aType tmp;          // Створення тимчасової змінної
	tmp = a;
	a = b;
	b = tmp;
}

int main()
{
	int i = 10, j = 20;
	double x = 10.1, y = 23.3;
	char a = 'x', b = 'z';

	cout << "Початкові значення i, j: " << i << " " << j << endl;
	cout << "Початкові значення x, y: " << x << " " << y << endl;
	cout << "Початкові значення a, b: " << a << " " << b << endl;

	swapAB(i, j);       // Перестановка цілих чисел
	swapAB(x, y);       // Перестановка чисел з плинною крапкою
	swapAB(a, b);       // Перестановка символів

	cout << "Після перестановки i, j: " << i << " " << j << endl;
	cout << "Після перестановки x, y: " << x << " " << y << endl;
	cout << "Після перестановки a, b: " << a << " " << b << endl;
	getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Початкові значення i, j: 10 20
Початкові значення x, y: 10.1 23.3
Початкові значення a, b: x z
Після перестановки i, j: 20 10
Після перестановки x, y: 23.3 10.1
Після перестановки a, b: z x
```

Отже, розглянемо уважно код програми. Рядок

```
template <class aType> void swapAB(aType &a, aType &b)
```

повідомляє компілятор, по-перше, що створюється шаблон, і, по-друге, що тут починається узагальнене визначення шаблонної функції. Позначення aType є узагальненим типом, який використовується як "заповнювач". За **template**-заголовком знаходиться оголошення функції swapAB(), у якому символ aType означає тип даних для значень, які мінятимуться місцями. У функції **main**() продемонстровано виклик функції swapAB() з використанням трьох різних типів даних: **int**, **double** і **char**. Оскільки функція swapAB() є узагальненою, то компілятор автоматично створює три версії функції swapAB(): одну для обміну цілих чисел, другу для обміну чисел з плинною крапкою і третю для обміну символів.

Тут необхідно уточнити деякі важливі терміни, пов'язані з шаблонами. По-перше, узагальнена функція (тобто функція, оголошення якої передує **template**-настанові) також називається *шаблонною функцією*. Обидва терміни використовуються у цьому навчальному посібнику як взаємозамінні. Коли компілятор створює конкретну версію цієї функції, то вважають, що створюється її *спеціалізація* (або *конкретизація*). Спеціалізація також називається *породженою функцією* (generated function). Дію породження функції визначають як її *реалізацію* (instantiating). Іншими словами, породжувана функція є конкретним примірником шаблонної функції.

Оскільки мова програмування С++ не розпізнає символ кінця рядка як ознаку кінця настанови, то **template**-частина визначення узагальненої функції може не знаходитися в одному рядку з іменем цієї функції. У наведеному нижче прикладі показано ще один (достатньо поширений) спосіб форматування функції swapAB():

```
template <class aType>
void swapAB(aType &a, aType &b)
{
	aType tmp;	// Створення тимчасової змінної

	tmp = a;
	a = b;
	b = tmp;
}
```

При використанні цього формату важливо розуміти, що між **template**-настановою і початком визначення узагальненої функції ніякі інші настанови знаходитися не можуть. Наприклад, наведений нижче код програми не відкомпілюється:

```
// Цей програмний код не відкомпілюється
template <class aType>
	int i; // Тут помилка!
```

```
void swapAB(aType &a, aType &b)
{
        aType tmp;          // Створення тимчасової змінної

        tmp = a;
        a = b;
        b = tmp;
}
```

Як зазначено в коментарі, **template**-специфікація повинна знаходитися безпосередньо перед визначенням функції. Між ними не може знаходитися ні настанова оголошення змінної, ні будь-яка інша настанова.

7.1.2. Безпосередньо задане перевизначення узагальненої функції

Незважаючи на те, що узагальнена функція сама перевизначається в міру потреби, однак це можна робити і безпосередньо. Формально цей процес називається *безпосередньою спеціалізацією.* При перевизначенні узагальнена функція перевизначається "на вигоду" цієї конкретної версії. Розглянемо, наприклад, наведену нижче програму, яка є переробленою версією першого прикладу з цього розділу.

"Вручну" перевизначена версія узагальненої функції називається ***! е' пон середньоюаспеціалі' ацією****.*

Код програми 7.2. Демонстрація механізму перевизначення шаблонної функції

```
#include <vcl>
#include <iostream>    // Для потокового введення-виведення
#include <conio>       // Для консольного режиму роботи
using namespace std; // Використання стандартного простору імен

template <class aType> void swapAB(aType &a, aType &b)
{
        aType tmp;          // Створення тимчасової змінної

        tmp = a;
        a = b;
        b = tmp;
        cout << "Виконується шаблонна функція swapAB" << endl;
}

// Ця функція перевизначає узагальнену версію функції swapAB() для int-параметрів.
void swapAB(int &a, int &b)
{
        int tmp;

        tmp = a;
        a = b;
        b = tmp;
        cout << "Це int-спеціалізація функції swapAB" << endl;
}
```

```
int main()
{
	int i = 10, j = 20;
	double x = 10.1, y = 23.3;
	char a = 'x', b = 'z';

	cout << "Початкові значення i, j: " << i << " " << j << endl;
	cout << "Початкові значення x, y: " << x << " " << y << endl;
	cout << "Початкові значення a, b: " << a << " " << b << endl;
	swapAB(i, j);		// Викликається безпосередньо перевизначена функція swapAB().
	swapAB(x, y);		// Викликається узагальнена функція swapAB().
	swapAB(a, b);		// Викликається узагальнена функція swapAB().

	cout << "Після перестановки i, j: " << i << " " << j << endl;
	cout << "Після перестановки x, y: " << x << " " << y << endl;
	cout << "Після перестановки a, b: " << a << " " << b << endl;

	getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Початкові значення i, j: 10 20
Початкові значення x, y: 10.1 23.3
Початкові значення a, b: x z
Це int-спеціалізація функції swapAB.
Виконується шаблонна функція swapAB.
Виконується шаблонна функція swapAB.
Після перестановки i, j: 20 10
Після перестановки x, y: 23.3 10.1
Після перестановки a, b: z x
```

Як зазначено в коментарях до цієї програми, під час виклику функції swapAB(i, j) виконується безпосередньо перевизначена версія функції swapAB(), яку визначено у програмі. Компілятор у цьому випадку не генерує узагальнену функцію swapAB(), оскільки вона перевизначається безпосередньо заданим варіантом перевизначеної функції. Для позначення безпосередньої спеціалізації функції можна використовувати новий альтернативний синтаксис, що містить ключове слово **template**. Наприклад, якщо задати спеціалізацію з використанням цього альтернативного синтаксису, то перевизначена версія функції swapAB() з попереднього коду програми виглядатиме так:

```
// Використання нового синтаксису задавання спеціалізації
template<> void swapAB<int>(int &a, int &b)
{
	int tmp;

	tmp = a;
	a = b;
	b = tmp;
	cout << "Це int-спеціалізація функції swapAB" << endl;
}
```

Як бачите, в новому синтаксисі для позначення спеціалізації використовують конструкцію **template<>**. Тип даних, для яких створюється ця спеціалізація, вказується в кутових дужках після імені функції. Для задавання будь-якого типу узагальненої функції використовується один і той самий синтаксис. На даний момент жоден з синтаксичних способів задавання спеціалізації не має жодних переваг перед іншим, але з погляду перспективи розвитку мови програмування, можливо, все ж таки краще використовувати новий стиль.

Безпосередня спеціалізація шаблону дає змогу спроектувати версію узагальненої функції з розрахунку на деяку унікальну ситуацію, щоб, можливо, скористатися перевагами підвищеної швидкодії програми тільки для одного типу даних. Але, як правило, якщо виникає потреба мати різні версії функції для різних типів даних, то доцільно використовувати перевизначені функції, а не шаблони.

7.1.3. Шаблонна функція з двома узагальненими типами

У **template**-настанові можна визначити декілька узагальнених типів даних, використовуючи перелік елементів, розділених між собою комами. Наприклад, у наведеному нижче коді програми створюється шаблонна функція з двома узагальненими типами.

Код програми 7.3. Демонстрація механізму застосування шаблонної функції з двома узагальненими типами

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен

template <class aType, class bType> void FunC(aType a, bType b)
{
	cout << a << " " << b << endl;
}

int main()
{
	FunC(10, "Привіт");

	FunC(0.23, 10L);

	getch(); return 0;
}
```

У наведеному прикладі у процесі виконання функції **main**(), коли компілятор генерує конкретні примірники функції FunC(), заповнювачі типів aType і bType заміняються спочатку парою типів даних **int** і **char** *, а потім парою **double** і **long** відповідно.

Нео! хідноа пам'ятати! *Створюючи шаблонну функцію, програміст дає змогу компіляторові генерувати стільки її різних версій, скільки необхідно для оброблення різних типів даних, які використовує програма для її виклику.*

7.1.4. Механізм перевизначення специфікації шаблону функції

Окрім створення безпосередньо перевизначених версій узагальненої функції, можна також перевизначати саму специфікацію шаблону функції. Для цього достатньо створити ще одну версію шаблону, яка відрізнятиметься від інших переліком параметрів. Розглянемо такий приклад.

Код програми 7.3. Демонстрація механізму перевизначення специфікації шаблону функції

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен

// Перша версія шаблону Fun().
template <class aType> void Fun(aType a)
{
    cout << "Виконується функція Fun(aType a)" << endl;
}

// Друга версія шаблону Fun().
template <class aType, class bType> void Fun(aType a, bType b)
{
    cout << "Виконується функція Fun(aType a, bType b)" << endl;
}

int main()
{
    Fun(10);          // Викликається функція Fun(a).
    Fun(10, 20);      // Викликається функція Fun(a, b).

    getch(); return 0;
}
```

У цьому коді програми шаблон для функції Fun() перевизначається, щоб забезпечити можливість прийняття як одного, так і двох параметрів.

7.1.5. Використання стандартних параметрів у шаблонних функціях

У шаблонних функціях можна змішувати стандартні параметри з узагальненими параметрами типу. Ці параметри працюють так само, як і у будь-якій іншій функції. Розглянемо такий приклад.

Код програми 7.4. Демонстрація механізму використання стандартних параметрів у шаблонній функції

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен

// Відображення даних задану кількість разів.
```

```
template<class aType> void repeat(aType data, int times)
{
	int tim = times;
	do {
		cout << tim – times + 1 << " ==> " << data << endl;
		times--;
	} while(times);
	cout << endl;
}

int main()
{
	repeat("Це тест.", 3);
	repeat(100, 5);
	repeat(99.0/2, 4);

	getch(); return 0;
}
```

Ось які результати генерує ця програма.

```
1 ==> Це тест.
2 ==> Це тест.
3 ==> Це тест.

1 ==> 100
2 ==> 100
3 ==> 100
4 ==> 100
5 ==> 100

1 ==> 49.5
2 ==> 49.5
3 ==> 49.5
4 ==> 49.5
```

У цьому коді програми функція **repeat**() відображає свій перший аргумент стільки раз, скільки задано її другим аргументом. Оскільки перший аргумент має узагальнений тип, то функцію **repeat**() можна використовувати для відображення даних будь-якого типу. Параметр times – стандартний, він передається за значенням. Змішане задавання узагальнених і стандартних параметрів, як правило, не викликає жодних проблем їх реалізації, застосовується найчастіше у програмуванні.

7.1.6. Обмеження, які застосовуються при використанні узагальнених функцій

Узагальнені функції подібні до перевизначених функцій, але мають більше обмежень з їх застосування. При перевизначенні функцій в тілі кожної з них зазвичай записують різні дії. Але узагальнена функція повинна виконувати одні і ті самі дії для всіх її версій; відмінність між версіями полягає тільки в типі оброблюваних даних. Розглянемо приклад, у якому перевизначені функції не можна замінити узагальненою функцією, оскільки вони виконують різні дії:

```
void outdata(int i)
{
	cout << i << endl;
}

void outdata(double d)
{
	cout << d * 3.1416 << endl;
}
```

7.1.7. Приклад створення узагальненої функції abs()

Давайте знову звернемося до функції **abs**(). Пригадайте, як у розд. 8 стандартні бібліотечні функції **abs**(), **labs**() і **fabs**() були згруповані в три перевизначені функції із загальним іменем myAbs(). Кожна з перевизначених версій функції myAbs() призначена для повернення абсолютного значення для даних "свого" типу. Незважаючи на те, що показане в розд. 8 **[9]** перевизначення функції **abs**() можна вважати кроком уперед порівняно з використанням трьох різних бібліотечних функцій (з різними іменами), все ж таки це не кращий механізм реалізації функції, яка повертає абсолютне значення заданого аргументу. Оскільки процедура повернення абсолютного значення числа однакова для всіх типів числових значень, то функція **abs**() може слугувати типовим прикладом для створення шаблонної функції. За наявності узагальненої версії функції **abs**() компілятор зможе автоматично створювати необхідну її версію. Програміст у цьому випадку звільняється від написання окремих версій для кожного типу даних[1].

У наведеному нижче коді програми міститься узагальнена версія функції myAbs(). Є сенс порівняти її з перевизначеними версіями, наведеними у розд. 8 **[9]**. Неважко переконатися в тому, що узагальнена версія цієї функції є набагато коротшою і володіє більшою гнучкістю.

Код програми 7.5. Демонстрація механізму створення узагальненої версії функції, яка повертає абсолютне значення числа

```
#include <vcl>
#include <iostream>           // Для потокового введення-виведення
#include <conio>              // Для консольного режиму роботи
using namespace std;          // Використання стандартного простору імен

template <class aType> aType myAbs(aType n)
{
	return n < 0 ? -n : n;
}

int main()
{
	cout << myAbs(-10) << endl;             // Для типу int
	cout << myAbs(-10.0) << endl;           // Для типу double
```

[1] Окрім цього, початковий код програми не захаращується декількома "вручну" перевизначеними версіями.

```
        cout << myAbs(-10L) << endl;          // Для типу long
        cout << myAbs(-10.0F) << endl;        // Для типу float

        getch(); return 0;
}
```

Для вправи було б непогано, якби Ви спробували знайти інші бібліотечні функції-претенденти для перероблення їх в узагальнені функції. Необхідно пам'ятати – головне тут те, що один і той самий алгоритм повинен застосовуватися до широкого діапазону типу даних.

7.2. Поняття про узагальнені класи

Окрім визначення узагальнених функцій, можна також визначити узагальнений клас. Для цього створюється клас, у якому визначаються всі використовувані ним алгоритми. При цьому реальний тип оброблюваних у ньому даних задається як параметр при побудові об'єктів цього класу.

Узагальнені класи особливо є корисними тоді, коли в них використовується логіка, яку можна узагальнити. Наприклад, алгоритми, які підтримують функціонування черги цілочисельних значень, також надаються і для підтримки символьних значень. Механізм, який забезпечує підтримку зв'язного списку поштових адрес, також годиться для підтримки зв'язного списку, призначеного для зберігання даних про запчастини до автомобілів. Після свого створення узагальнений клас виконує визначену програмістом операцію (наприклад, підтримку черги або зв'язного списку) для будь-якого типу даних. Компілятор автоматично згенерує коректний тип об'єкта на основі типу, який задається під час його створення.

Загальний формат оголошення узагальненого класу має такий вигляд:

```
template <class tType> class ім'я_класу {
        // Тіло класу
};
```

У цьому записі елемент *tType* є "заповнювачем" для імені типу, який задається під час реалізації класу. У разі потреби можна визначити декілька узагальнених типів даних, використовуючи перелік елементів, розділених між собою комами.

Створивши узагальнений клас, можна створити його конкретний примірник, використовуючи такий загальний формат:

```
ім'я_класу <тип> ім'я_об'єкту;
```

У цьому записі елемент *тип* означає ім'я типу даних, які оброблятимуться примірником узагальненого класу. Функції-члени узагальненого класу автоматично є узагальненими. Тому програмісту не потрібно використовувати ключове слово **template** для безпосереднього визначення їх такими.

7.2.1. Створення класу з одним узагальненим типом даних

У наведеному нижче коді програми клас queueClass перероблений в узагальнений клас з одним узагальненим типом даних. Це означає, що його можна використовувати для організації черги об'єктів будь-якого типу. У цьому прикладі ство-

рюються дві черги: для цілих чисел і значень з плинною крапкою, але можна використовувати дані і будь-якого іншого типу.

Код програми 7.6. Демонстрація механізму створення класу з одним узагальненим типом даних

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен

const int size=100;

// Створення узагальненого класу queueClass.
template <class qType> class queueClass {
    qType array[size];
    int sloc, rloc;
public:
    queueClass() { sloc = rloc = 0; }
    void Get(qType c);
    qType Put();        // Виведення з об'єкта значення
};

// Занесення об'єкта в чергу.
template <class qType> void queueClass<qType>::Get(qType c)
{
    if(sloc==size) { cout << "Черга заповнена" << endl; return; }
    sloc++;
    array[sloc] = c;
}

// Вилучення об'єкта з черги.
template <class qType> qType queueClass<qType>::Put()
{
    if(rloc == sloc) { cout << "Черга порожня" << endl; return 0; }
    rloc++;
    return array[rloc];
}

int main()
{
    queueClass<int> ObjA, ObjB;    // Створюємо дві черги для int-значень.
    ObjA.Get(10);
    ObjB.Get(19);
    ObjA.Get(20);
    ObjB.Get(1);

    cout << ObjA.Put() << " ";
    cout << ObjA.Put() << " ";
    cout << ObjB.Put() << " ";
    cout << ObjB.Put() << endl;
```

```
        queueClass<double> ObjC, ObjD;      // Створюємо дві черги для double-значень
        ObjC.Get(10.12);
        ObjD.Get(19.99);
        ObjC.Get(-20.0);
        ObjD.Get(0.986);

        cout << ObjC.Put() << " ";
        cout << ObjC.Put() << " ";
        cout << ObjD.Put() << " ";
        cout << ObjD.Put() << endl;

        getch(); return 0;
    }
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
10 20 19 1
10.12 -20 19.99 0.986
```

У цьому коді програми оголошення узагальненого класу є подібним до оголошення узагальненої функції. Тип даних, який зберігаються в черзі, узагальнений в оголошенні класу. Він невідомий доти, доки не буде оголошений об'єкт класу queueClass, який і визначить реальний тип даних. Після оголошення конкретного примірника класу queueClass компілятор автоматично згенерує всі функції та змінні, необхідні для оброблення реальних даних. У цьому прикладі оголошуються два різних типи черги: дві черги для зберігання цілих чисел і дві черги для значень типу **double**. Зверніть особливу увагу на ці оголошення:

```
queueClass<int> ObjA, ObjB;
queueClass<double> ObjC, ObjD;
```

Зауважте, як вказується потрібний тип даних: він поміщається в кутові дужки. Змінюючи тип даних при створенні об'єктів класу queueClass, можна змінити тип даних, який зберігаються в черзі. Наприклад, використовуючи таке оголошення, можна створити ще одну чергу, яка міститиме покажчики на символи:

```
queueClass<char *> chrptrQ;
```

Можна також створювати черги для зберігання даних, тип яких створений програмістом. Наприклад, нехай необхідно використати таку структуру для зберігання інформації про адресу:

```
struct addrStruct {         // Оголошення типу структури
    char name[40];
    char street[40];
    char city[30];
    char state[3];
    char zip[12];
};
```

Тепер для того, щоби за допомогою узагальненого класу queueClass, можна було згенерувати чергу для зберігання об'єктів типу addrStruct, достатньо використовувати таке оголошення:

```
queueClass<addrStruct> obj;
```

На прикладі класу queueClass неважко переконатися, що узагальнені функції та класи є потужними засобами, які допоможуть збільшити ефективність роботи програміста, оскільки вони дають змогу визначити загальний формат об'єкта, який можна потім використовувати з будь-яким типом даних. Узагальнені функції та класи позбавляють Вас від рутинної праці зі створення окремих реалізацій для кожного типу даних, які підлягають обробленню єдиним алгоритмом. Цю роботу зробить за Вас компілятор: він автоматично створить конкретні версії визначеного Вами класу.

7.2.2. Створення класу з двома узагальненими типами даних

Шаблонний клас може мати декілька узагальнених типів даних. Для цього достатньо оголосити всі потрібні типи даних в **template**-специфікації у вигляді елементів списку, що розділяються між собою комами. Наприклад, у наведеному нижче коді програми створюється клас, який використовує два узагальнених типи даних.

Код програми 7.7. Демонстрація механізму створення класу з двома узагальненими типами даних

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен

template <class aType, class bType> class myClass {
	aType c;
	bType d;
public:
	myClass(aType ObjA, bType ObjB) { c = ObjA; d = ObjB; }
	void showType()
		{cout << "c= " << c << "; d= " << d << endl; }
};

int main()
{
	myClass<int, double> ObjA(10, 0.23);
	myClass<char, char *> ObjB('x', "Це тест.");

	ObjA.showType();         // Відображення int- і double-значень
	ObjB.showType ();        // Відображення значень типу char і char *

	getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
c= 10; d= 0.23
c= x; d= Це тест.
```

У цьому коді програми оголошується два види об'єктів. Об'єкт ObjA використовує дані типу **int** і **double**, а об'єкт ObjB – символ і покажчик на символ. Для цих

ситуацій компілятор автоматично генерує дані та функції відповідного способу побудови об'єктів.

7.2.3. Приклад створення узагальненого класу для організації безпечного масиву

Розглянемо ще одне застосування узагальненого класу. Як було показано в розд. 4.4, можна перевизначати оператор "[]", що дає змогу створювати власні реалізації масивів, у тому числі і "безпечні масиви", які забезпечують динамічну перевірку порушення його меж. Як уже зазначалося вище, у процесі виконання програми, написаній мовою програмування C++, можливий вихід за межі масиву без видачі повідомлення про помилку. Але, якщо створити клас, який би містив масив, і дати змогу доступ до цього масиву тільки через перевизначений оператор індексації ("[]"), то можна перехопити індекс, що відповідає адресі за межами адресного простору масиву.

Об'єднавши перевизначення оператора з узагальненим класом, можна створити узагальнений тип безпечного масиву, який потім буде використано для створення безпечних масивів, призначених для зберігання даних будь-якого типу. Такий тип масиву і створюється в наведеному нижче коді програми.

Код програми 7.8. Демонстрація механізму створення та використання узагальненого класу для організації безпечного масиву

```
#include <vcl>
#include <iostream>            // Для потокового введення-виведення
#include <conio>               // Для консольного режиму роботи
using namespace std;           // Використання стандартного простору імен

const int size = 10;

template <class aType> class arrClass {        // Оголошення класового типу
        aType aMas[size];
public:
        arrClass() { for(int i=0; i<size; i++) aMas[i] = i; }
        arrClass &operator[](int i);
};

// Забезпечення контролю меж для класу aType.
template <class aType> aType &arrClass<aType>::operator[](int i)
{
        if(i<0 || i> size-1) {
                cout << "Значення індексу " << i << " за межами масиву" << endl;
                exit(1);
        }
        return aMas[i];
}

int main()
{
        arrClass<int> ObjI;                // Масив int-значень
```

```
        arrClass<double> ObjD;          // Масив double-значень

        cout << "Масив int-значень: ";
        for(int i=0; i<size; i++) ObjI[i] = i;
        for(int i=0; i<size; i++) cout << ObjI[i] << " ";
        cout << endl;

        cout << "Масив double-значень: ";
        for(int i=0; i<size; i++) ObjD[i] = i/3.0;
        for(int i=0; i<size; i++) cout << ObjD[i] << " ";
        cout << endl;

        ObjI[12] = 100;             // Помилка, спроба вийти за межі масиву!

        getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Масив int-значень: 0 1 2 3 4 5 6 7 8 9
Масив double-значень: 0 0.333333 0.666667 1 1.333333 1.666667 2 2.333333 2.666667 3
```

У цьому коді програми спочатку створюється узагальнений тип безпечного масиву, а потім продемонстровано його використання шляхом побудови масиву цілих чисел і масиву **double**-значень. Було б непогано, якби Ви спробували створити масиви інших типів. Як доводить цей приклад, одна з великих переваг узагальнених класів полягає у тому, що вони дають змогу тільки один раз написати програмний код, відлагодити його, а потім застосовувати його до даних будь-якого типу, не переписуючи його для кожного конкретного застосування.

7.2.4. Використання в узагальнених класах аргументів, що не є узагальненими типами

У **template**-специфікації для узагальненого класу можна також задавати аргументи, що не є узагальненими типами. Це означає, що в шаблонній специфікації можна вказувати те, що зазвичай приймається як стандартний аргумент, наприклад, аргумент типу **int** або аргумента-покажчика. Синтаксис (він практично такий самий, як під час задавання звичайних параметрів функції) містить визначення типу і імені аргумента. Ось, наприклад, як можна по-іншому реалізувати клас безпечного масиву, представленого в попередньому розділі.

Код програми 7.9. Демонстрація механізму використання в узагальнених класах аргументів, що не є узагальненими типами

```
#include <vcl>
#include <iostream>             // Для потокового введення-виведення
#include <conio>                // Для консольного режиму роботи
using namespace std;            // Використання стандартного простору імен

// Тут елемент int size – це аргумент, що не є узагальненими типом.
template <class aType, int size> class arrClass {
        aType aMas[size];       // В аргументі size передається розмір масиву.
```

```
public:
    arrClass() { for(int i=0; i<size; i++) aMas[i] = i; }
    arrClass &operator[](int i);
};

// Забезпечення контролю меж для класу aType.
template <class aType, int size> aType & arrClass<aType, size>::operator[](int i)
{
    if(i<0 || i> size-1) {
        cout << "Значення індексу " << i << " за межами масиву" << endl;
        exit(1);
    }
    return aMas[i];
}

int main()
{
    arrClass<int, 10> ObjI;          // 10-елементний масив цілих чисел
    arrClass<double, 15> ObjD;       // 15-елементний масив double-значень

    cout << "Масив цілих чисел: ";
    for(int i=0; i<10; i++) ObjI[i] = i;
    for(int i=0; i<10; i++) cout << ObjI[i] << " ";
    cout << endl;

    cout << "Масив double-значень: ";
    for(int i=0; i<15; i++) ObjD[i] = i/3.0;
    for(int i=0; i<15; i++) cout << ObjD[i] << " ";
    cout << endl;

    ObjI[12] = 100;     // Помилка тривалості виконання!

    getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Масив int-значень: 0 1 2 3 4 5 6 7 8 9
Масив double-значень: 0 0.333333 0.666667 1 1.333333 1.666667 2 2.333333 2.666667 3 3.333333
3.666667 4 4.333333 4.666667
```

Розглянемо уважно **template**-специфікацію для класу aType. Зверніть увагу на те, що аргумент size оголошений з вказанням типу **int**. Цей параметр потім використовують в тілі класу aType для оголошення розміру масиву aMas. Хоча у початковому коді програми член size має вигляд "змінної", його значення відоме вже при компілюванні. Тому його можна успішно використовувати для встановлення розміру масиву. Окрім цього, значення "змінної" size використовують для контролю виходу за межі масиву в операторній функції **operator**[](). Зверніть також увагу на те, як у функції **main**() створюється масив цілих чисел і масив значень з плинною крапкою. При цьому розмір кожного з них визначається другим параметром **template**-специфікації.

На тип параметрів, які не представляють типи, накладаються обмеження. У цьому випадку дозволено використовувати тільки цілочисельні типи, покажчики і посилання. Інші типи (наприклад, **float**) не допускаються. Аргументи, які передаються параметру, що не є узагальненими типом, повинні містити або цілочисельну константу, або покажчик, або посилання на глобальну функцію чи об'єкт. Таким чином, ці "нетипові" параметри необхідно розглядати як константи, оскільки їхні значення не можуть бути змінені. Наприклад, у тілі функції **operator**[]() така настанова недопустима:

```
size = 10;          // Помилка
```

Оскільки параметри-"не типи" обробляються як константи, то їх можна використовувати для встановлення розміру масиву, що істотно полегшує роботу програмісту.

Як показує наведений вище приклад створення безпечного масиву, використання "нетипових" параметрів дуже розширює сферу застосування шаблонних класів. І хоча інформація, яка передається через "нетиповий" аргумент, повинна бути відома при компілюванні, урахування цього обмеження непорівнянне з перевагами, пропонованими такими параметрами.

Шаблонний клас queueClass, представлено вище у цьому розділі, також виграв би від застосування до нього "нетипового" параметра, що задає розмір черги. Для домашньої вправи спробуйте удосконалити клас queueClass самостійно.

7.2.5. Використання в шаблонних класах аргументів за замовчуванням

Шаблонний клас може за замовчуванням визначати аргумент, який відповідає узагальненому типу. Наприклад, внаслідок оголошення такої **template**-специфікації

```
template <class aType=int> class myClass { // ...
```

використовуватиметься тип **int**, якщо при створенні об'єкта класу myClass відсутнє задавання будь-якого типу.

Для аргументів, які не представляють тип в **template**-специфікації, також дозволяється задавати значення за замовчуванням. Вони використовуються у випадку, якщо під час реалізації класу значення для такого аргументу безпосередньо не вказане. Аргументи за замовчуванням для "нетипових" параметрів задаються за допомогою синтаксису, аналогічного використовуваному під час задавання аргументів за замовчуванням для параметрів функцій.

Розглянемо ще одну версію класу безпечного масиву, у якому використовуються аргументи за замовчуванням як для типу даних, так і для розміру масиву.

Код програми 7.10. Демонстрація механізму використання шаблонних аргументів за замовчуванням

```
#include <vcl>
#include <iostream>           // Для потокового введення-виведення
#include <conio>              // Для консольного режиму роботи
using namespace std;          // Використання стандартного простору імен
```

```
// Тут параметр aType за замовчуванням приймає тип int, а параметр
// size за замовчуванням встановлюється таким, що дорівнює 10.
template <class aType=int, int size=10> class arrClass {
	aType aMas[size];		// Через параметр size передається розмір масиву.
public:
	arrClass() { for(int i=0; i<size; i++) aMas[i] = i; }
	arrClass &operator[](int i);
};

// Забезпечення контролю меж для класу aType.
template <class aType, int size> aType &arrClass<aType, size>::operator[](int i)
{
	if(i<0 || i> size-1) {
		cout << "Значення індексу " << i << " за межами масиву" << endl;
		exit(1);
	}
	return aMas[i];
}

int main()
{
	arrClass<int, 100> intMas;		// 100-елементний масив цілих чисел
	arrClass<double> doubleMas;	// 10-елементний масив double-значень
					// (розмір масиву встановлено за замовчуванням)
	arrClass<> defMas;		// 10-елементний масив int-значень
				// (розмір і тип int встановлені за замовчуванням)

	cout << "Масив цілих чисел: ";
	for(int i=0; i<100; i++) intMas[i] = i;
	for(int i=0; i<100; i++) cout << intMas[i] << " ";
	cout << endl;

	cout << "Масив double-значень: ";
	for(int i=0; i<10; i++) doubleMas[i] = i/3.0;
	for(int i=0; i<10; i++) cout << doubleMas[i] << " ";
	cout << endl;

	cout << "Масив за замовчуванням: ";
	for(int i=0; i<10; i++) defMas[i] = i;
	for(int i=0; i<10; i++) cout << defMas[i] << " ";
	cout << endl;

	getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Масив int-значень: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 ... 99
Масив double-значень: 0 0.333333 0.666667 1 1.333333 1.666667 2 2.333333 2.666667 3
Масив за замовчуванням: 0 1 2 3 4 5 6 7 8 9
```

Зверніть особливу увагу на цей рядок:

```
template <class aType=int, int size=10> class arrClass {
```

У цьому записі параметр aType за замовчуванням замінюється типом **int**, а параметр size за замовчуванням встановлюється таким, що дорівнює числу 10. Як показано у наведеному вище коді програми, об'єкти класу aType можна створити трьома способами:

- шляхом безпосереднього задавання як типу, так і розміру масиву;
- задавши безпосередньо тільки тип масиву, при цьому його розмір за замовчуванням встановлюється таким, що дорівнює 10 елементам;
- взагалі без задавання типу і розміру масиву, при цьому він за замовчуванням зберігатиме елементи типу **int**, а його розмір за замовчуванням встановлюється таким, що дорівнює 10.

Використання аргументів за замовчуванням (особливо типів) робить шаблонні класи ще гнучкішими. Тип, що використовується за замовчуванням, можна передбачити для найбільш вживаного типу даних, даючи змогу при цьому користувачу Ваших класів задавати потрібний тип даних.

7.2.6. Механізм реалізації безпосередньо заданої спеціалізації класів

Подібно до шаблонних функцій можна створювати і спеціалізації узагальнених класів. Для цього використовується конструкція **template<>**, яка працює за аналогією із безпосередньо заданими спеціалізаціями функцій. Розглянемо такий приклад.

Код програми 7.11. Демонстрація безпосередньо заданої спеціалізації класів

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен

template <class tType> class myClass {        // Оголошення класового типу
	tType x;
public:
	myClass(tType ObjA) { cout << "У тілі узагальненого класу myClass" << endl; x = ObjA; }
	tType getX() { return x; }
};

// Безпосередня спеціалізація для типу int.
template <> class myClass<int> {
	int x;
public:
	myClass(int a) { cout << "У тілі спеціалізації myClass<int>" << endl; x = a * a; }
	int getX() { return x; }
};

int main()
{
	myClass<double> ObjD(10.1);
	cout << "double: " << ObjD.getX() << endl << endl;
```

```
    myClass<int> ObjI(5);
    cout << "int: " << ObjI.getX() << endl;

    getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
У тілі узагальненого класу myClass.
double: 10.1

У тілі спеціалізації myClass<int>.
int: 25
```

У цьому коді програми зверніть особливу увагу на такий рядок:

```
template <> class myClass<int> {
```

Він повідомляє компіляторові про те, що створюється безпосередня **int**-спеціалізація даних класу myClass. Аналогічний синтаксис використовується і для будь-якого іншого типу спеціалізації класу.

Безпосередня спеціалізація класів розширює діапазон застосування узагальнених класів, оскільки вона дає змогу легко обробляти один або два спеціальні випадки, залишаючи всю решту варіантів для автоматичного оброблення компілятором. Але, якщо Ви побачите, що у Вас створюється дуже багато спеціалізацій, то тоді, можливо, краще взагалі відмовитися від створення шаблонного класу.

Розділ 8. МЕХАНІЗМИ ОБРОБЛЕННЯ ВИНЯТКОВИХ СИТУАЦІЙ

У цьому розділі розглядаються механізми оброблення різних виняткових ситуацій. *Виняткова ситуація* (або виняток) – це помилка, яка виникає у процесі виконання програми. Використовуючи С++-підсистему оброблення виняткових ситуацій, такі помилки легко можна виправляти. Їх виникнення під час роботи коду програми автоматично призводить до виклику так званого *обробника винятків.* При цьому програміст не забезпечує перевірку результату виконання кожної конкретної операції або функції "вручну". У цьому й полягає принципова перевага системи оброблення винятків, оскільки саме вона відповідає за код оброблення помилок, який раніше доводилося "вручну" вводити в громіздкі програми.

У цьому розділі також розглянемо С++-оператори динамічного розподілу пам'яті: **new** і **delete**. Як пояснювалося вище у цьому навчальному посібнику, якщо оператор **new** не може виділити необхідну пам'ять, то він генерує винятки. Як саме обробляються такі винятки якраз і буде розглянуто у цьому розділі. Окрім цього, тут ми навчимося перевизначати оператори **new** і **delete**, що дасть змогу визначати власні схеми виділення області пам'яті.

8.1. Основні особливості оброблення виняткових ситуацій

Керування С++-механізмом оброблення винятків тримається на трьох ключових словах: **try**, **catch** і **throw**. Вони утворюють взаємопов'язану підсистему, у якій використання одного з них припускає застосування іншого. Спершу спробуємо отримати загальне уявлення про ті вигоди, які вони надають програмісту під час оброблення виняткових ситуацій.

Оброблення винятків – це системні засоби, за допомогою яких програма може справитися з помилками тривалості виконання.

8.1.1. Системні засоби оброблення винятків

Робота системних засобів оброблення винятків полягає в такому. Програмні настанови, які потрібно проконтролювати на предмет винятків, поміщаються в **try**-блок. Якщо виняток (тобто помилка) таки виникає у процесі виконання блоку, то він дає знати про себе шляхом *викидання* певної інформації (за допомогою ключового слова **throw**). Цей *викинутий виняток* можна перехопити програмно за допомогою **catch**-блоку і відповідно обробити.

---

*Настанова **throw** генерує виняток, який перехоплюється **catch**-настановою.*

Отже, програмні настанови, у яких можливе виникнення виняткових ситуацій, мають виконуватися у межах **try**-блоку. Будь-яка функція, що викликається з

цього **try**-блоку, також піддається контролю. Винятки, які можуть бути викинуті контрольованими настановами, перехоплюються **catch**-настановою, що йде безпосередньо за **try**-блоком, у якому фіксуються ці "викиди". Загальний формат **try**- і **catch**-блоків має такий вигляд:

```
try {
    // try-блок (блок коду програми, що підлягає перевірці на наявність помилок)
}
catch(aType arg) {
    // catch-блок (обробник винятків типу aType)
}
catch(bType arg) {
    // catch-блок (обробник винятків типу bType)
}
catch(cType arg) {
    // catch-блок (обробник винятків типу cType)
}
//....
catch(nType arg) {
    // catch-блок (обробник винятків типу nType)
}
```

Блок **try** повинен містити програмні настанови, який, на Вашу думку, мають перевірятися на предмет виникнення помилок. Цей блок може містити тільки декілька настанов певної функції або охоплювати весь код функції **main**() (у цьому випадку, по суті, "під ковпаком" системи оброблення винятків знаходитиметься вся програма).

Після "викиду" виняток перехоплюється відповідною настановою **catch**, яка здійснює його оброблення. З одним **try**-блоком може бути пов'язана не одна, а декілька **catch**-настанов. Яка саме з них буде виконуватися, визначається типом винятку. Іншими словами, виконуватиметься та **catch**-настанова, тип винятку якої (тобто тип даних, який задається в **catch**-настанові) збігається з типом згенерованого винятку (а всі інші будуть проігноровані). Після перехоплення винятку параметр *arg* прийме його значення. Так само можуть перехоплюватися дані будь-якого типу, в т.ч. об'єкти класів, що були створені програмістом.

---

*Щоб виняток перехоплювати, необхідно забезпечити його "викид" в **try**-блоці.*

Загальний формат настанови **throw** має такий вигляд:

```
throw exception;
```

У цьому записі за допомогою елемента *exception* задається виняток, що згенерується настановою **throw**. Якщо цей виняток підлягає перехопленню, то настанова **throw** має виконуватися або в самому блоці **try**, або в будь-якій функції, яка викликається з нього (тобто прямо або опосередковано).

Вартоа' нати! *Якщо у програмі забезпечується "викид" винятку, для якого не передбачено відповідну **catch**-настанову, то відбудеться аварійне завершення*

роботи коду програми, що викликається стандартною бібліотечною функцією **terminate()**. *За замовчуванням функція* **terminate()** *викликає функцію* **abort()** *для зупинки програми, але при бажанні можна визначити власний обробник для її завершення. За подробицями щодо оброблення цієї ситуації необхідно звернутися до документації, що додається до Вашого компілятора.*

Розглянемо простий приклад оброблення винятків засобами мови C++.

Код програми 8.1. Демонстрація механізму оброблення винятків

```
#include <iostream>              // Для потокового введення-виведення
using namespace std;             // Використання стандартного простору імен

int main()
{
	cout << "Початок" << endl;

	try { // Початок try-блоку
		cout << "У try-блоці" << endl;
		throw 99; // Генерування помилки
		cout << "Ця настанова не буде виконана.";
	}
	catch(int c) { // Перехоплення помилки
		cout << "Значення перехопленого винятку дорівнює: " << c << endl;
	}

	cout << "Кінець програми";

	getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Початок.
У try-блоці
Значення перехопленого винятку дорівнює: 99
Кінець програми
```

Розглянемо уважно код цієї програми. Як бачите, тут **try**-блок містить три настанови, а настанова **catch**(**int** c) призначена для оброблення винятку цілочисельного типу. У цьому **try**-блоці виконуються тільки дві з трьох настанов: **cout** і **throw**. Після генерування винятку керування передається **catch**-виразу, при цьому виконання **try**-блоку припиняється. Необхідно розуміти, що **catch**-настанова не викликається, а просто з неї продовжується виконання програми після "викиду" винятку. Стек програми автоматично налаштовується відповідно до ситуації, що виникла. Тому **cout**-настанова, що знаходиться після **throw**-настанови, ніколи не виконається.

Після виконання **catch**-блоку керування програмою передається настанові, що знаходиться за цим блоком. Тому обробник винятків має виправити помилку, що спричинила його виникнення, щоб програма могла нормально продовжити виконання. У випадках, коли помилку виправити не можна, **catch**-блок зазвичай завершується зверненням до функцій **exit**() або **abort**() (див. розд. 8.1.2).

Як уже зазначалося вище, тип винятку повинен збігатися з типом, заданим у **catch**-настанові. Наприклад, якщо в попередній програмі тип винятку **int**, який було вказано в **catch**-виразі, замінити типом **double**, то виняток не перехопиться, тобто відбудеться аварійне завершення роботи коду програми. Ось як виглядають наслідки внесення такої зміни.

Код програми 8.2. Демонстрація не коректної роботи коду програми

```
#include <iostream>          // Для потокового введення-виведення
using namespace std;         // Використання стандартного простору імен

int main()
{
	cout << "Початок" << endl;

	try { // Початок try-блоку
		cout << "У try-блоці" << endl;
		throw 99; // генерування помилки
		cout << "Ця настанова не буде виконана.";
	}
	catch(double c) { // Перехоплення винятку типу int не відбудеться.
		cout << "Значення перехопленого винятку дорівнює: ";
		cout << c << endl;
	}

	cout << "Кінець програми";
	getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Початок.
У try-блоці
Ця настанова не буде виконана.
```

Такі результати виконання цієї програми пояснюються тим, що винятки цілочисельного типу не перехоплює настанова **catch**(**double** c).

8.1.2. Використання функцій exit() і abort() для завершення роботи коду програми

Функції **exit**() і **abort**() входять до складу стандартної бібліотеки C++ і їх часто використовують при програмуванні мовою C++. Обидві вони забезпечують завершення роботи коду програми, але відбувається це по-різному.

Виклик функції **exit**() негайно приводить до "правильного" припинення роботи коду програми[1]. Зазвичай цей спосіб завершення роботи використовують для зупинки програми під час виникнення непоправної помилки, яка робить подальше її виконання безглуздим або небезпечним. Для використання функції **exit**() потрібно залучити до програми заголовок <**cstdlib**>. Її прототип має такий вигляд:

```
void exit(int status);
```

[1] "Правильне" закінчення означає виконання стандартної послідовності дій після завершення роботи.

Оскільки функція **exit**() викликає негайне завершення роботи коду програми, то вона не передає керування процесу, який її викликає, і не повертає ніякого значення. Проте процесу, що її викликає, як код завершення, передається значення параметра *status*. За домовленістю нульове значення параметра *status* вказує на успішне завершення роботи коду програми. Будь-яке інше його значення свідчить про те, що завершення роботи коду програми є помилковим. Для індикації успішного завершення роботи можна також використовувати константу EXIT_SUCCESS, а для індикації помилки – константу EXIT_FAILURE. Ці константи визначаються у заголовку <**cstdlib**>.

Прототип функції **abort**() має такий вигляд:

void abort();

Функція **abort**() викликає негайне завершення роботи коду програми. Але, на відміну від функції **exit**(), вона не повертає операційній системі ніякої інформації про статус завершення роботи коду програми і не здійснює стандартної ("правильної") послідовності дій під час зупинки програми. Для використання функції **abort**() потрібно залучити до програми заголовок **<cstdlib>**. Функцію **abort**() можна назвати аварійним "стоп-краном" для С++-програми. Її необхідно використовувати тільки після виникнення непоправної помилки.

Останнє повідомлення про аварійне завершення роботи коду програми (Abnormal program termination) може відрізнятися від наведеного в результатах виконання попереднього прикладу. Це залежить від використовуваного Вами компілятора. Виняток, що генерує функція, яку було викликано з **try**-блоку, можна перехопити цим самим **try**-блоком. Розглянемо, наприклад, таку цілком коректну програму.

Код програми 8.3. Демонстрація механізму генерування винятку функцією, що викликається з try-блоку

```
#include <iostream>            // Для потокового введення-виведення
using namespace std;           // Використання стандартного простору імен

void Xtest(int test)
{
	cout << "У функції Xtest() значення test дорівнює: " << test << endl;
	if(test) throw test;
}

int main()
{
	cout << "Початок" << endl;

	try { // Початок try-блоку
		cout << "У try-блоці" << endl;
		Xtest(0);
		Xtest(1);
		Xtest(2);
	}

	catch(int c) { // Перехоплення помилки
		cout << "Значення перехопленого винятку дорівнює: ";
```

```
                cout << c << endl;
        }
        cout << "Кінець програми";
        getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Початок.
У try-блоці
У функції Xtest() значення test дорівнює: 0
У функції Xtest() значення test дорівнює: 1
Значення перехопленого винятку дорівнює: 1
Кінець програми
```

Блок **try** можна локалізувати у межах роботи самої функції. У цьому випадку під час кожного її виконання запускається і оброблення винятків, пов'язаних з роботою цією функцією. Розглянемо таку навчальну програму.

Код програми 8.4. Демонстрація механізму локалізації блоку try у рамках роботи самої функції

```
#include <iostream>           // Для потокового введення-виведення
using namespace std;          // Використання стандартного простору імен

// Функціонування блоків try/catch поновлюється під час кожного входження у функцію.
void Xhandler(int test)
{
        try {
                if(test) throw test;
        }
        catch(int c) {
                cout << "Перехоплення! Виняток №: " << c << endl;
        }
}

int main()
{
        cout << "Початок" << endl;

        Xhandler(1);
        Xhandler(2);
        Xhandler(0);
        Xhandler(3);

        cout << "Кінець програми";
        getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Початок.
Перехоплення! Виняток №: 1
Перехоплення! Виняток №: 2
Перехоплення! Виняток №: 3
Кінець програми
```

Як бачите, програма згенерувала три різних винятки. Після кожного винятку функція Xhandler() передавала керування у функцію **main**(). Коли вона знову викликалася, поновлювалося і оброблення винятків.

У загальному випадку **try**-блок відновлює своє функціонування під час кожного входу в нього. Тому **try**-блок, який є частиною циклу, запускатиметься при кожному повторенні цього циклу.

8.1.3. Перехоплення винятків класового типу

Виняток може мати будь-який тип, у тому числі і класового типу, створеного програмістом. У реальних програмах більшість винятків мають саме класовий тип, а не вбудований тип. Ймовірно, тип класу найбільше підходить для опису помилки, яка потенційно може виникнути у програмі, як це показано у наведеному нижче прикладі. Інформація, яка міститься в об'єкті класу винятків, дає змогу спростити процес їх оброблення.

Код програми 8.5. Демонстрація механізму перехоплення винятків класового типу

```
#include <iostream>           // Для потокового введення-виведення
#include <cstring>            // Для роботи з рядковими типами даних
using namespace std;          // Використання стандартного простору імен

class myException {
public:
	char str[80];
	myException() { *str = 0; }
	myException(char *s) { strcpy(str, s); }
};

int main()
{
	int a, b;
	try {
		cout << "Введіть чисельник і знаменник: ";
		cin >> a >> b;
		if(!b) throw myException("Ділити на нуль не можна!");
		else cout << "Частка дорівнює " << a/b << endl;
	}
	catch(myException e) { // Перехоплення помилки
		cout << e.str << endl;
	}

	getch(); return 0;
}
```

Один з можливих результатів виконання цієї програми.

```
Введіть чисельник і знаменник: 10 0
Ділити на нуль не можна!
```

Після запуску програми користувачу пропонується ввести чисельник і знаменник. Якщо знаменник дорівнює нулю, то створюється об'єкт класу myException,

який містить інформацію про спробу ділення на нуль. Також клас myException інкапсулює інформацію про помилку, яка потім використовується обробником винятків для повідомлення користувача про те, що трапилося.

Безумовно, реальні винятки класового типу набагато складніші за клас myException. Як правило, створення винятків класового типу має сенс у тому випадку, якщо вони інкапсулюють інформацію, яка дає змогу обробнику винятків ефективно справитися з помилкою і за змогою відновлює працездатність програми.

8.1.4. Використання декількох catch-настанов

Як уже зазначалося вище, з **try**-блоком можна пов'язувати не одну, а декілька **catch**-настанов. Насправді саме такий підхід і застосовується найчастіше. Але при цьому всі **catch**-настанови повинні перехоплювати винятки різних типів. Наприклад, у наведеному нижче коді програми забезпечується перехоплення як цілих чисел, так і покажчиків на символи.

Код програми 8.6. Демонстрація механізму використання декількох catch-настанов

```
#include <iostream>            // Для потокового введення-виведення
using namespace std;           // Використання стандартного простору імен

// Тут можливе перехоплення винятків різних типів.
void Xhandler(int test)
{
	try {
		if(test) throw test;
		else throw "Значення дорівнює нулю.";
	}
	catch(int c) {
		cout << "Перехоплення! Виняток №: " << c << endl;
	}

	catch(char *str) {
		cout << "Перехоплення рядка: " << str << endl;
	}
}

int main()
{
	cout << "Початок" << endl;

	Xhandler(1);
	Xhandler(2);
	Xhandler(0);
	Xhandler(3);

	cout << "Кінець програми";
	getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Початок.
Перехоплення! Виняток №: 1
Перехоплення! Виняток №: 2
Перехоплення рядка: Значення дорівнює нулю.
Перехоплення! Виняток №: 3
Кінець програми
```

Як бачите, кожна **catch**-настанова відповідає тільки за винятки "свого" типу.

У загальному випадку **catch**-вирази перевіряються у порядку їх проходження, тобто виконується тільки той **catch**-блок, у якому тип заданого винятку збігається з типом винятку, що згенерувався. Всі інші **catch**-блоки ігноруються.

Перехоплення винятків базового класу. Важливо розуміти, як виконуються **catch**-настанови, пов'язані з похідними класами. Йдеться про те, що **catch**-вираз для базового класу відреагує збігом на винятки будь-якого похідного типу (тобто типу, виведеного з цього базового класу). Отже, якщо потрібно перехоплювати винятки як базового, так і похідного типів, то у **catch**-послідовності **catch**-настанову для похідного типу необхідно помістити перед **catch**-настановою для базового типу. Інакше **catch**-вираз для базового класу перехоплюватиме крім "своїх" і винятки всіх похідних класів. Розглянемо, наприклад, такий код програми.

Код програми 8.7. Демонстрація механізму перехоплення винятків базових і похідних типів

```
#include <iostream>              // Для потокового введення-виведення
using namespace std;             // Використання стандартного простору імен

class bClass {    // Оголошення класового типу
};
class dClass: public bClass {    // Оголошення класового типу
};

int main()
{
        dClass derived;

        try { throw derived; }
        catch(bClass ObjB) {cout << "Перехоплення винятку базового класу" << endl; }
        catch(dClass ObjD) {cout << "Це перехоплення не відбудеться" << endl; }

        getch(); return 0;
}
```

Оскільки тут об'єкт derived – це об'єкт класу **dClass**, який виведено з базового класу **bClass**, то виняток типу derived завжди перехоплюватиметься першим **catch**-виразом; друга ж **catch**-настанова при цьому ніколи не виконається. Одні компілятори відреагують на такий стан речей застережним повідомленням, інші можуть видати повідомлення про помилку. У будь-якому випадку, щоб виправити ситуацію, достатньо поміняти порядок слідування цих **catch**-настанов на протилежний.

8.2. Варіанти оброблення винятків

Окрім розглянутих вище С++-засобів оброблення винятків, існують й інші засоби, які створюють певні зручності для програмістів. Про них і йтиметься у цьому підрозділі.

8.2.1. Перехоплення всіх винятків

Іноді варто створити спеціальний обробник для перехоплення всіх винятків, а не винятків тільки певного типу. Для цього достатньо використовувати такий формат **catch**-блоку:

```
catch(...) {
    // Оброблення всіх винятків
}
```

У цьому записі занесені в круглі дужки крапки забезпечують збіг з будь-яким типом. Використання формату **catch**(...) продемонстровано в такому коді програми.

Код програми 8.8. Демонстрація механізму перехоплення винятків усіх типів

```
#include <iostream>              // Для потокового введення-виведення
using namespace std;             // Використання стандартного простору імен

void Xhandler(int test)
{
    try {
        if(test == 0) throw test;      // Генерує int-винятки
        if(test == l) throw 'a';        // Генерує char-винятки
        if(test == 2) throw 123.23; // Генерує double-винятки
    }
    catch(...) { // Перехоплення всіх винятків
        cout << "Перехоплення!" << endl;
    }
}

int main()
{
    cout << "Початок" << endl;
    Xhandler(0);
    Xhandler(1);
    Xhandler(2);

    cout << "Кінець програми";
    getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Початок.
Перехоплення!
Перехоплення!
Перехоплення!
Кінець програми
```

Як бачите, всі три **throw**-винятки перехоплені за допомогою однієї **catch**-настанови.

Часто доцільно використовувати настанову **catch**(...) як останній "рубіж" **catch**-послідовності. У цьому випадку вона забезпечує перехоплення винятків усіх інших типів (тобто не передбачених попередніми **catch**-виразами). Наприклад, розглянемо ще одну версію попереднього коду програми, у якій безпосередньо забезпечується перехоплення винятків цілочисельного типу, а перехоплення усіх інших можливих винятків здійснюється за допомогою настанови **catch**(...).

Код програми 8.9. Демонстрація механізму використання настанови catch(...) для перехоплення винятків усіх інших типів

```
#include <iostream>             // Для потокового введення-виведення
using namespace std;            // Використання стандартного простору імен

void Xhandler(int test)
{
	try {
		if(test == 0) throw test;      // Генерує int-винятки
		if(test == l) throw 'a';        // Генерує char-винятки
		if(test == 2) throw 123.23;            // Генерує double-винятки
	}
	catch(int c) { // Перехоплює int-винятки
		cout << "Перехоплення " << c << endl;
	}
	catch(...) { // Перехоплює усі інші винятки
		cout << "Перехоплення-перехоплення!" << endl;
	}
}

int main()
{
	cout << "Початок" << endl;

	Xhandler(0);
	Xhandler(1);
	Xhandler(2);

	cout << "Кінець програми";
	getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Початок.
Перехоплення 0
Перехоплення-перехоплення!
Перехоплення-перехоплення!
Кінець програми
```

Як підтверджує цей приклад, використання формату **catch**(...) як "останнього рубежу" **catch**-послідовності – це зручний спосіб перехопити всі винятки, які інколи не хочеться обробляти безпосередньо. Окрім цього, перехоплюючи абсолютно всі винятки, Ви запобігаєте можливості аварійного завершення роботи коду прог-

рами, яке може бути викликане якимсь непередбаченим (а значить, необробленим) винятком.

8.2.2. Накладання обмежень на тип винятків, які генеруються функціями

Існують засоби, які дають змогу обмежити тип винятків, котрі може генерувати функція за межами свого тіла. Можна також захистити функцію від генерування будь-яких винятків взагалі. Для формування цих обмежень необхідно внести у визначення функції **throw**-вираз. Загальний формат визначення функції з використанням **throw**-виразу має такий вигляд:

```
тип ім'я_функції (перелік_аргументів) throw (перелік_імен_типів)
{
    //...
}
```

У цьому записі елемент *перелік_імен_типів* повинен містити тільки ті імена типів даних, які дозволяється генерувати функції (елементи списку розділяються між собою комами). Генерування винятку будь-якого іншого типу призведе до аварійного завершення роботи коду програми. Якщо потрібно, щоб функція взагалі не могла генерувати винятки, то як цей елемент використовують порожній перелік.

Вартоа' нати! *При спробі згенерувати винятки, які не підтримуються функцією, викликається стандартна бібліотечна функція* unexpected()*. За замовчуванням вона викликає функцію* ***abort()****, яка забезпечує аварійне завершення роботи коду програми. Але при бажанні можна задати власний обробник процесу її завершення. За подробицями необхідно звернутися до документації, що додається до Вашого компілятора.*

На прикладі наведеної нижче програми показано, як можна обмежити типи винятків, які здатна генерувати функція.

Код програми 8.10. Демонстрація механізму накладання обмежень на тип винятків, які генеруються функцією

```
#include <iostream>              // Для потокового введення-виведення
using namespace std;             // Використання стандартного простору імен

// Ця функція може генерувати винятки тільки типу int, char i double.
void Xhandler(int test) throw (int, char, double)
{
    if(test == 0) throw test;   // Генерує int-винятки
    if(test == l) throw 'a';       // Генерує char-винятки
    if(test == 2) throw 123.23; // Генерує double-винятки
}

int main()
{
    cout << "Початок" << endl;
    try {
```

```
        Xhandler(0); // Спробуйте також передати функції Xhandler() аргументи 1 і 2.
    }

    catch(int c) { cout << "Перехоплення int-винятку" << endl; }

    catch(char c) { cout << "Перехоплення char-винятку" << endl; }

    catch(double d) { cout << "Перехоплення double-винятку" << endl; }

    cout << "Кінець програми";

    getch(); return 0;
}
```

У цьому коді програми функція Xhandler() може генерувати винятки тільки типу **int**, **char** і **double**. При спробі згенерувати винятки будь-якого іншого типу відбудеться аварійне завершення роботи коду програми (завдяки виклику функції unexpected()). Щоб переконатися у цьому, видаліть з **throw**-списку, наприклад, тип **int** і перезапустіть заново програму.

Важливо розуміти, що діапазон винятків, дозволених для генерування функції, можна обмежувати тільки типами, які вона генерує в **try**-блоці, звідки була викликана. Іншими словами, будь-який **try**-блок, розташований в тілі самої функції, може генерувати винятки будь-якого типу, якщо вони перехоплюються в тілі тієї ж самої функції. Обмеження застосовується тільки для ситуацій, коли "викид" винятків відбувається за межі функції.

Наступна зміна завадить функції Xhandler() генерувати будь-які зміни.

```
// Ця функція взагалі не може генерувати винятки!
void Xhandler(int test) throw ()
{
    /* Наведені нижче настанови більше не працюють.
    Тепер вони можуть викликати тільки аварійне завершення роботи коду програми*/
    if(test == 0) throw test;
    if(test == l) throw 'a';
    if(test == 2) throw 123.23;
}
```

8.2.3. Повторне генерування винятку

Для того, щоби повторно згенерувати винятки в його обробнику, необхідно використовувати **throw**-настанову без вказання типу винятку. У цьому випадку поточний виняток передається в зовнішню **try**/**catch**-послідовність. Найчастіше причиною для такого виконання настанови **throw** слугує прагнення мати доступ до одного винятку декільком обробникам. Наприклад, перший обробник винятків повідомлятиме про один аспект винятку, а другий – про щось інше. Винятки можна повторно згенерувати тільки в **catch**-блоці (або в будь-якій функції, яка викликається з цього блоку). При повторному генеруванні винятку сам виняток не перехоплюватиметься тією ж **catch**-настановою. Він пошириться на найближчу **try**/**catch**-послідовність.

Повторне генерування винятку типу **char** * продемонстровано у наведеному нижче коді програми.

Код програми 8.11. Демонстрація механізму повторного генерування винятку типу char *

```
#include <iostream>            // Для потокового введення-виведення
using namespace std;           // Використання стандартного простору імен

void Xhandler() {
	try {
		throw "Привіт"; // Генерує винятки типу char *
	}
	catch(char *) { // Перехоплює винятки типу char *
		cout << "Перехоплення винятку у функції Xhandler" << endl;
		throw; // Повторне генерування винятку типу char *,
		// яке перехоплюватиметься поза функцією Xhandler.
	}
}

int main()
{
	cout << "Початок" << endl;

	try {
		Xhandler();
	}
	catch(char *) {
		cout << "Перехоплення винятку у функції main()" << endl;
	}

	cout << "Кінець програми";

	getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Початок.
Перехоплення винятку у функції Xhandler.
Перехоплення винятку у функції main().
Кінець програми
```

8.3. Оброблення винятків, згенерованих оператором new

Як зазначалося вище, оператор **new** генерує винятки, якщо не вдається задовольнити запит на виділення області пам'яті. Там опис оброблення винятків такого типу було відкладено на потім. Якраз тепер настав час розглянути цю проблему детальніше.

Спершу необхідно зазначити, що у цьому розділі описано поведінку оператора **new** відповідно до стандарту мови C++. Проте, дії, які здійснює система при неуспішному використанні оператора **new**, з моменту появи мови C++ змінювалися

вже декілька разів. Спочатку оператор **new** повертав при невдачі значення **NULL**. Пізніше така поведінка була замінена генеруванням винятку, ім'я якого також мінялося кілька разів. Нарешті, було вирішено, що оператор **new** генеруватиме виняток за замовчуванням, але як альтернативний варіант він може повертати і нульовий покажчик. Отже, оператор **new** у різний час був реалізований різними способами. І хоча всі сучасні компілятори реалізують оператор **new** відповідно до стандарту мови С++, компілятори "поважнішого" віку можуть містити відхилення від нього. Якщо наведені тут приклади кодів програм не працюють коректно з Вашим компілятором, то зверніться до документації, що додається до компілятора, і поцікавтеся, як саме він реалізує функціонування оператора **new**.

Згідно зі стандартом мови С++, у випадку неможливості задовольнити запит на виділення області пам'яті, потрібної операторові **new**, генерується виняток типу **bad_alloc**. Якщо Ваша програма не перехопить його, то вона буде завчасно завершена. Хоча така поведінка виправдана для коротких прикладів програм, в реальних додатках необхідно перехоплювати цей виняток і розумно обробляти його. Щоб отримати доступ до винятку типу **bad_alloc**, потрібно залучити до програми заголовок **<new>**.

Розглянемо приклад використання оператора **new**, поміщеного в **try**/**catch**-блок для вистежування невдалих результатів запиту на виділення області пам'яті.

Код програми 8.12. Демонстрація механізму оброблення винятків, згенерованих оператором new

```
#include <iostream>          // Для потокового введення-виведення
#include <new>               // Для перевизначення операторів new i delete
using namespace std;         // Використання стандартного простору імен

int main()
{
	int *p;
	try {
		p = new int[32];   // Запит на виділення області пам'яті
		                   // для 32-елементного int-масиву
	}
	catch(bad_alloc xa) {
		cout << "Пам'ять не виділена" << endl;
		return 1;
	}

	for(int i=0; i<32; i++) p[i] = i;
	for(int i=0; i<32; i++) cout << p[i] << " ";
	delete [] p; // Звільнення пам'яті

	getch(); return 0;
}
```

Під час невдалого виконання оператора **new** виняток у цьому коді програми буде перехоплено **catch**-настановою. Цей самий підхід можна використовувати для вистежування будь-яких помилок, взаємопов'язаних з використанням оператора **new**: достатньо помістити кожну **new**-настанову в **try**-блок.

Альтернативна форма оператора new – nothrow. Стандарт мови C++ при невдалій спробі виділення області пам'яті оператором **new** замість генерування винятку також може повертати значення **NULL**. Ця форма використання оператора **new** особливо корисна у процесі компілювання старих програм із застосуванням сучасного C++-компілятора. Вона також корисна при заміні викликів функції **malloc**() оператором **new**. Це звичайна практика у процесі перекладу С-коду програми на мову програмування C++. Отже, формат оператора **new** має такий вигляд:

p_var = **new**(**nothrow**) *тип*;

У цьому записі елемент *p_var* – це покажчик на змінну типу *тип*. Цей **nothrow**-формат оператора **new** працює подібно до оригінальної версії оператора **new**, яка використовувалася кілька років тому. Оскільки оператор **new**(**nothrow**) повертає при невдачі значення **NULL**, його можна "упровадити" в старий код програми, не вдаючись до оброблення винятків. Проте в нових програмах, написаних мовою C++, все ж таки краще мати справу з генеруванням винятків.

У наведеному нижче прикладі показано, як використовується альтернативний варіант **new**(**nothrow**). Неважко здогадатися, що нижче наведено можливий варіант попереднього коду програми.

Код програми 8.13. Демонстрація механізму використання nothrow як старої версії оператора new

```
#include <iostream>         // Для потокового введення-виведення
#include <new>              // Для перевизначення операторів new і delete
using namespace std;        // Використання стандартного простору імен

int main()
{
	int *p;
	p = new(nothrow) int[32];	// Використання nothrow-версії

	if(!p) {
		cout << "Пам'ять не виділена" << endl;
		return 1;
	}

	for(int i=0; i<32; i++) p[i] = i;
	for(int i=0; i<32; i++) cout << p[i] << " ";
	delete [] p; // Звільнення пам'яті
	getch(); return 0;
}
```

У цьому коді програми під час використання **throw**-версії після кожного запиту на виділення області пам'яті необхідно перевіряти значення покажчика, що повертається оператором **new**.

8.4. Механізми перевизначення операторів new і delete

Оскільки **new** і **delete** – оператори, то їх також можна перевизначати. Хоча тема перевизначення операторів **new** і **delet** розглядалася в розд. 4 **[9]**, проте деяка їх

частина була відкладена до знайомства з темою оброблення винятків, оскільки правильно перевизначений оператор **new** (який відповідає стандарту мови С++) повинен у разі невдачі генерувати виняток типу **bad_alloc**. З кількох причин Вам варто створити власну версію оператора **new**. Наприклад, створіть процедури виділення області пам'яті, які, якщо область множини виявиться вичерпаною, автоматично починають використовувати дисковий файл як віртуальну пам'ять. У будь-якому випадку реалізація перевизначення цих операторів є не складнішою за перевизначення будь-яких інших.

Нижче наведено скелет функцій, які перевизначають оператори **new** і **delete**.

```
// Виділення області пам'яті для об'єкта
void *operator new(size_t size)
{
    /* У разі неможливості виділити пам'ять генерується виняток
     типу bad_alloc. Конструктор викликається автоматично. */

    return pointer_to_memory;
}

// Видалення об'єкта.
void operator delete(void *p)
{
    /* Звільняється область пам'яті, яка адресується покажчиком р.
    Деструктор викликається автоматично. */
}
```

Тип **size_t** спеціально визначено, щоб забезпечити зберігання розміру максимально можливої області пам'яті, яку можна виділити для об'єкта[1]. Параметр size визначає кількість байтів пам'яті, потрібної для зберігання об'єкта, для якого виділяється пам'ять. Іншими словами, це об'єм пам'яті, який повинна виділити Ваша версія оператора **new**. Операторна функція **new** повинна повертати покажчик на пам'ять, що виділяється нею, або генерувати винятки типу **bad_alloc** у випадку виникнення помилки. Окрім цих обмежень, операторна функція **new** може виконувати будь-які потрібні дії. Під час виділення області пам'яті для об'єкта за допомогою оператора **new** (його початкової форми або Вашої власної) автоматично викликається конструктор об'єкта.

Функція **delete** отримує покажчик на область пам'яті, яку необхідно звільнити. Потім вона повинна повернути цю область пам'яті системі. Під час видалення об'єкта автоматично викликається його деструктор.

Щоб виділити пам'ять для масиву об'єктів, а потім звільнити її, необхідно використовувати такі формати операторів **new** і **delete**.

```
// Виділення області пам'яті для масиву об'єктів
void *operator new[](size_t size)
{
    /* У разі неможливості виділити пам'ять генерується виняток
    типу bad_alloc. Кожен конструктор викликається автоматично. */
```

[1] Тип **size_t**, по суті, – це цілочисельний тип без знаку.

```
        return pointer_to_memory;
    }

    // Видалення масиву об'єктів.
    void operator delete[](void *p)
    {
        /* Звільняється область пам'яті, яка адресується покажчиком p.
         При цьому автоматично викликається деструктор для
         кожного елемента масиву. */
    }
```

Під час виділення області пам'яті для масиву автоматично викликається конструктор кожного об'єкта, а при звільненні масиву автоматично викликається деструктор кожного об'єкта. Це означає, що для виконання цих дій не потрібно безпосередньо програмувати їх.

Оператори **new** і **delete**, як правило, перевизначаються відносно класу. Заради простоти у наведеному нижче прикладі використовується не нова схема розподілу пам'яті, а перевизначені функції **new** і **delete**, які просто викликають С-орієнтовані функції виділення області пам'яті **malloc**() і **free**(). У своєму власному додатку Ви можете реалізувати будь-який метод виділення області пам'яті.

Щоб перевизначити оператори **new** і **delete** для конкретного класу, достатньо зробити ці перевизначені операторні функції членами цього класу. У наведеному нижче прикладі коду програми оператори **new** і **delete** перевизначаються для класу kooClass. Це перевизначення дає змогу виділити пам'ять для об'єктів і масивів об'єктів, а потім звільнити її.

Код програми 8.14. Демонстрація механізму перевизначення операторів new і delete

```
#include <iostream>          // Для потокового введення-виведення
#include <new>               // Для перевизначення операторів new і delete
#include <cstdlib>           // Для використання бібліотечних функцій
using namespace std;         // Використання стандартного простору імен

class kooClass {// Оголошення класового типу
    int x, y, z; // Тривимірні координати
public:
    kooClass() { x = y = z = 0; cout << "Створення об'єкта 0, 0, 0" << endl; }
    kooClass(int c, int d, int f) { x = c; y = d; z = f;
        cout << "Створення об'єкта " << c << ", ";
        cout << d << ", " << f << endl; }
        ~kooClass() { cout << "Руйнування об'єкта" << endl; }
    void *operator new(size_t size);
    void *operator new[](size_t size);
    void operator delete(void *p);
    void operator delete[](void *p);
    void showB(char *s);
};

// Перевизначення оператора new для класу kooClass.
void *kooClass::operator new(size_t size)
```

```
{
	void *p;
	cout << "Виділення області пам'яті для об'єкта класу kooClass" << endl;
	p = malloc(size);

	// Генерування винятку у разі невдалого виділення області пам'яті.
	if(!p) {
		bad_alloc ba;
		throw ba;
	}
	return p;
}

// Перевизначення оператора new для масиву об'єктів типу kooClass.
void *kooClass::operator new[](size_t size)
{
	void *p;

	cout << "Виділення області пам'яті для масиву kooClass-об'єктів" << endl;
	// Генерування винятку при невдачі.
	p = malloc(size);
	if(!p) {
		bad_alloc ba;
		throw ba;
	}

	return p;
}

// Перевизначення оператора delete для класу kooClass.
void kooClass::operator delete(void *p)
{
	cout << "Видалення об'єкта класу kooClass" << endl;
	free(p);
}

// Перевизначення оператора delete для масиву об'єктів типу kooClass.
void kooClass::operator delete[](void *p)
{
	cout << "Видалення масиву об'єктів типу kooClass" << endl;
	free(p);
}

// Відображення тривимірних координат x, y, z.
void kooClass::showB(char *s)
{
	cout << "Координати об'єкта <" << s << ">: ";
	cout << "\t\tx= " << x << ", y= " << y << ", z= " << z << endl;
}
```

```
int main()
{
    kooClass *p1, *p2;
    try {
        p1 = new kooClass[3];       // Виділення області пам'яті для масиву
        p2 = new kooClass(5, 6, 7);         // Виділення області пам'яті для об'єкта
    }
    catch(bad_alloc ba) {
        cout << "Помилка під час виділення області пам'яті" << endl;
        return 1;
    }

    p1[1].showB("Базовий клас: ");
    p2->showB("Базовий клас: ");

    delete [] p1;         // Видалення масиву
    delete p2;            // Видалення об'єкта
    getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Виділення області пам'яті для масиву kooClass-об'єктів.
Створення об'єкта 0, 0, 0
Створення об'єкта 0, 0, 0
Створення об'єкта 0, 0, 0
Виділення області пам'яті для об'єкта класу kooClass.
Створення об'єкта 5, 6, 7
0, 0, 0
5, 6, 7
Руйнування об'єкта
Руйнування об'єкта
Руйнування об'єкта
Видалення масиву об'єктів типу kooClass.
Руйнування об'єкта
Видалення об'єкта класу kooClass.
```

Перші три повідомлення Створення об'єкта 0, 0, 0 видані конструктором класу kooClass (який не має параметрів) під час виділення області пам'яті для триелементного масиву. Як уже зазначалося вище, під час виділення області пам'яті для масиву автоматично викликається конструктор кожного елемента. Повідомлення Створення об'єкта 5, 6, 7 видано конструктором класу kooClass (який приймає три аргументи) під час виділення області пам'яті для одного об'єкта. Перші три повідомлення Руйнування об'єкта видані деструктором внаслідок видалення триелементного масиву, оскільки при цьому автоматично викликався деструктор кожного елемента масиву. Останнє повідомлення Руйнування об'єкта видане під час видалення одного об'єкта класу kooClass. Важливо розуміти, що, коли оператори **new** і **delete** перевизначені для конкретного класу, то внаслідок їх використання для даних інших типів будуть задіяні оригінальні версії операторів **new** і **delete**. Це означає, що при додаванні у функцію **main**() наступного рядка буде виконано стандартну версію оператора **new**:

```
int *f = new int; // Використовується стандартна версія оператора new.
```

Вартоа' нати! *Оператори* ***new*** *і* ***delete*** *можна перевизначати глобально. Для цього достатньо оголосити їх операторні функції поза класами. У цьому випадку стандартні версії С++-операторів* ***new*** *і* ***delete*** *ігноруються взагалі, і в усіх запитах на виділення області пам'яті використовуються їх перевизначені версії. Безумовно, якщо Ви при цьому визначите версію операторів* ***new*** *і* ***delete*** *для конкретного класу, то ці "класові" версії застосовуватимуться під час виділення області пам'яті (та її звільнення) для об'єктів цього класу. В усій же решті випадків використовуватимуться глобальні операторні функції.*

Перевизначення nothrow-версії оператора new. Можна також створити перевизначені **nothrow**-версії операторів **new** і **delete**. Для цього використовуються такі механізми:

```
// Перевизначення nothrow-версії оператора new
void *operator new(size_t size, const nothrow_t &n)
{
	// Виділення області пам'яті.
	if(success) return pointer_to_memory;
	else return 0;
}

// Перевизначення nothrow-версії оператора new для масиву.
void *operator new[](size_t size, const nothrow_t &n)
{
	// Виділення області пам'яті.
	if(success) return pointer_to_memory;
	else return 0;
}

// Перевизначення nothrow-версії оператора delete.
void operator delete(void *p, const nothrow_t &n)
{
	// Звільнення пам'яті.
}

// Перевизначення nothrow-версії оператора delete для масиву.
void operator delete[](void *p, const nothrow_t &n)
{
	// Звільнення пам'яті.
}
```

Тип **nothrow_t** визначається в заголовку **<new>**. Параметр типу **nothrow_t** не використовується. Як вправу проекспериментуйте з **throw**-версіями операторів **new** і **delete** самостійно.

Розділ 9. ОРГАНІЗАЦІЯ С++-СИСТЕМИ ВВЕДЕННЯ-ВИВЕДЕННЯ ПОТОКОВОЇ ІНФОРМАЦІЇ

З самого початку висвітлення матеріалу у цьому навчальному посібнику використовувалася С++-система введення-виведення потокової інформації, але не наводилось детальних пояснень з механізму її реалізації. Оскільки С++-система введення-виведення побудована на ієрархії класів, то її теорію і деталі реалізації неможливо було засвоїти, не розглянувши спочатку механізм реалізації класів, успадкування в класах і оброблення винятків. Якраз після цього і настає момент для детального вивчення С++-засобів введення-виведення потокової інформації.

У цьому розділі будуть розглядатися засоби як консольного, так і файлового введення-виведення потокової інформації. Необхідно зразу ж відзначити, що С++-система введення-виведення – достатньо широка тема. Тому у поданому нижче матеріалі описано тільки найважливіші та часто вживані засоби організації С++-системи введення-виведення. Зокрема, спочатку дізнаємося про те, що розуміють під потоками у мові програмування С++, про перевизначення операторів "**<<**" і "**>>**" для введення та виведення об'єктів, про форматування різних типів даних, а також про використання маніпуляторів введення-виведення. На завершення розділу переглянемо засоби файлового введення-виведення потокової інформації.

9.1. Порівняння С- та С++-систем введення-виведення

На сьогодні існують дві версії бібліотеки об'єктно-орієнтованого введення-виведення даних, причому обидві широко використовуються програмістами: стара [дод. А, 9], що базується на оригінальних специфікаціях мови С, і нова – визначається стандартом мови програмування С++. Стара бібліотека введення-виведення даних підтримується за рахунок заголовного файлу <**iostream.h**>, а нова – за допомогою заголовка <**iostream**>. Нова бібліотека введення-виведення даних загалом є оновленою і вдосконаленою версією старої. *Основна відмінність* між ними полягає в механізмі реалізації засобів введення-виведення потокової інформації, а не у тому, як їх потрібно використовувати.

З погляду програміста, є дві істотні відмінності між старою С- і новою С++-бібліотеками введення-виведення. *По-перше*, нова бібліотека містить ряд додаткових засобів і визначає декілька нових типів даних. Тому нову бібліотеку С++-системи введення-виведення можна вважати надбудовою над старою С-системою. Практично всі програми, що були написані раніше з використанням старої бібліотеки, успішно компілюються тепер за наявності нової бібліотеки, не вимагаючи внесення будь-яких значних змін у самій програмі. *По-друге*, стара бібліотека С-системи введення-виведення була визначена в глобальному просторі імен, а нова використовує простір імен **std** (пригадайте, що простір імен **std** використовується всіма бібліотеками стандарту мови програмування С++). Оскільки С-бібліотека

введення-виведення даних вже дещо застаріла, то у цьому навчальному посібнику описується тільки нова її версія. Водночас велика частина наведеного нижче матеріалу повною мірою стосується і старої бібліотеки.

9.2. Поняття "потоків" у мові програмування С++

Принциповим для розуміння С++-системи введення-виведення є те, що вона базується на понятті "потоку". *Потік* (**stream**) – це загальний логічний інтерфейс з різними пристроями, з яких складається комп'ютер. Потік або синтезує інформацію, або споживає її, після чого зв'язується з будь-яким фізичним пристроєм за допомогою С++-системи введення-виведення. Характер поведінки всіх потоків є однаковим, незважаючи на різні фізичні пристрої, з якими вони пов'язуються. Оскільки різні потоки діють однаково, то практично до всіх типів пристроїв можна застосувати одні і ті ж самі функції та оператори введення-виведення. Наприклад, методи, що використовуються для виведення даних на екран, також можна використовувати для виведення їх на друкувальний пристрій або для запису у дисковий файл.

9.2.1. Файлові С++-потоки

У загальному випадку потік можна назвати логічним інтерфейсом з файлом. У мові програмування С++ до терміну "файл" належать дискові файли, екран монітора, клавіатура, порти, файли на магнітній стрічці й ін. Хоча файли між собою відрізняються за формою і можливостями представлення, проте робота з різними потоками інформації є однаковою. Перевага цього підходу (з погляду програміста) полягає у тому, що один пристрій комп'ютера може бути подібним до будь-якого іншого. Це означає, що потік забезпечує інтерфейс, який узгоджується *зі всіма пристроями.*

Потік зв'язується з файлом у процесі виконання операції відкриття файлу, а відокремлюється від нього за допомогою операції його закриття.

Існує два типи потоків: текстовий і двійковий. *Текстовий потік* використовують для введення-виведення символів. При цьому можуть відбуватися деякі перетворення символів. Наприклад, під час виведення символ нового рядка може перетворюватися у послідовність символів: повернення каретки і переходу на новий рядок. Тому часто не буває взаємно-однозначної відповідності між тим, що посилається у потік, і тим, що насправді записується у файл. *Двійковий потік* можна використовувати з даними будь-якого типу, причому у цьому випадку ніякого перетворення символів не здійснюється, тобто між тим, що посилається у потік, і тим, що потім реально міститься у файлі, існує взаємно-однозначна відповідність.

Розглядаючи потоки, необхідно розуміти, що вкладається у поняття "поточної позиції". Наприклад, якщо довжина файлу дорівнює 100 байтів, і відомо, що вже прочитано його половину, то наступна операція зчитування почнеться на 50-му байті, який у цьому випадку і є поточною позицією.

Поточнаапо' иція *– це те місце у файлі, з якого буде виконуватися наступна операція доступу до його даних.*

Отже, у мові програмування C++ механізм введення-виведення потокової інформації функціонує з використанням логічного інтерфейсу, який називається *потоком*. Всі потоки мають аналогічні властивості, які дають змогу використовувати однакові функції введення-виведення, незалежно від того, з файлом якого типу існує зв'язок. Під *файлом* розуміють реальний фізичний пристрій, який містить дані. Якщо файли можуть відрізнятися між собою, то потоки – ні. Водночас, якщо деякі пристрої можуть не підтримувати всі операції введення-виведення даних (наприклад, операції з довільною вибіркою), то і пов'язані з ними потоки теж не підтримуватимуть ці операції.

9.2.2. Вбудовані C++-потоки

У мові програмування C++ міститься ряд вбудованих однобайтових (8-бітових) потоків (**cin**, **cout**, **cerr** і **clog**), які автоматично відкриваються, як тільки програма починає виконуватися. Як уже зазначалося вище, **cin** – це стандартний вхідний, а **cout** – стандартний вихідний потік. Потоки **cerr** і **clog** (вони призначені для виведення інформації про помилки) також пов'язані із стандартним виведенням даних. Різниця між ними полягає у тому, що потік **clog** є буферизованим, а потік **cerr** – ні. Це означає, що будь-які вихідні дані, послані у потік **cerr**, будуть негайно виведені на екран, а при використанні потоку **clog** дані спочатку записуються в буфер, а реальне їх виведення починається тільки тоді, коли буфер повністю заповнено. Зазвичай потоки **cerr** і **clog** використовують для виведення інформації на екран про стан відлагодження програми або її помилки.

У мові програмування C++ також передбачено двобайтові (16-бітові) символьні версії стандартних потоків, що іменуються **wcin**, **wcout**, **wcerr** і **wclog**. Вони призначені для підтримки таких мов як китайська, японська та інших східноазійських, для представлення яких потрібні великі символьні набори. У цьому навчальному посібнику двобайтові стандартні потоки не розглядаються.

За замовчуванням стандартні C++-потоки зв'язуються з монітором, але програмним способом їх можна перенаправити на інші зовнішні пристрої або дискові файли. Перенаправлення може також виконати сама операційна система.

9.2.3. Класи потоків

Як було зазначено в розд. 9.1, C++-система введення-виведення використовує заголовок **<iostream>**, у якому для підтримки операцій введення-виведення даних визначено достатньо складну ієрархію класів. Ця ієрархія починається з *системи шаблонних класів*. Як наголошувалося в розд. 7, шаблонний клас визначає зміст виконуваних дій, не задаючи у повному обсязі типи даних, які він повинен обробляти. Маючи шаблонний клас, можна створювати його конкретні примірники. Для бібліотеки введення-виведення стандарт мови програмування C++ створює дві спеціалізації шаблонних класів: одну для 8-, а іншу для 16-бітових ("широких") символів. У цьому навчальному посібнику описуються шаблонні класи тільки для 8-бітових символів, оскільки вони найчастіше використовуються.

С++-система введення-виведення побудована на двох взаємопов'язаних, але різних ієрархіях шаблонних класів. ***Перша ієрархія*** виведена з класу низькорівневого введення-виведення **basic_streambuf**. Цей клас підтримує базові низькорівневі операції введення та виведення і забезпечує підтримку всієї С++-системи введення-виведення. Якщо Ви не плануєте займатися програмуванням спеціалізованих операцій введення-виведення даних, то Вам навряд чи доведеться використовувати безпосередньо клас **basic_streambuf**. ***Друга ієрархія*** класів, з якою С++-програмістам доводиться працювати безпосередньо, виведена з класу **basic_ios**. Це – клас високорівневого введення-виведення, який забезпечує *форматування даних*, *контроль помилок* і надає *статусну інформацію*, пов'язану з потоками введення-виведення[1]. Клас **basic_ios** використовується як базовий для декількох похідних від нього класів, у т.ч. класів **basic_istream**, **basic_ostream** і **basic_iostream**. Ці класи використовуються для створення потоків, призначених для окремого введення та виведення даних і їх одночасного введення-виведення.

Як уже зазначалося вище, бібліотека введення-виведення даних створює дві спеціалізовані ієрархії шаблонних класів: одну для 8-, а іншу для 16-бітових символів. У табл. 9.1 наведено перелік імен шаблонних класів і відповідних їм "символьних" версій.

Табл. 9.1. Перелік імен шаблонних класів і відповідних їм "символьних" версій

Шаблонні класи	Символьні класи
Базові низькорівневі операції введення-виведення	
basic_streambuf	**streambuf**
Високорівневі операції введення-виведення	
basic_ios	**ios**
basic_istream	**istream**
basic_ostream	**ostream**
basic_iostream	**iostream**
basic_fstream	**fstream**
basic_ifstream	**ifstream**
basic_ofstream	**ofstream**

У наступній частині цього розділу розглядатимемо імена символьних класів, оскільки саме їх найчастіше застосовують у програмах. Такі ж самі імена використовуються і старою бібліотекою введення-виведення. Ось тому стара і нова бібліотеки сумісні між собою на рівні початкового коду програми.

Вартоа' нати! *Клас* ***ios*** *містить багато функцій-членів класу і змінних, які керують основними операціями над потоками або відстежують результати їх виконання. Тому ім'я класу* ***ios*** *у цьому посібнику вживаєтьса достатньо часто.*

Нео! хідноапам'ятати! *Якщо помістити у програму заголовок* ***<iostream>****, то вона матиме доступ до цього важливого класу.*

[1] Клас **basic_ios** виведений з класу **ios_base**, який визначає ряд нешаблонних властивостей, що використовуються класом **basic_ios**.

9.3. Особливості механізмів перевизначення операторів введення-виведення даних

У розглянутих раніше прикладах програм для виконання операції введення або виведення "класових" даних створювалися функції-члени класів, призначення яких полягало тільки у тому, щоб ввести або вивести ці дані. Незважаючи на те, що у такому вирішенні цих питань немає нічого неправильного, проте у мові програмування С++ передбачено дещо вдаліший спосіб виконання операцій введення-виведення "класових" даних – шляхом перевизначення операторів введення-виведення "<<" і ">>".

У мові С++ оператор "<<" називається *оператором виведення* або *вставлення*, оскільки він вставляє символи у потік. Аналогічно оператор ">>" називається *оператором введення* або *вилучення*, оскільки він вилучає символи з потоку.

Як уже зазначалося вище, оператори введення-виведення вже перевизначені (у заголовку **<iostream>**) для того, щоби вони могли виконувати операції потокового введення або виведення даних будь-яких вбудованих С++-типів. У цьому підрозділі можна буде дізнатися про те, як визначити ці оператори для створення власних класів.

9.3.1. Створення перевизначених операторів виведення даних

Як простий приклад розглянемо механізм створення оператора виведення даних для уже відомої нам з попередніх розділів (див. поч. у розд. 4.1) такої версії класу kooClass:

```
class kooClass {// Оголошення класового типу
public:
	int x, y, z; // Тривимірні координати
	kooClass(int a, int b, int c) { x = a; y = b; z = c; }
};
```

Щоб створити операторну функцію виведення даних для об'єктів типу kooClass, необхідно перевизначити оператор виведення даних "<<". Один з можливих способів його реалізації має такий вигляд:

```
// Відображення тривимірних координат x, y, z
// Перевизначений оператор виведення даних для класу kooClass
ostream &operator<<(ostream &stream, kooClass obj)
{
	stream << obj.x << ", ";
	stream << obj.y << ", ";
	stream << obj.z << endl;

	return stream; // Повертає посилання на параметр stream
}
```

Розглянемо уважно цю операторну функцію, оскільки її вміст характерний для багатьох операторних функцій виведення даних. По-перше, зверніть увагу на те, що, згідно з оголошенням, вона повертає посилання на об'єкт типу **ostream**. Це дає змогу декілька звичайних операторів виведення даних об'єднати в одному

складеному виразі. По-друге, зверніть увагу також на те, що ця функція має два параметри. Перший є посиланням на потік, який використовується в лівій частині оператора "<<". Другим є об'єкт, який знаходиться у правій частині цього оператора[1]. Саме тіло операторної функції складається з настанов виведення трьох значень координат, що містяться в об'єкті типу kooClass, і настанови повернення потоку **stream**. Нижче наведено навчальну програму, у якій продемонстровано механізм використання перевизначеного оператора виведення даних.

Код програми 9.1. Демонстрація механізму реалізації перевизначеного оператора виведення даних

```
#include <iostream>              // Для потокового введення-виведення
using namespace std;             // Використання стандартного простору імен

class kooClass {// Оголошення класового типу
public:
	int x, y, z; // Тривимірні координати
	kooClass(int a, int b, int c) { x = a; y = b; z = c; }
};

// Відображення тривимірних координат x, y, z
// Перевизначений оператор виведення даних для класу kooClass
ostream &operator<<(ostream &stream, kooClass obj)
{
	stream << obj.x << ", ";
	stream << obj.y << ", ";
	stream << obj.z << endl;

	return stream; // Повертає посилання на параметр stream
}

int main()
{
	kooClass ObjA(1, 2, 3), ObjB(3, 4, 5), ObjC(5, 6, 7);

	// Перевизначений оператор виведення даних
	cout << ObjA << ObjB << ObjC;

	getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
1, 2, 3
3, 4, 5
5, 6, 7
```

Якщо видалити програмний код, який стосується звичайних операторів виведення даних конкретного класу kooClass, то залишиться "скелет", що відповідає будь-якій операторній функції виведення даних:

---

[1] У разі потреби другий параметр також може мати тип посилання на об'єкт.

```
ostream &operator<<(ostream &stream, class_type obj)
{
	// Код операторної функції виведення даних

	return stream; // Повертає посилання на параметр stream
}
```

Як було уже наголошено вище, для параметра obj дозволяється використовувати посилання для його передачі.

Загалом конкретні дії, які має виконувати операторна функція виведення даних, визначає програміст. Але, якщо у Вас виникає бажання дотримуватися професійного стилю програмування, то створена програмістом операторна функція виведення даних повинна тільки виводити інформацію. Проте, завжди не зайве впевнитися в тому, що вона повертає тільки параметр **stream**.

Перш ніж переходити до наступного розділу, подумайте, чому операторна функція виведення даних для класу kooClass не була запрограмована так:

```
// Версія обмеженого застосування (використанню не підлягає).
ostream &operator<<(ostream &stream, kooClass obj)
{
	cout << obj.x << ", ";
	cout << obj.y << ", ";
	cout << obj.z << endl;

	return stream; // Повертає посилання на параметр stream
}
```

Не важко здогадатися, що у такій версії операторної функції жорстко закодований потік **cout**. Це обмежує перелік ситуацій, в яких її можна використовувати.

Нео! хідноапам'ятати! *Перевизначений оператор "<<" можна застосувати для виведення будь-якого потоку. Потік даних, який використовується у "<<"-виразі, передається параметру* ***stream****.*

Отже, програміст повинен передавати операторній функції виведення даних потік, який коректно працює в усіх конкретних випадках. Тільки так можна створити досконалу операторну функцію виведення даних, яка підійде для використання в будь-яких виразах введення-виведення.

9.3.2. Використання функцій-"друзів" класу для перевизначення операторів виведення даних

У попередній програмі операторну функцію виведення даних не було визначено як член класу kooClass. Насправді ні будь-яка операторна функція виведення даних, ні функція їх введення не можуть бути членами класу. Справа тут полягає ось в чому. Якщо операторна функція є членом класу, то лівий операнд (що опосередковано передається за допомогою покажчика **this**) повинен бути об'єктом класу, який генерує звернення до цієї операторної функції. І це змінити не можна. Проте при перевизначенні операторів виведення даних лівий операнд повинен бути потоком, а правий – об'єктом класу, дані якого підлягають виведенню. Отже,

перевизначені оператори виведення даних не можуть бути функціями-членами класу.

У зв'язку з тим, що операторні функції виведення даних не можуть бути членами класу, для якого вони визначаються, то виникає серйозне запитання: як перевизначений оператор виведення даних може отримати доступ до закритих членів класу? У попередній програмі (див. код програми 9.1) змінні x, y і z були визначені як відкриті, тому оператор виведення даних без перешкод міг отримати до них доступ. Водночас закриття даних – важлива частина об'єктно-орієнтованого програмування, отже, вимагати, щоб усі дані були відкритими, просто нелогічно. Проте існує просте вирішення цього питання – оператор виведення даних можна зробити "другом" класу. Якщо функція є "другом" деякого класу, то вона отримує легальний доступ до його **private**-даних. Оголошення "другом" класу операторної функції виведення даних продемонструємо на прикладі класу kooClass.

Код програми 9.2. Демонстрація механізму використання функцій-"друзів" класу для перевизначення оператора виведення даних

```
#include <iostream>             // Для потокового введення-виведення
using namespace std;            // Використання стандартного простору імен

class kooClass {            // Оголошення класового типу
	int x, y, z;            // Тривимірні координати (тепер це private-члени)
public:
	kooClass(int a, int b, int c) { x = a; y = b; z = c; }
	friend ostream &operator<<(ostream &stream, kooClass obj);
};

// Відображення тривимірних координат x, y, z
// Перевизначений оператор виведення даних для класу kooClass
ostream &operator<<(ostream &stream, kooClass obj)
{
	stream << obj.x << ", ";
	stream << obj.y << ", ";
	stream << obj.z << endl;

	return stream; // Повертає посилання на параметр stream
}

int main()
{
	kooClass ObjA(1, 2, 3), ObjB(3, 4, 5), ObjC(5, 6, 7);

	// Перевизначений оператор виведення даних
	cout << ObjA << ObjB << ObjC;

	getch(); return 0;
}
```

Зверніть увагу на те, що змінні x, y і z у цій версії коду програми є закритими у класі kooClass, проте операторна функція виведення даних звертається до них

безпосередньо. У наведеному прикладі якраз і виявляється велика перевага "друзів" класу: оголошуючи операторні функції введення та виведення даних "друзями" класу, для якого вони визначаються, ми тим самим підтримуємо *механізм інкапсуляції* об'єктно-орієнтованого програмування.

9.3.3. Створення перевизначених операторів введення даних

Для перевизначення операторів введення даних використовують аналогічний підхід, який було застосовано при перевизначенні операторів виведення даних. Наприклад, наведений нижче перевизначений оператор введення даних забезпечує введення тривимірних координат. Зверніть увагу на те, що він також виводить відповідне повідомлення для користувача.

```
// Прийняття тривимірних координат x, y, z
// Перевизначений оператор введення даних для класу kooClass
istream &operator>>(istream &stream, kooClass &obj)
{
	cout << "Введіть координати x, y i z: ";

	// Перевизначений оператор введення даних
	stream >> obj.x >> obj.y >> obj.z;
	return stream; // Повертає посилання на параметр stream
}
```

Перевизначений оператор введення даних повинен повертати посилання на об'єкт типу **istream**. Окрім цього, перший параметр повинен бути посиланням на об'єкт типу **istream**. Цей тип належить потоку, що вказується зліва від оператора ">>". Другий параметр є посиланням на об'єкт, який приймає значення, що вводяться. Оскільки другий параметр – посилання на об'єкт, то його можна модифікувати при введенні інформації.

Загальний формат перевизначеного оператора введення даних має такий вигляд:

```
istream &operator>>(istream &stream, objectType &obj)
{
	// код операторної функції введення даних
	turn stream; // Повертає посилання на параметр stream
}
```

Особливості використання операторних функцій введення та виведення даних для об'єктів типу kooClass продемонстровано в такому коді програми.

Код програми 9.3. Демонстрація механізму перевизначення операторів введення/виведення потокової інформації

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен

class kooClass {// Оголошення класового типу
	int x, y, z; // Тривимірні координати
```

```
public:
        kooClass(int a, int b, int c) { x = a; y = b; z = c; }
        friend ostream &operator<<(ostream &stream, kooClass obj);
        friend istream &operator>>(istream &stream, kooClass &obj);
};

// Відображення тривимірних координат x, y, z
// Перевизначений оператор виведення даних для класу kooClass
ostream &operator<<(ostream &stream, kooClass obj)
{
        stream << obj.x << ", ";
        stream << obj.y << ", ";
        stream << obj.z << endl;

        return stream; // Повертає посилання на параметр stream
}

// Прийняття тривимірних координат x, y, z
// Перевизначений оператор введення даних для класу kooClass
istream &operator>>(istream &stream, kooClass &obj)
{
        cout << "Введіть координати x, y і z: ";

        // Перевизначений оператор введення даних
        stream >> obj.x >> obj.y >> obj.z;
        return stream;    // Повертає посилання на параметр stream
}

int main()
{
        kooClass ObjA(1, 2, 3);

        cout << ObjA;     // Перевизначений оператор виведення даних

        cin >> ObjA;      // Перевизначений оператор введення даних
        cout << ObjA;     // Перевизначений оператор виведення даних
        getch(); return 0;
}
```

Ось як виглядає результат виконання цієї програми:

```
1, 2, 3
Введіть координати x, y і z: 5 6 7
5, 6, 7
```

Подібно до операторних функцій виведення даних, операторні функції їх введення не можуть бути членами класу, для оброблення даних якого вони призначені. Вони можуть бути тільки "друзями" цього класу або просто незалежними операторними функціями.

За винятком того, що операторна функція введення даних повертає посилання на об'єкт типу **istream**, тіло цієї функції може містити все те, що програміст вважає за потрібне у неї помістити. Але логічніше використовувати оператори вве-

дення все ж таки за прямим призначенням, тобто для виконання операцій введення даних.

9.3.4. Порівняння С- і С++-систем введення-виведення

Як Вам уже відомо, попередниця мови програмування С++, мова С оснащена однією з найгнучкіших (серед структурованих мов) і водночас дуже могутньою системою введення-виведення[1]. Однак виникає логічне запитання: чому ж тоді у мові програмування С++ визначається власна система введення-виведення, якщо в ній продубльовано велику частину того, що міститься у мові С (маємо на увазі потужний набір С-функцій введення-виведення)? Відповісти на це запитання неважко. Йдеться про те, що С-система введення-виведення не забезпечує ніякої підтримки для об'єктів, які визначає користувач. Наприклад, якщо створити таку структуру

```
struct myStruct {// Оголошення типу структури
    int count;
    char str[80];
    double balance;
} oust;,
```

то наявну у мові С систему введення-виведення неможливо налаштувати так, щоб вона могла виконувати операції введення-виведення даних безпосередньо над об'єктами типу myStruct. Але, оскільки центром об'єктно-орієнтованого програмування є саме об'єкти, то виникає потреба у тому, щоб у мові програмування С++ функціонувала така система введення-виведення, яку можна було б динамічно "навчати" поводженню з будь-якими об'єктами, що створюються програмістом. Саме тому для мови програмування С++ і було винайдено нову об'єктно-орієнтовану систему введення-виведення. Як уже зазначалося вище, С++-підхід до введення-виведення даних дає змогу перевизначати оператори "<<" і ">>" так, що вони можуть працювати з класами, які створюють програмісти.

Вартоа' нати! *Оскільки мова програмування С++ є надбудовою мови С, то увесь вміст засобів С-системи введення-виведення включено у мову С++[2]. Тому у процесі перекладу С-програм на мову С++ програмісту не потрібно підряд змінювати усі настанови введення-виведення даних. Працюючі С-настанови скомпілюються і успішно працюватимуть у новому С++-середовищі. Просто програміст повинен врахувати те, що стара С-система введення-виведення не володіє об'єктно-орієнтованими можливостями.*

9.4. Форматне введення-виведення даних

Дотепер під час введення або виведення інформації в наведених вище прикладах програм діяли параметри форматування, які за замовчуванням використовує

---

[1] Не буде перебільшенням наголосити на тому, що серед усіх відомих структурованих мов С-система введення-виведення не має подібних до себе.

[2] Див. додаток В, у якому представлено огляд С-орієнтованих функцій введення-виведення.

С++-система введення-виведення. Але програміст може сам керувати форматом представлення даних, причому двома способами. *Перший спосіб* передбачає використання функцій-членів класу **ios**, а другий – функцій спеціального типу, що іменуються *маніпуляторами* (manipulator). У цьому підрозділі розглянемо основні можливості форматування даних, починаючи з функцій-членів класу **ios**, і завершуючи створенням власних маніпуляторних функцій.

9.4.1. Форматування даних з використанням функцій-членів класу ios

У С++-системі введення-виведення кожен потік пов'язаний з набором опцій, які керують процесом форматування даних. У класі **ios** оголошують перерахунок **fmtflags**, у якому визначено стандартні значення опцій форматування[1] (табл. 9.2).

Табл. 9.2. Стандартні С++-значення опцій форматування

adjustfield	**floatfield**	**right**	**skipws**
basefield	**hex**	**scientific**	**unitbuf**
boolalpha	**internal**	**showbase**	**uppercase**
dec	**left**	**showpoint**	
fixed	**oct**	**showpos**	

Ці значення використовують для встановлення або скидання опцій форматування за допомогою таких функцій: **setf**() і **unsetf**(). При використанні старого компілятора може так статися, що він не зможе визначити тип перерахунку **fmtflags**. У цьому випадку опції форматування кодуватимуться як цілочисельні **long**-значення.

1. Якщо опція **skipws** встановлена, то при потоковому введенні даних провідні "пропускні" символи, або символи пропуску (тобто пропуски, символи табуляції та нового рядка), відкидаються. Якщо ж опція **skipws** скинута, то пропускні символи не відкидаються.
2. Якщо встановлено опцію **left**, то дані, що виводяться, вирівнюються лівим краєм, а якщо встановлено опцію **right** – то правим краєм. Якщо встановлено опцію **internal**, то числове значення доповнюється пропусками, якими заповнюється поле між ним і знаком числа або символом основи системи числення. Якщо жодну з цих опцій не встановлено, то результат вирівнюється правим краєм за замовчуванням.
3. За замовчуванням числові значення виводяться у десятковій системі числення. Проте основу системи числення можна змінити. Встановлення опції **oct** приведе до виведення результату у вісімковому представленні, а встановлення опції **hex** – в шістнадцятковому. Щоб під час відображення результату повернутися до десяткової системи числення, достатньо встановити опцію **dec**.
4. Встановлення опції **showbase** приводить до відображення позначення основи системи числення, у якій представляються числові значення. Наприклад, якщо використовується шістнадцяткове представлення, то значення 1F буде відображено як 0x1F.
5. За замовчуванням при використанні експоненціального представлення чисел відображається рядковий варіант букви "e". Окрім цього, під час відображення

[1] Точніше, ці значення визначено у класі `ios_base`, який, як згадувалося вище, є базовим для класу `ios`.

шістнадцяткового значення використовують також рядкову букву "x". Після встановлення опції **uppercase** відображається прописний варіант цих символів.

6. Встановлення опції **showpos** викликає відображення провідного знаку "плюс" перед позитивними значеннями.
7. Встановлення опції **showpoint** приводить до відображення десяткової крапки і хвостових нулів для всіх чисел з плинною крапкою – потрібні вони чи ні.
8. Після встановлення опції **scientific** числові значення з плинною крапкою відображаються в експоненціальному представленні. Якщо встановлено опцію **fixed**, то дійсні значення відображаються у звичайному представленні. Якщо не встановлено жодну з цих опцій, то компілятор сам вибирає відповідний формат їх представлення.
9. При встановленому опції **unitbuf** вміст буфера скидається на диск після кожної операції виведення даних.
10. Якщо встановлено опцію **boolalpha**, то значення булевого типу можна вводити або виводити, використовуючи ключові слова **true** і **false**.

Оскільки часто доводиться звертатися до полів **oct**, **dec** і **hex**, то на них допускається колективне посилання **ios::basefield**. Аналогічно поля **left**, **right** і **internal** можна узагальнено назвати **ios**::**adjustfield**. Нарешті, поля **scientific** і **fixed** можна назвати **ios**::**floatfield**.

Для встановлення будь-якої опції використовується функція **setf**(), яка є членом класу **ios**. Ось як виглядає її формат:

fmtflags setf(**fmtflags** ***flags***);

Ця функція повертає значення попередніх установок опцій форматування і встановлює їх відповідно до значення, які задаються параметром ***flags***. Наприклад, щоб встановити опцію **showbase**, можна використовувати таку настанову:

stream.**setf**(**ios**::**showbase**);

У цьому записі елемент ***stream*** означає потік, параметри форматування якого Ви хочете змінити. Зверніть увагу на використання префікса **ios**:: для уточнення належності параметра **showbase**. Оскільки параметр **showbase** представляє собою перерахункову константу, що визначається у класі **ios**, то під час звернення до неї необхідно вказувати ім'я класу **ios**. Цей принцип стосується всіх опцій форматування. У наведеному нижче коді програми функцію **setf**() використовують для встановлення опцій **showpos** і **scientific**.

```
#include <iostream>            // Для потокового введення-виведення
using namespace std;           // Використання стандартного простору імен

int main()
{
    cout.setf(ios::showpos); // Відображення знаку "+" перед позитивним значенням
    cout.setf(ios::scientific); // Відображення чисел у експоненціальному вигляді
    cout << 123 << " " << 123.23 << " ";

    getch(); return 0;
}
```

Ось як виглядають результати виконання цієї програми:

```
+123 +1.232300e+002
```

За допомогою операції АБО можна встановити відразу декілька потрібних опцій форматування в одному виклику функції **setf**(). Наприклад, наведену вище програму можна скоротити, об'єднавши за допомогою АБО опції **scientific** і **showpos**, оскільки у цьому випадку буде виконуватися тільки одне звернення до функції **setf**():

```
// Відображення чисел у експоненціальному вигляді або знаку "+" перед позитивним значенням
cout.setf(ios::scientific || ios::showpos);
```

Щоб скинути опцію, потрібно використовувати функцію **unsetf**(), прототип якої має такий вигляд:

```
void unsetf(fmtflags flags);
```

У цьому випадку будуть очищені опції, що задаються параметром **flags**. При цьому всі інші опції залишаються у попередньому стані.

Для того, щоби дізнатися про поточні установки опцій форматування, потрібно скористатися функцією **flags**(), прототип якої має такий вигляд:

```
fmtflags flags();
```

Ця функція повертає поточні значення опцій форматування для потоку, що викликається.

При використанні наведеного нижче формату виклику функції **flags**() встановлюються значення опцій форматування відповідно до вмісту параметра **flags** і повертаються їх попередні значення:

```
fmtflags flags(fmtflags flags);
```

Щоб зрозуміти, як працюють функції **flags**() і **unsetf**(), розглянемо детально наведену нижче програму. Вона містить функцію **showflags**(), яка відображає поточний стан опцій форматування потоку інформації.

Код програми 9.4. Демонстрація механізму використання функцій flags() і unsetf()

```
#include <iostream>          // Для потокового введення-виведення
using namespace std;         // Використання стандартного простору імен

void showflags(ios::fmtflags f); // Відображення поточного стану опцій

int main()
{
	ios::fmtflags f;    // Оголошення параметру для поточного стану опцій

	f = cout.flags();   // Отримання поточного стану опцій
	showflags(f);              // Відображення поточного стану опцій
	cout.setf(ios::showpos); // Відображення знаку "+" перед позитивним значенням
	cout.setf(ios::scientific); // Відображення чисел у експоненціальному вигляді

	f = cout.flags();   // Отримання поточного стану опцій
	showflags(f);       // Відображення поточного стану опцій

	// Скидання опції, що відображає числа в експоненціальному вигляді
	cout.unsetf(ios::scientific);
```

```
        f = cout.flags();   // Отримання поточного стану опцій
        showflags(f);       // Відображення поточного стану опцій

        getch(); return 0;
}

void showflags(ios::fmtflags f) // Відображення поточного стану опцій
{
        long i;
        for(i=0x4000; i; i = i >> 1)
        if(i & f) cout << "1 ";
        else cout << "0 ";
        cout << endl;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати[1]:

```
0 0 0 0 0 1 0 0 0 0 0 0 0 0 1
0 0 1 0 0 1 0 0 0 1 0 0 0 0 1
0 0 0 0 0 1 0 0 0 1 0 0 0 0 1
```

У наведеному вище коді програми зверніть увагу на те, що тип **fmtflags** вказано з префіксом **ios**::. Йдеться про те, що тип **fmtflags** визначено у класі **ios**. У загальному випадку при використанні імені типу або перерахованої константи, визначеної у певному класі, необхідно вказувати відповідне ім'я разом з іменем класу.

9.4.2. Встановлення ширини поля, точності значення та символів заповнення

Окрім опцій форматування можна також встановлювати ширину поля, символ заповнення і кількість цифр після десяткової крапки (точність). Для цього достатньо використовувати такі функції:

```
streamsize width(streamsize len);

char fill(char ch);

streamsize precision(streamsize num);
```

1. Функція **width**() повертає поточну ширину поля і встановлює нову, що дорівнює значенню параметра ***len***. Ширина поля, яка встановлюється за замовчуванням, визначається кількістю символів, необхідних для зберігання даних у кожному конкретному випадку.
2. Функція **fill**() повертає поточний символ заповнення (за замовчуванням використовується пропуск) і встановлює як новий поточний символ заповнення значень, які задаються параметром *ch*. Цей символ використовують для доповнення результату символами, яких не вистачає для досягнення заданої ширини поля.
3. Функція **precision**() повертає поточну кількість цифр, що відображаються після десяткової крапки, і встановлює нове поточне значення точності, що дорівнює значенню параметра num[1]. Тип **streamsize** визначено як цілочисельний.

[1] Між цими і Вашими результатами можливо виникне розбіжність, спричинена використанням різних компіляторів.

Розглянемо код програми, яка демонструє використання цих трьох функцій.

Код програми 9.5. Демонстрація механізму використання функцій встановлення ширини поля, точності та символів заповнення

```
#include <iostream>             // Для потокового введення-виведення
using namespace std;            // Використання стандартного простору імен

int main()
{
      cout.setf(ios::showpos); // Відображення знаку "+" перед позитивним значенням
      cout.setf(ios::scientific); // Відображення чисел у експоненціальному вигляді
      cout << 123 << " " << 123.23 << endl;

      cout.precision(2);        // Дві цифри після десяткової крапки.
      cout.width(10);           // Все поле складається з 10 символів.
      cout << 123 << " ";
      cout.width(10);           // Встановлення ширини поля в 10 символів.
      cout << 123.23 << endl;

      cout.fill('#');           // Як заповнювач використаємо символ "#"
      cout.width(10);           // Встановлення ширини поля в 10 символів.
      cout << 123 << " ";
      cout.width(10)            // Встановлення ширини поля в 10 символів.
      cout << 123.23
      getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
+123 +1.232300e+002
  +123 +1.23e+002
######+123 +1.23e+002
```

У деяких реалізаціях С++-компілятора необхідно встановлювати значення ширини поля перед виконанням кожної операції виведення даних. Тому функція **width**() у попередній програмі викликалася декілька разів.

У С++-системі введення-виведення визначено і перевизначено різні версії функцій **width**(), **precision**() і **fill**(), які не змінюють поточні значення відповідних параметрів форматування і використовуються тільки для їх отримання. Ось як виглядають їх прототипи:

- **streamsize width()** – повертає поточну ширину поля;
- **char fill()** – повертає поточний символ заповнення;
- **streamsize precision()** – повертає поточну кількість цифр, що відображаються після десяткової крапки.

9.4.3. Використання маніпуляторів введення-виведення даних

У С++-системі введення-виведення передбачено і другий спосіб зміни параметрів форматування даних, пов'язаних з потоком інформації. Він реалізується за

[1] За замовчуванням після десяткової крапки відображається шість цифр.

допомогою спеціальних функцій – так званих *маніпуляторів,* які можна помістити у вираз введення-виведення. Призначення та функції стандартних С++маніпуляторів введення-виведення описано в табл. 9.3.

Табл. 9.3. Стандартні С++-маніпулятори введення-виведення

Маніпулятор	Призначення	Функція
boolalpha	Встановлює опцію **boolalpha**	Введення-виведення
dec	Встановлює опцію **dec**	Введення-виведення
endl	Виводить символ нового рядка і "скидає" потік, тобто переписує вміст буфера, пов'язаного з потоком, на відповідний пристрій	Виведення
ends	Вставляє у потік нульовий символ ('\0')	Виведення
fixed	Встановлює опцію **fixed**	Виведення
flush	"Скидає" потік	Виведення
hex	Встановлює опцію **hex**	Введення-виведення
internal	Встановлює опцію **internal**	Виведення
left	Встановлює опцію **left**	Виведення
noboolalpha	Онулює опція **boolalpha**	Введення-виведення
noshowbase	Онулює опція **showbase**	Виведення
noshowpoint	Онулює опція **showpoint**	Виведення
noshowpos	Онулює опція **showpos**	Виведення
noskipws	Онулює опція **skipws**	Введення
nounitbuf	Онулює опція **unitbuf**	Виведення
nouppercase	Онулює опція **uppercase**	Виведення
oct	Встановлює опцію **oct**	Введення-виведення
resetiosflags(**fmtflags** f)	Онулює опції, що задаються у параметрі *f*	Введення-виведення
right	Встановлює опцію **right**	Виведення
scientific	Встановлює опцію **scientific**	Виведення
setbase(**int** baseClass)	Встановлює основу системи числення, що дорівнює значенню baseClass	Виведення
setfill(**int** *ch*)	Встановлює символ-заповнювач, що дорівнює значенню параметра *ch*	Виведення
setiosflags(**fmtflags** f)	Встановлює опції, що задаються у параметрі *f*	Введення-виведення
setprecision(**int** p)	Встановлює кількість цифр точності (після десяткової крапки), що дорівнює значенню параметра *p*	Виведення
setw(**int** w)	Встановлює ширину поля, що дорівнює значенню параметра *w*	Виведення
showbase	Встановлює опцію **showbase**	Виведення
showpoint	Встановлює опцію **showpoint**	Виведення
showpos	Встановлює опцію **showpos**	Виведення
skipws	Встановлює опцію **skipws**	Введення
unitbuf	Встановлює опцію **unitbuf**	Виведення
uppercase	Встановлює опцію **uppercase**	Виведення
ws	Пропускає провідні "пропускні" символи	Введення

У процесі використання маніпуляторів, які приймають аргументи, необхідно приєднати до програми заголовок **<iomanip>**.

Будь-який маніпулятор використовується як частина виразу введення-виведення. Нижче наведено код програми, у якій показано, як за допомогою маніпуляторів можна керувати процесом форматування даних, які виводяться.

Код програми 9.6. Демонстрація механізму використання маніпуляторів для керування процесом форматування даних, які виводяться

```
#include <iostream>          // Для потокового введення-виведення
#include <iomanip>           // Використання маніпуляторів введення-виведення
using namespace std;         // Використання стандартного простору імен

int main()
{
    cout << setprecision(2) << 1000.243 << endl;
    cout << setw(20) << "Всім привіт! ";

    getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
1e+003
 Всім привіт!
```

Зверніть увагу на те, як використовуються маніпулятори у послідовності операцій введення-виведення даних. Окрім цього, зверніть увагу також на те, коли маніпулятор викликається без аргументів (як, наприклад, маніпулятор **endl** у наведеному вище коді програми), то його ім'я вказується без пари круглих дужок.

У наведеному нижче коді програми використано маніпулятор **setiosflags**() для встановлення опцій **scientific** і **showpos**.

Код програми 9.7. Демонстрація механізму використання маніпулятора setiosflags() для встановлення опцій scientific і showpos

```
#include <iostream>          // Для потокового введення-виведення
#include <iomanip>           // Використання маніпуляторів введення-виведення
using namespace std;         // Використання стандартного простору імен

int main()
{
    // Відображення знаку "+" перед позитивним значенням числа
    cout << setiosflags(ios::showpos);

    // Відображення чисел у експоненціальному вигляді
    cout << setiosflags(ios::scientific);
    cout << 123 << " " << 123.23;

    getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
+123 +1.232300e+002
```

А у цьому коді програми продемонстровано механізм використання маніпулятора **ws**, який пропускає провідні "пропускні" символи при введенні рядка в масив str.

Код програми 9.8. Демонстрація механізму використання маніпулятора ws, який пропускає провідні "пропускні" символи

```
#include <iostream>            // Для потокового введення-виведення
using namespace std;           // Використання стандартного простору імен

int main()
{
    char str[80];
    cin >> ws >> str;
    cout << str;

    getch(); return 0;
}
```

9.4.4. Створення власних маніпуляторних функцій

Програміст може самостійно створювати власні маніпуляторні функції. Існує два типи маніпуляторних функцій – ті, що ***приймають*** (параметризовані) і ***не приймають аргументи*** (непараметризовані). Для створення параметризованих маніпуляторів використовуються підходи, вивчення яких виходить за рамки цього навчального посібника. Проте створення непараметризованих маніпуляторів не викликає особливих труднощів.

Всі непараметризовані маніпуляторні функції виведення даних мають таку структуру:

```
ostream &manip_name(ostream &stream)
{
    // код маніпуляторної функції

    return stream; // Повертає посилання на параметр stream
}
```

У цьому записі елемент *manip_name* вказує ім'я маніпулятора. Важливо розуміти особливості використання не параметризованих маніпуляторів. Незважаючи на те, що маніпулятор приймає як єдиний аргумент покажчик на потік, який він обробляє, проте при використанні маніпулятора в остаточному виразі введення-виведення аргументи не вказуються взагалі.

У наведеному нижче коді програми створюється маніпулятор **setup**(), який встановлює опцію вирівнювання лівим краєм, ширину поля в 10 символів і задає як заповнювальний символ знак долара.

Код програми 9.9. Демонстрація механізму створення маніпулятора setup()

```
#include <iostream>            // Для потокового введення-виведення
#include <iomanip>             // Використання маніпуляторів введення-виведення
using namespace std;           // Використання стандартного простору імен
```

```
ostream &setup(ostream &stream)
{
    stream.setf(ios::left); // Вирівнювання лівим краєм

    /* Встановлює ширину поля в 10 символів і задає як заповнювальний
     символ знак долара */
    stream << setw(10) << setfill('$');

    return stream; // Повертає посилання на параметр stream
}

int main()
{
    cout << 10 << " " << setup << 10;
    getch(); return 0;
}
```

Створення власних маніпуляторів є корисним з двох причин:

- *по-перше*, іноді виникає потреба виконувати операції введення-виведення даних з використанням пристрою, до якого жоден з вбудованих маніпуляторів не застосовується (наприклад, плоттер). У цьому випадку створення власних маніпуляторів зробить виведення даних на цей пристрій зручнішим;
- *по-друге*, може так статися, що у створеній Вами програмі певна послідовність настанов повторюється декілька разів. І тоді ці операції можна об'єднати в один маніпулятор так, як це показано у попередній програмі.

Всі непараметризовані маніпуляторні функції введення даних мають таку структуру:

```
istream &manip_name(istream &stream)
{
    // код маніпуляторної функції

    return stream; // Повертає посилання на параметр stream
}
```

Наприклад, у наведеному нижче коді програми створюється маніпулятор **prompt**(). Він налаштовує вхідний потік на прийняття даних у шістнадцятковому представленні та виводить для користувача відповідне повідомлення.

Код програми 9.10. Демонстрація механізму створення маніпулятора prompt()

```
#include <iostream>         // Для потокового введення-виведення
#include <iomanip>          // Використання маніпуляторів введення-виведення
using namespace std;        // Використання стандартного простору імен

istream &prompt(istream &stream)
{
    cin >> hex;
    cout << "Введіть число в шістнадцятковому форматі: ";

    return stream; // Повертає посилання на параметр stream
}
```

```
int main()
{
    int c;
    cin >> prompt >> c;
    cout << c; // Виведення числа в шістнадцятковому форматі

    getch(); return 0;
}
```

Нео! хідноапам'ятати! Надзвичайно важливо, щоб створений програмістом маніпулятор повертав потоковий об'єкт (елемент **stream***). Інакше цей маніпулятор не можна буде використовувати у складеному виразі введення або виведення.*

9.5. Організація файлового введення-виведення даних

У C++-системі введення-виведення також передбачено засоби для виконання відповідних операцій з використанням файлів. Файлові операції введення-виведення даних можна реалізувати після внесення у програму заголовка **<fstream>**, у якому визначено всі необхідні для цього класи і значення.

9.5.1. Відкриття та закриття файлу

У мові програмування C++ файл відкривається шляхом зв'язування його з потоком. Як уже зазначалося вище, існують потоки трьох типів: введення, виведення і введення-виведення. Щоб відкрити вхідний потік, необхідно оголосити потік класу **ifstream**. Для відкриття вихідного потоку потрібно оголосити потік класу **ofstream**. Потік, який передбачається використовувати для операцій як введення, так і виведення, повинен бути оголошений як потік класу **fstream**. Наприклад, у процесі виконання такого фрагмента коду програми буде створено вхідний і вихідний потоки, а також потік, що дає змогу виконувати операції в обох напрямах:

```
ifstream in;     // Вхідний потік
ofstream out;    // Вихідний потік
fstream both;    // Потік введення-виведення
```

Щоб відкрити файл, використовується функція ***open()****.*

Створивши потік, його потрібно пов'язати з файлом. Це можна зробити за допомогою функції **open**(), причому у кожному з трьох потокових класів є своя функція-член **open**(). Їх прототипи мають такий вигляд:

```
void ifstream::open(const char *filename, ios::openmode mode = ios::in);

void ofstream::open(const char * filename, ios::openmode mode = ios::out | ios::trunc);

void fstream::open(const char *filename, ios::openmode mode = ios::in | ios::out);
```

У цих записах елемент *filename* означає ім'я файлу, яке може містити специфікатор шляху, що вказує доступ до нього. Елемент *mode* називається специфікато-

ром режиму, який визначає спосіб відкриття файлу. Він повинен приймати одне або декілька значень перерахунку **openmode**, який визначено у класі **ios**:

- **ios::app** – приєднує до кінця файлу усі дані, що виводяться;
- **ios::ate** – пошук потрібних даних починатиметься з кінця файлу;
- **ios::binary** – відкриває файл у двійковому режимі;
- **ios::in –** забезпечує відкриття файлу для введення даних;
- **ios::out –** забезпечує відкриття файлу для виведення даних
- **ios::trunc** – призводить до руйнування вмісту файлу.

Декілька значень перерахунку **openmode** можна об'єднувати за допомогою логічного додавання (АБО).

Вартоа' нати!а*Параметр mode для функції **fstream::open()** за замовчуванням може не дорівнювати значенню **in | out** (це залежить від використовуваного компілятора). Тому у разі потреби цей параметр Вам доведеться задавати безпосередньо.*

Нео! хідноапам'ятати! *За замовчуванням усі файли відкриваються в текстовому режимі.*

1. Внесення значення **ios::app** у параметр *mode* забезпечить приєднання до кінця файлу всіх даних, які виводяться. Це значення можна застосовувати тільки до файлів, відкритих для виведення даних.
2. Під час відкриття файлу з використанням значення **ios::ate** пошук потрібних даних починатиметься з кінця файлу. Незважаючи на це, операції введення-виведення даних можуть, як і раніше, виконуватися по всьому файлу.
3. Значення **ios::binary** дає змогу відкрити файл у двійковому режимі. Як уже зазначалося вище, в текстовому режимі можуть відбуватися деякі перетворення символів (наприклад, послідовність, що складається з символів повернення каретки і переходу на новий рядок, може бути перетворена у символ нового рядка). Під час відкриття файлу у двійковому режимі ніякого перетворення символів не здійснюється.
4. Значення **ios::in** вказує на те, що цей файл відкривається для введення даних, а значення **ios::out** забезпечує відкриття файлу для виведення даних.
5. Використання значення **ios::trunc** призводить до руйнування вмісту файлу, ім'я якого збігається з параметром *filename*, а сам цей файл урізається до нульової довжини. При створенні вихідного потоку типу **ofstream** будь-який наявний файл з іменем *filename* автоматично урізається до нульової довжини.

Вартоа' нати! *Будь-який файл, що містить форматний текст або ще необроблені дані, можна відкрити як у двійковому, так і в текстовому режимі. Єдина відмінність між цими режимами полягає у перетворенні (чи ні) символів.*

У процесі виконання такий фрагмент коду програми відкриває звичайний вихідний файл:

```
ofstream myStream;
out.open("тест");
```

Оскільки параметру *mode* функції **open**() за замовчуванням встановлює значення, яке дорівнює відповідному типу потоку, який відкривається, то у наведеному вище прикладі взагалі немає потреби задавати його значення.

У результаті невдалого виконання функції **open**() не відкритому потоку при використанні булевого виразу встановлюється значення, яке дорівнює ФАЛЬШ. Такий механізм може слугувати для підтвердження успішного відкриття файлу, наприклад, за допомогою такої **if**-настанови:

```
if(!myStream) {
    cout << "Не вдається відкрити файл" << endl;
    // оброблення помилки
}
```

Нео! хідноапам'ятати!а*Перед спробою отримання доступу до даних файла, необхідно завжди перевіряти результат виклику функції* ***open()***.

Рекомендується також перевірити факт успішного відкриття файлу за допомогою функції **is_open**(), яка є членом класів **fstream**, **ifstream** і **ofstream**. Її прототип має такий вигляд:

```
bool is_open();
```

Ця функція повертає значення ІСТИНА, якщо потік пов'язаний з відкритим файлом, і ФАЛЬШ – в іншому випадку. Наприклад, використовуючи такий фрагмент коду програми, можна дізнатися про те, чи відкрито у цей момент потоковий об'єкт myStream:

```
if(!myStream.is_open()) {
    cout << "Файл не відкрито" << endl;
    //. . .
}
```

Хоча цілком коректно використовувати функцію **open**() для відкриття файлу, проте здебільшого це робиться по-іншому, оскільки класи **ifstream**, **ofstream** і **fstream** містять конструктори, які автоматично відкривають заданий файл. Параметри у цих конструкторів і їх значення (що діють за замовчуванням) збігаються з параметрами і відповідними значеннями функції **open**(). Тому найчастіше файл відкривається так, як це показано в наведеному нижче прикладі:

```
ifstream myStream("myFile"); // Файл відкривається для введення
```

Якщо з якоїсь причини файл відкрити неможливо, то потоковій змінній, що пов'язується з цим файлом, встановлюється значення, яке дорівнює ФАЛЬШ.

Щоб закрити файл, використовується функція-член **close**(). Наприклад, щоб закрити файл, який пов'язано з потоковим об'єктом myStream, потрібно використати таку настанову:

```
myStream.close();
```

Функція **close**() не має параметрів і не повертає ніякого значення.

9.5.2. Зчитування та запис текстових файлів

Найпростіше зчитувати дані з текстового файлу або записувати їх до нього за допомогою операторів "<<" і ">>". Наприклад, у наведеному нижче коді програми здійснюється спочатку запис у файл test цілого числа, потім значення з плинною крапкою і на завершення текстового рядка.

Код програми 9.11. Демонстрація механізму запису даних у файл

```
#include <iostream>          // Для потокового введення-виведення
#include <fstream>           // Для роботи з файлами
using namespace std;         // Використання стандартного простору імен

int main()
{
	ofstream out("test");
	if(!out) {
		cout << "Не вдається відкрити файл" << endl;
		return 1;
	}

	out << 10 << " " << 123.23 << endl;
	out << "Це короткий текстовий файл.";

	out.close();
	getch(); return 0;
}
```

Наведений нижче код програми зчитує ціле число, **float**-значення, символ і рядок з файлу, створеного у процесі виконання попередньою програмою:

Код програми 9.12. Демонстрація механізму зчитування даних з файлу

```
#include <iostream>          // Для потокового введення-виведення
#include <fstream>           // Для роботи з файлами
using namespace std;         // Використання стандартного простору імен

int main()
{
	char ch;
	int c;
	float f;
	char str[80];

	ifstream in("test");
	if(!in) {
		cout << "Не вдається відкрити файл" << endl;
		return 1;
	}
	in >> c;
	in >> f;
	in >> ch;
	in >> str;

	cout << c << " " << f << " " << ch << endl;
	cout << str;

	in.close();
	getch(); return 0;
}
```

__Вартоа' нати!__ При використанні перевизначеного оператора ">>" для зчитування даних з текстових файлів відбувається перетворення деяких символів. Наприклад, "пропускні" символи опускаються. Якщо необхідно запобігти будь-яким перетворенням символів, то потрібно відкрити файл у двійковому режимі доступу до його даних.

__Нео! хідноапам'ятати!__ При використанні перевизначеного оператора ">>" для зчитування рядка введення даних припиняється внаслідок виявлення першого "пропускного" символу.

9.5.3. Неформатне введення-виведення даних у двійковому режимі

Форматні текстові файли (подібні до тих, які використовувалися у попередніх прикладах) корисні в багатьох ситуаціях, але вони не мають гнучкості неформатних двійкових файлів. Тому мова програмування C++ підтримує ряд функцій файлового введення-виведення у двійковому режимі, які можуть виконувати операції без форматування даних.

Для виконання двійкових операцій файлового введення-виведення необхідно відкрити файл з використанням специфікатора режиму **ios**::**binary**. Необхідно відзначити, що функції, які використовуються для оброблення неформатних файлів, можуть також виконувати дії з файлами, відкритими в текстовому режимі доступу, але при цьому використовується перетворення символів, яке зводить нанівець основну мету виконання двійкових файлових операцій.

*Функція **get**() зчитує символ з файлу, а функція **put**() записує символ у файл.*

У загальному випадку існує два способи запису неформатних двійкових даних у файл і зчитування їх з файлу. *Перший спосіб* полягає у використанні функції-члена класу **put**() (для запису байта у файл) і функції-члена класу **get**() (для зчитування байта з файлу). *Другий спосіб* передбачає застосування "блокових" C++-функцій введення-виведення **read**() і **write**(). Розглянемо кожен спосіб окремо.

Функції **get**() і **put**() мають багато форматів, але найчастіше використовуються такі їх версії:

```
istream &get(char &ch);
ostream &put(char ch);
```

1. Функція **get**() зчитує один символ з відповідного потоку і поміщає його значення у змінну *ch*. Вона повертає посилання на потік, що пов'язаний із заздалегідь відкритим файлом. Досягнувши кінця цього файлу, значення посилання дорівнюватиме нулю.
2. Функція **put**() записує символ *ch* у потік і повертає посилання на цей потік.

У процесі виконання наведеної нижче програми на екран буде виведено вміст будь-якого заданого файлу. Тут використовується функція **get**().

Код програми 9.13. Демонстрація механізму відображення вмісту файлу за допомогою функції get()

```
#include <iostream>          // Для потокового введення-виведення
#include <fstream>           // Для роботи з файлами
```

```
using namespace std;            // Використання стандартного простору імен

int main(int argc, char *argv[])
{
        char ch;

        if(argc !=2) {
                cout << "Застосування: ім'я_програми <ім'я_файлу>" << endl;
                return 1;
        }

        ifstream in(argv[1], ios::in | ios::binary);
        if(!in) {
                cout << "Не вдається відкрити файл" << endl;
                return 1;
        }

        while(in) { // Досягши кінця файлу потоковий об'єкт in прийме значення false.
                in.get(ch);
                if(in) cout << ch;
        }

        in.close();
        getch(); return 0;
}
```

Досягши кінця файлу, потоковий об'єкт **in** прийме значення ФАЛЬШ, яке зупинить виконання циклу **while**. Проте існує дещо коротший варіант коду програми організації циклу, призначеного для зчитування і відображення вмісту файлу:

```
while(in.get(ch)) cout << ch;
```

Цей варіант організації циклу також має право на існування, оскільки функція **get**() повертає потоковий об'єкт **in**, який, досягши кінця файлу, прийме значення **false**. У наведеному нижче коді програми для запису рядка у файл використовується функція **put**().

Код програми 9.14. Демонстрація механізму використання функції put() для запису рядка у файл

```
#include <iostream>             // Для потокового введення-виведення
#include <fstream>              // Для роботи з файлами
using namespace std;            // Використання стандартного простору імен

int main()
{
        char *p = "Всім привіт!";

        ofstream out("test", ios::out | ios::binary);

        if(!out) {
                cout << "Не вдається відкрити файл" << endl;
                return 1;
        }
```

```
        while(*p) out.put(*p++);

        out.close();
        getch(); return 0;
}
```

9.5.4. Зчитування та записування у файл блоків даних

Для зчитування і записування у файл блоків двійкових даних використовуються функції-члени **read**() і **write**(). Їх прототипи мають такий вигляд:

```
istream &read(char *buf, streamsize num);
ostream &write(const char *buf, int streamsize num);
```

1. Функція **read**() зчитує *num* байт даних з пов'язаного з файлом потоку і поміщає їх у буфер, який адресується покажчиком *buf*.
2. Функція **write**() записує *num* байт даних у пов'язаний з файлом потік з буфера, який адресується покажчиком *buf*.

Як уже зазначалося вище, тип **streamsize** визначається як певний різновид цілочисельного типу. Він дає змогу зберігати найбільшу кількість байтів, яку можна передана у процесі будь-якої операції введення-виведення даних.

У процесі виконання наведеної нижче програми спочатку у файл записується масив цілих чисел, а потім його значення зчитується з файлу.

Код програми 9.15. Демонстрація механізму використання функцій read() і write()

```
#include <iostream>          // Для потокового введення-виведення
#include <fstream>           // Для роботи з файлами
using namespace std;         // Використання стандартного простору імен

int main()
{
        int n[5] = {1, 2, 3, 4, 5};
        register int i;

        ofstream out("test", ios::out | ios::binary);
        if(!out) {
                cout << "Не вдається відкрити файл" << endl;
                return 1;
        }

        out.write((char *) &n, sizeof n);
        out.close();

        for(i=0; i<5; i++) n[i] = 0; // Очищує масив

        ifstream in("test", ios::in | ios::binary);
        if(!in) {
                cout << "Не вдається відкрити файл" << endl;
                return 1;
        }
```

```
        in.read((char *) &n, sizeof n);

        for(i=0; i<5; i++) // Відображаємо значення, зчитані з файлу.
        cout << n[i] << " ";

        in.close();
        getch(); return 0;
}
```

Зверніть увагу на те, що в настановах звернення до функцій **read**() і **write**() виконуються операції приведення типу, які є обов'язковими при використанні буфера, що визначається у вигляді не символьного масиву.

---

*Функція **gcount()** повертає кількість символів, зчитаних у процесі виконання останньої операції введення даних.*

Якщо кінець файлу досягнуто ще до того моменту, як було зчитано *num* символів, то функція **read**() просто припинить своє виконання, а буфер міститиме стільки символів, скільки вдалося зчитати до цього моменту. Точну кількість зчитаних символів можна дізнатися за допомогою ще однієї функції-члена класу **gcount**(), яка має такий прототип:

```
streamsize gcount();
```

Функція **gcount**() повертає кількість символів, зчитаних у процесі виконання останньої операції введення даних.

9.5.5. Використання функції eof() для виявлення кінця файлу

Виявити кінець файлу можна за допомогою функції-члена класу **eof**(), яка має такий прототип:

```
bool eof();
```

Ця функція повертає значення **true** у випадку досягнення кінця файлу; інакше вона повертає значення **false**.

У наведеному нижче коді програми для виведення на екран вмісту файлу використовується функція **eof**().

Код програми 9.16. Демонстрація механізму виявлення кінця файлу за допомогою функції eof()

```
#include <iostream>              // Для потокового введення-виведення
#include <fstream>               // Для роботи з файлами
using namespace std;             // Використання стандартного простору імен

int main(int argc, char *argv[])
{
        char ch;
        if(argc != 2) {
                cout << "Застосування: ім'я_програми <ім'я_файлу>" << endl;
                return 1;
        }

        ifstream in(argv[1], ios::in | ios::binary);
```

```
	if(!in) {
		cout << "Не вдається відкрити файл" << endl;
		return 1;
	}

	while(!in.eof()) { // Використання функції eof()
		in.get(ch);
		if(!in.eof()) cout << ch;
	}

	in.close();
	getch(); return 0;
}
```

9.5.6. Застосування С++ файлової системи для порівняння файлів

Наведений нижче код програми ілюструє потужність і простоту застосування у мові програмування С++ файлової системи. Тут порівнюються два файли за допомогою функцій двійкового введення-виведення **read**(), **eof**() і **gcount**(). Програма спочатку відкриває порівнювані файли для виконання двійкових операцій (щоб не допустити перетворення символів). Потім з кожного файлу по черзі зчитуються блоки інформації у відповідні буфери і порівнюється їхній вміст. Оскільки об'єм зчитаних даних може бути меншим за розмір буфера, то у програмі використовується функція **gcount**(), яка точно визначає кількість зчитаних у буфер байтів. Неважко переконатися у тому, що при використанні файлових С++-функцій для виконання цих операцій була потрібна зовсім невелика за розміром програма.

Код програми 9.17. Демонстрація механізму застосування файлової системи для порівняння файлів

```
#include <iostream>		// Для потокового введення-виведення
#include <fstream>		// Для роботи з файлами
using namespace std;		// Використання стандартного простору імен

int main(int argc, char *argv[])
{
	register int i;

	unsigned char buf1[1024], buf2[1024];
	if(argc !=3) {
		cout << "Застосування: ім'я_програми <ім'я_файла1> " << "<ім'я_файла2>" << endl;
		return 1;
	}

	ifstream f1(argv[1], ios::in | ios::binary);
	if(!f1) {
		cout << "Не вдається відкрити перший файл" << endl;
		return 1;
	}

	ifstream f2(argv[2], ios::in | ios::binary);
```

```
    if(!f2) {
        cout << "Не вдається відкрити другий файл" << endl;
        return 1;
    }

    cout << "Порівняння файлів" << endl;
    do {
        f1.read((char *) buf1, sizeof buf1);
        f2.read((char *) buf2, sizeof buf2);
        if(f1.gcount() != f2.gcount()) {
            cout << "Файли мають різні розміри" << endl;
            f1.close();
            f2.close();
            getch(); return 0;
        }

        // Порівняння вмісту буферів.
        for(i=0; i<f1.gcount(); i++)
            if(buf1[i] != buf2[i]) {
                cout << "Файли різні" << endl;
                f1.close();
                f2.close();
                getch(); return 0;
            }
    } while(!f1.eof() && !f2.eof());

    cout << "Файли однакові" << endl;

    f1.close();
    f2.close();
    getch(); return 0;
}
```

Спробуйте провести такий експеримент. Розмір буфера у цьому коді програми жорстко встановлено таким, що дорівнює 1024. Як вправу замініть це значення за допомогою **const**-змінної та випробуйте інші розміри буферів. Визначте оптимальний розмір буфера для свого операційного середовища.

9.5.7. Використання інших функцій для двійкового введення-виведення даних

Крім наведеного вище формату використання функції **get**() існують і інші її перевизначені версії. Наведемо прототипи для трьох з них, які використовуються найчастіше:

```
istream &get(char *buf, streamsize num);

istream &get(char *buf, streamsize num, char delim);

int get();
```

1. Перша версія функції **get**() дає змогу зчитувати символи і заносити їх у масив, який задається параметром *buf,* доти, доки не буде зчитано *num*-1 символів, або

не трапиться символ нового рядка, або не буде досягнуто кінець файлу. Після завершення роботи функції **get**() масив, який адресується покажчиком *buf*, матиме завершальний нуль-символ. Символ нового рядка, якщо такий виявиться у вхідному потоці, не вилучається. Він залишається там доти, доки не виконається наступна операція введення-виведення.

2. Друга версія функції **get**() призначена для зчитування символів і занесення їх у масив, який адресується покажчиком *buf*, доти, доки не буде зчитано *num*-1 символів, або не виявиться символ, який задається параметром *delim,* або не буде досягнуто кінець файлу. Після завершення роботи функції **get**() масив, який адресується покажчиком *buf*, матиме завершальний нуль-символ. Символ-роздільник (заданий параметром *delim*) , якщо такий виявиться у вхідному потоці, не вилучається. Він залишається там доти, доки не виконається наступна операція введення-виведення.
3. Третя перевизначена версія функції **get**() повертає з потоку наступний символ. Він міститься в молодшому байті значення, що повертається функцією. Отже, значення, що повертається функцією **get**(), можна присвоїти змінній типу **char**. Досягши кінця файлу, ця функція повертає значення **EOF**, яке визначено у заголовку <**iostream**>.

Функцію **get**() корисно використовувати для зчитування рядків, що містять пропуски. Як уже зазначалося вище, якщо для зчитування рядка використовують оператор ">>", то процес введення даних зупиняється внаслідок виявлення першого ж пропускного символу. Це робить оператор ">>" не придатним для зчитування рядків, що містять пропуски. Але цю проблему, як це показано в такому коді програми, можна обійти за допомогою функції **get**(*buf*, *num*).

Код програми 9.18. Демонстрація механізму використання функції get() для зчитування рядків, що містять пропуски

```
#include <iostream>            // Для потокового введення-виведення
#include <fstream>             // Для роботи з файлами
using namespace std;           // Використання стандартного простору імен

int main()
{
    char str[80];

    cout << "Введіть ім'я: ";
    cin.get(str, 79);

    cout << str << endl;
    getch(); return 0;
}
```

У цьому коді програми як символ-роздільник під час зчитування рядка за допомогою функції **get**() використовується символ нового рядка. Це робить поведінку функції **get**() багато в чому схожою з поведінкою стандартної функції **gets**(). Проте перевага функції **get**() полягає у тому, що вона дає змогу запобігти можливому виходу за межі масиву, який приймає символи, які вводяться користувачем, оскільки у програмі оголошено максимальну кількість зчитаних символів. Це робить функцію **get**() набагато безпечнішою за функцію **gets**().

Розглянемо ще одну функцію, яка дає змогу вводити дані. Йдеться про функцію **getline**(), яка є членом кожного потокового класу, призначеного для введення інформації. Ось як виглядають прототипи версій цієї функції:

istream &getline(**char** **buf,* **streamsize** *num*);

istream &getline(**char** **buf,* **streamsize** *num*, **char** *delim*);

1. При використанні першої версії функції **getline**()символи зчитуються і заносяться у масив, який адресується покажчиком *buf*, доти, доки не буде зчитано *num*-1 символів, або не трапиться символ нового рядка, або не буде досягнуто кінець файлу. Після завершення роботи функції **getline**() масив, який адресується покажчиком *buf*, матиме завершальний нуль-символ. Символ нового рядка, якщо такий виявиться у вхідному потоці, при цьому вилучається, але не заноситься у масив *buf*.
2. Друга версія функції **getline**()призначена для зчитування символів і занесення їх у масив, який адресується покажчиком *buf*, доти, доки не буде зчитано *num*-1 символів, або не виявиться символ, який задається параметром *delim*, або не буде досягнуто кінець файлу. Після завершення роботи функції **getline**() масив, який адресується покажчиком *buf*, матиме завершальний нуль-символ. Символ-роздільник (який задаються параметром *delim*) , якщо такий виявиться у вхідному потоці, вилучається, але не заноситься у масив *buf*.

Як бачите, ці дві версії функцій **getline**() практично ідентичні версіям **get**(*buf*, *num*) і **get**(*buf*, *num*, *delim*) функції **get**(). Обидві зчитують символи з вхідного потоку і заносять їх у масив, який адресується покажчиком *buf*, доти, доки не буде зчитано *num*-1 символів, або не виявиться символ, який задається параметром *delim*. Відмінність між функціями **get**() і **getline**() полягає у тому, що функція **getline**() зчитує і видаляє символ-роздільник з вхідного потоку, а функція **get**() цього не робить.

Наступний символ з вхідного потоку можна отримати і не видаляти його з потоку за допомогою функції **peek**(). Її прототип має такий вигляд:

int peek();

Функція **peek**() повертає наступний символ потоку, або значення **EOF**, якщо досягнуто кінець файлу. Зчитаний символ повертається в молодшому байті значення, що повертається функцією. Тому значення, що повертається функцією **peek**(), можна присвоїти змінній типу **char**.

Останній символ, що зчитується з потоку, можна повернути у потік, використовуючи функцію **putback**(). Її прототип має такий вигляд:

istream &putback(**char** *ch*);

У цьому записі параметр *ch* містить символ, що зчитується з потоку останнім.

Під час виведення даних за допомогою функції **flush**()не відбувається негайного їх запису на фізичний пристрій, що у даний момент пов'язаний з потоком. Інформація, яка підлягає виведенню, спочатку нагромаджується у внутрішньому буфері доти, доки він цілком не заповниться. І тільки тоді його вміст переписується на диск. Проте існує можливість негайного перезапису на диск даних, які зберігається в буфері функції **flush**(), не чекаючи його повного заповнення. Цей засіб полягає у виклику функції **flush**(). Її прототип має такий вигляд:

ostream &flush();

До викликів функції **flush**() необхідно вдаватися у випадку, якщо програма призначена для роботи в несприятливих середовищах (для яких характерні часті відключення, наприклад, електрики).

9.5.8. Перевірка статусу введення-виведення даних

С++-система введення-виведення підтримує статусну інформацію про результати виконання кожної операції введення-виведення даних. Поточний статус потоку введення-виведення описується в об'єкті типу **iostate**, який є перерахунком (воно визначене у класі **ios**), що містить такі члени:

Ім'я	Значення
ios::goodbit	Помилок немає
ios::eofbit	1 внаслідок виявлення кінця файлу; 0 – в іншому випадку
ios::failbit	1 під час виникнення поправної помилки введення-виведення; 0 – в іншому випадку
ios::badbit	1 під час виникнення непоправної помилки введення-виведення; 0 – в іншому випадку

Статусну інформацію про результат виконання операцій введення-виведення даних можна отримувати двома способами. По-перше, можна викликати функцію **rdstate**(), яка є членом класу **ios**. Вона має такий прототип:

iostate rdstate();

Функція **rdstate**() повертає поточний статус опцій помилок. Неважко здогадатися, що, судячи з наведеного вище переліку опцій, функція **rdstate**() поверне значення **goodbit** за відсутності будь-яких помилок. У іншому випадку вона повертає відповідний код помилки.

По-друге, про наявність помилки можна дізнатися за допомогою однієї або декількох наступних функцій-членів класу **ios**:

- функція **bool bad**(); повертає значення ІСТИНА, якщо внаслідок виконання операції введення-виведення даних було встановлено опцію badbit;
- функцію **bool eof**(); розглянуто вище;
- функція **bool fail**(); повертає значення ІСТИНА, якщо внаслідок виконання операції введення-виведення даних було встановлено опцію failbit;
- функція **bool good**(); повертає значення ІСТИНА, якщо у процесі виконання операції введення-виведення даних помилок не відбулося.

У інших випадках ці функції повертають значення ФАЛЬШ.

Якщо у процесі виконання операції введення-виведення даних Ви допустилися помилки, то, можливо, перш ніж продовжувати виконання програми, є сенс скинути опції помилок. Для цього використовують функцію **clear**() (член класу **ios**), прототип якої має такий вигляд:

void clear(**iostate** *flags* = **ios**::**goodbit**);

Якщо параметр *flags* дорівнює значенню **goodbit**(воно встановлюється за замовчуванням), то всі опції помилок очищаються. Інакше опції встановлюються відповідно до заданого Вами значення.

Перш ніж переходити до наступного підрозділу, варто випробувати функції, які повідомляють дані про поточний стан опцій помилок, внісши у попередні приклади кодів програм опції перевірки помилок.

9.6. Використання файлів довільного доступу

Дотепер використовувалися файли, доступ до вмісту яких було організовано строго послідовно, байт за байтом. Але у мові програмування С++ також можна отримувати доступ до файла у довільному порядку.

9.6.1. Функції довільного доступу

Для довільного доступу до даних файла необхідно використовувати функції **seekg**() і **seekp**(). Їх прототипи мають такий вигляд:

istream &**seekg**(**off_type** *offset,* seekdir *origin*);

ostream &**seekp**(**off_type** *offset,* seekdir *origin*);

Використовуваний тут цілочисельний тип **off_type** (його визначено у класі **ios**) дає змогу зберігати найбільше допустиме значення, яке може мати параметр *offset.* Тип seekdir визначено як перерахунок, який має такі значення.

Значення	Опис
ios::beg	Початок файлу
ios::cur	Поточна позиція файлу
ios::end	Кінець файлу

У С++-системі введення-виведення передбачено можливість керування двома покажчиками, пов'язаними з файлом. Ці так звані **get**- і **put**-покажчики визначають, у якому місці файлу повинна виконатися наступна операція введення та виведення відповідно. При кожному виконанні операції введення або виведення відповідний покажчик автоматично переміщається у вказану позицію. Використовуючи функції **seekg**() і **seekp**(), можна отримувати доступ до файла у довільному порядку.

1. Функція **seekg**() переміщає поточний **get**-покажчик відповідного файлу на *offset* байт відносно позицій, які задаються параметром *origin*.
2. Функція **seekp**() переміщає поточний **put**-покажчик відповідного файлу на *offset* байт відносно позицій, які задаються параметром *origin*.

Вартоа' нати! *У загальному випадку довільний доступ для операцій введення-виведення даних повинен виконуватися тільки для файлів, відкритих у двійковому режимі. Перетворення символів, які можуть відбуватися в текстових файлах, можуть призвести до того, що запитувана позиція файлу не відповідатиме його реальному вмісту.*

*Функція **seekg()** переміщає покажчик, що "відповідає" за введення даних, а функція **seekp()** – покажчик, що "відповідає" за виведення.*

Поточну позицію кожного файлового покажчика можна визначити за допомогою таких двох функцій:

- **pos_type tellg**(); – повертає поточну позицію **get**-покажчика;
- **pos_type tellp**(); – повертає поточну позицію **put**-покажчика.

У цих записах використовується тип **pos_type** (він визначений у класі **ios**), що дає змогу зберігати найбільше значення, яке може повернути будь-яка з цих функцій.

Існують перевизначені версії функцій **seekg**() і **seekp**(), які переміщають файлові покажчики у позиції файлу, що задаються значеннями, які повертаються функціями **tellg**() і **tellp**()відповідно. Ось як виглядають їх прототипи:

```
istream &seekg(pos_type position);
ostream &seekp(pos_type position);
```

9.6.2. Приклади використання довільного доступу до вмісту файлу

У наведеному нижче коді програми продемонстровано механізм використання функції **seekp**(). Вона дає змогу задати ім'я файлу у командному рядку, а за ним – конкретний байт, який потрібно у ньому змінити. Програма потім записує у вказану позицію символ "X". Зверніть увагу на те, що оброблюваний файл повинен бути відкритим для виконання операцій зчитування-запису.

Код програми 9.19. Демонстрація механізму використання функції seekp() для довільного доступу до вмісту файлу

```
#include <iostream>           // Для потокового введення-виведення
#include <fstream>            // Для роботи з файлами
#include <cstdlib>            // Для використання бібліотечних функцій
using namespace std;          // Використання стандартного простору імен

int main(int argc, char *argv[])
{
    if(argc !=3) {
        cout << "Застосування: ім'я_програми " << "<ім'я_файлу> <байт>" << endl;
        return 1;
    }

    fstream out(argv[1], ios::in | ios::out | ios::binary);
    if(!out) {
        cout << "Не вдається відкрити файл" << endl;
        return 1;
    }
    out.seekp(atoi(argv[2]), ios::beg);

    out.put('X');
    out.close();
    getch(); return 0;
}
```

У наведеному нижче коді програми показано приклад використання функції **seekg**(). Вона відображає вміст файлу, починаючи з позиції, що задається у командному рядку.

Код програми 9.20. Демонстрація механізму відображення вмісту файлу із заданої позиції

```
#include <iostream>           // Для потокового введення-виведення
#include <fstream>            // Для роботи з файлами
#include <cstdlib>            // Для використання бібліотечних функцій
using namespace std;          // Використання стандартного простору імен
```

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

        if(argc !=3) {
                cout << "Застосування: ім'я_програми " << "<ім'я_файлу> <стартова_позиція>" << endl;
                return 1;
        }

        ifstream in(argv[1], ios::in | ios::binary);
        if(!in) {
                cout << "Не вдається відкрити файл" << endl;
                return 1;
        }

        in.seekg(atoi(argv[2]), ios::beg);

        while(in.get(ch)) cout << ch;

        getch(); return 0;
}
```

9.7. Використання перевизначених операторів введення-виведення даних при роботі з файлами

Вище Ви вже дізналися, як перевизначати оператори введення-виведення для власних класів, а також як створювати власні маніпулятори. У наведених вище прикладах програм виконувалися тільки операції консольного введення-виведення. Але, оскільки всі C++-потоки однакові, то одну і ту саму операторну функцію виведення даних, наприклад, можна використовувати для виведення інформації як на екран, так і у файл, не вносячи при цьому ніяких істотних змін. Саме у цьому і полягають основні переваги C++-системи введення-виведення.

У наведеному нижче коді програми використано перевизначений (для класу kooClass) оператор виведення даних для записування значень поточних координат у файл threed.

Код програми 9.21. Демонстрація механізму використання перевизначеного оператора введення-виведення даних для запису об'єктів класу у файл

```
#include <iostream>             // Для потокового введення-виведення
#include <fstream>              // Для роботи з файлами
using namespace std;            // Використання стандартного простору імен
class kooClass {// Оголошення класового типу
        int x, y, z; // Тривимірні координати; вони тепер закриті
public:
        kooClass(int a, int b, int c) { x = a; y = b; z = c; }
        friend ostream &operator<<(ostream &stream, kooClass obj);
};
```

```
// Відображення тривимірних координат x, y, z
// Перевизначений оператор виведення даних для класу kooClass
ostream &operator<<(ostream &stream, kooClass obj)
{
	stream << obj.x << ", ";
	stream << obj.y << ", ";
	stream << obj.z << endl;

	return stream; // Повертає посилання на параметр stream
}

int main()
{
	kooClass ObjA(1, 2, 3), ObjB(3, 4, 5), ObjC(5, 6, 7);
	ofstream out("threed");

	if(!out) {
		cout << "Не вдається відкрити файл.";
		return 1;
	}

	// Перевизначений оператор виведення даних
	out << ObjA << ObjB << ObjC;

	out.close();

	getch(); return 0;
}
```

Якщо порівняти цю версію операторної функції виведення даних для класу kooClass з тією, що була представлена на початку цього розділу, то можна переконатися у тому, що для "налаштування" її на роботу з дисковими файлами ніяких змін вносити не довелося. Якщо оператори введення-виведення визначені коректно, то вони успішно працюватимуть з будь-яким потоком.

Вартоа' нати!а *Перш ніж переходити до наступного розділу, не пошкодуйте часу і попрацюйте з С++-функціями введення-виведення. Створіть власний клас, а потім визначте для нього оператори введення-виведення. А ще створіть власні маніпулятори.*

Розділ 10. ДИНАМІЧНА ІДЕНТИФІКАЦІЯ ТИПІВ І ОПЕРАТОРИ ПРИВЕДЕННЯ ТИПУ

У цьому розділі розглянемо два засоби мови програмування С++, які підтримують сучасне ООП: динамічна ідентифікація типів (run-time type identification – RTTI) і набір додаткових операторів приведення типу. Жоден з цих засобів не був частиною оригінальної специфікації мови програмування С++, але обидва вони були додані у нову версію мови С++ з метою посилення підтримки поліморфізму тривалості виконання. Під RTTI розуміють можливість проведення ідентифікації типу об'єкта у процесі виконання програми. Оператори приведення типу, що розглядаються у цьому розділі, пропонують програмісту безпечніші способи виконання цієї операції. Як буде показано далі, один з них – **dynamic_cast** безпосередньо пов'язаний з RTTI-ідентифікацією, тому оператори приведення типу і RTTI є сенс розглядати в одному розділі.

10.1. Динамічна ідентифікація типів

З динамічною ідентифікацією типів (RTTI) більшість традиційних програмістів незнайомі, оскільки цей засіб відсутній у такій неполіморфній мові, як С. У неполіморфних мовах просто немає потреби в отриманні інформації про тип у процесі виконання програми, оскільки тип кожного об'єкта відомий при компілюванні (тобто ще під час написання програми). Але в такій поліморфній мові, як С++, можливі ситуації, в яких тип об'єкта невідомий у період компілювання, оскільки точна природа цього об'єкта не буде визначена доти, доки програма на почне виконуватися. Як уже зазначалося вище, мова програмування С++ реалізує поліморфізм за допомогою використання ієрархії класів, віртуальних функцій і покажчиків на об'єкти базових класів. Покажчик на базовий клас можна використовувати для посилання на члени як цього базового класу, так і на члени будь-якого об'єкта, виведеного з нього. Отже, не завжди наперед відомо, на об'єкт якого типу посилатиметься покажчик на базовий клас у довільний момент часу. Це з'ясується тільки у процесі виконання програми – при використанні одного із засобів динамічної ідентифікації типів.

10.1.1. Отримання типу об'єкта у процесі виконання програми

Для цього необхідно приєднати до програми заголовок <**typeinfo**>. Найпоширеніший формат використання оператора **typeid** такий:

typeid(*object*)

У цьому записі елемент *object* означає об'єкт, тип якого потрібно отримати. Можна робити запити не тільки про вбудований тип, але і про тип класу, створеного програмістом. Оператор **typeid** повертає посилання на об'єкт типу **type_info**, який описує тип об'єкта *object*.

У класі **type_info** визначено такі **public**-члени:

bool operator==(**const type_info** &*ob*);

bool operator!=(**const type_info** &*ob*);

bool before(**const type_info** &*ob*);

const char *name();

Перевизначені оператори "==" і "!=" слугують для порівняння типів. Функція **before**() повертає значення **true**, якщо викликаючий об'єкт у порядку зіставлення знаходиться перед об'єктом (елементом *ob*), що використовується як параметр[1]. Функція **name**() повертає покажчик на ім'я типу.

Розглянемо простий приклад використання оператора **typeid**.

Код програми 10.1. Демонстрація механізму використання оператора typeid

```
#include <iostream>            // Для потокового введення-виведення
#include <typeinfo>            // Для динамічної ідентифікації типів
using namespace std;           // Використання стандартного простору імен

class myClass { // Оголошення класового типу
	//. . .
};

int main()
{
	int c, d;
	float f;
	myClass obj;
	cout << "Тип змінної c: " << typeid(c).name() << endl;
	cout << "Тип змінної f: " << typeid(f).name() << endl;
	cout << "Тип змінної obj: " << typeid(obj).name() << endl << endl;

	if(typeid(c) == typeid(d))
		cout << "Типи змінних c та d однакові" << endl;

	if(typeid(c) != typeid(f))
		cout << "Типи змінних c та f неоднакові" << endl;
	getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Тип змінної c: int
Тип змінної f: float
Тип змінної obj: class myClass

Типи змінних c та d однакові.
Типи змінних c та f неоднакові.
```

[1] Ця функція призначена в основному для внутрішнього використання. Значення результату, який вона повертає, не має нічого спільного із успадкуванням або ієрархією класів.

Якщо оператор **typeid** застосовується до покажчика на поліморфний базовий клас (пригадайте: поліморфний клас – це клас, який містить хоч би одну віртуальну функцію), він автоматично повертає тип реального об'єкта, на який той вказує: будь то об'єкт базового класу або об'єкт класу, виведеного з базового.

Отже, оператор **typeid** можна використовувати для динамічного визначення типу об'єкта, який адресується покажчиком на базовий клас. Застосування цієї можливості продемонстровано в такому коді програми.

Код програми 10.2. Демонстрація механізму застосування оператора typeid до ієрархії поліморфних класів

```
#include <iostream>              // Для потокового введення-виведення
#include <typeinfo>              // Для динамічної ідентифікації типів
using namespace std;             // Використання стандартного простору імен

// Оголошення базового класу
class Base {
	virtual void Fun() {}; // Робимо клас Base поліморфним
	//. . .
};

class DerivedA: public Base {
	//. . .
};

class DerivedB: public Base {
	//. . .
};

int main()
{
	Base *p, baseob;
	DerivedA ObjA;	// Створення об'єкта класу
	DerivedB ObjB;	// Створення об'єкта класу

	p = &baseob;
	cout << "Змінна р вказує на об'єкт типу ";
	cout << typeid(*p).name() << endl;

	p = &ObjA;
	cout << "Змінна р вказує на об'єкт типу ";
	cout << typeid(*p).name() << endl;

	p = &ObjB;
	cout << "Змінна р вказує на об'єкт типу ";
	cout << typeid(*p).name() << endl;

	getch(); return 0;
}
```

Ось як виглядають результати виконання цієї програми:

```
Змінна р вказує на об'єкт типу Base
Змінна р вказує на об'єкт типу DerivedA
Змінна р вказує на об'єкт типу DerivedB
```

Якщо оператор **typeid** застосовується до покажчика на базовий клас поліморфного типу, то тип об'єкта, який реально адресується, як підтверджують ці результати, буде визначений у процесі виконання програми.

У всіх випадках застосування оператора **typeid** до покажчика на неполіморфну ієрархію класів буде отримано покажчик на базовий тип, тобто те, на що цей покажчик реально вказує, визначити не можна. Як експеримент спробуйте перетворити на коментар віртуальну функцію Fun() у класі Base і подивіться на результат. Ви побачите, що тип кожного об'єкта після внесення у програму цієї зміни буде визначений як Base, оскільки саме цей тип має покажчик p.

Оскільки оператор **typeid** зазвичай застосовується до перейменованого покажчика (тобто до покажчика, до якого вже застосовано оператор "*"), то для оброблення ситуації, коли цей перейменований покажчик виявиться нульовим, створено спеціальний виняток. У цьому випадку оператор **typeid** генерує виняток типу **bad_typeid**.

Посилання на об'єкти ієрархії поліморфних класів працюють подібно до покажчиків. Якщо оператор **typeid** застосовується до посилання на поліморфний клас, то він повертає тип об'єкта, на який вона реально посилається, і це може бути об'єкт не базового, а похідного типу. Описаний засіб найчастіше використовують при передачі об'єктів функціям за посиланням. Наприклад, у наведеному нижче коді програми функція WhatType() оголошує посилальний параметр на об'єкти типу Base. Це означає, що функції WhatType() можна передавати посилання на об'єкти типу Base або посилання на об'єкти будь-яких класів, похідних від Base. Оператор **typeid**, що застосовується до такого параметра, поверне реальний тип об'єкта, який передається функції.

Код програми 10.3. Демонстрація механізму застосування оператора typeid до посилального параметра

```
#include <iostream>             // Для потокового введення-виведення
#include <typeinfo>             // Для динамічної ідентифікації типів
using namespace std;            // Використання стандартного простору імен

// Оголошення базового класу
class Base {
	virtual void Fun() {; // робимо клас Base поліморфним
	//. . .
};

class DerivedA: public Base {
	//. . .
};

class DerivedB: public Base {
	//. . .
};

// Демонструємо застосування оператора typeid до посилального параметра.
void WhatType(Base &obj)
{
	cout << "Параметр obj посилається на об'єкт типу ";
```

```
        cout << typeid(obj).name() << endl;
}

int main()
{
        int c;

        Base baseob;
        DerivedA ObjA;    // Створення об'єкта класу
        DerivedB ObjB;    // Створення об'єкта класу

        WhatType(baseob);
        WhatType(ObjA);
        WhatType(ObjB);

        getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Параметр obj посилається на об'єкт типу Base
Параметр obj посилається на об'єкт типу DerivedA
Параметр obj посилається на об'єкт типу DerivedB
```

Існує ще одна версія застосування оператора **typeid**, яка як аргумент приймає ім'я типу. Формат його є таким:

typeid(*ім'я_типа*)

Наприклад, наступна настанова абсолютно допускається:

```
cout << typeid(int).name();
```

Призначення цієї версії оператора **typeid** – отримати об'єкт типу **type_info** (який описує заданий тип даних), щоб його можна було використовувати в настанові порівняння типів.

10.1.2. Приклад RTTI-застосування

У наведеному нижче коді програми показано, наскільки корисним може бути засіб динамічної ідентифікації типів (RTTI). Тут використовується модифікована версія ієрархії класів геометричних фігур з розд. 6 [9], який обчислює площу круга, трикутника і прямокутника. У наведеному нижче коді програми визначено функцію factory(), призначену для створення примірника круга, трикутника або прямокутника. Ця функція повертає покажчик на створений об'єкт[1]. Конкретний тип створюваного об'єкта визначається внаслідок звернення до функції **rand**() C++-генератора випадкових чисел. Таким чином, ми не можемо знати наперед, об'єкт якого типу буде згенеровано. Програма створює десять об'єктів і підраховує кількість створених фігур кожного типу. Оскільки під час виклику функції factory() може бути згенерована фігура будь-якого типу, то для визначення типу реально створеного об'єкта у програмі використовують оператор **typeid**.

---

[1] Функцію, яка генерує об'єкти, іноді називають *генератором об'єктів*.

Код програми 10.4. Демонстрація ефективності використання засобу динамічної ідентифікації типів

```
#include <iostream>            // Для потокового введення-виведення
#include <cstdlib>             // Для використання бібліотечних функцій
using namespace std;           // Використання стандартного простору імен

class figure {
protected:
	double x, y;
public:
	figure(double _x, double _y) { x = _x; y = _y; }
	virtual double area() = 0;
};

class triangle: public figure {
public:
	triangle(double _x, double _y): figure(_x, _y) {}
	double area() { return x * 0.5 * y; }
};

class rectangle: public figure {
public:
	rectangle(double _x, double _y): figure(_x, _y) {}
	double area() { return x * y; }
};

class circle: public figure {
public:
	circle(double _x, double _y=0): figure(_x, _y) {}
	double area() { return 3.14 * x * x; }
};

// Генератор об'єктів класу figure.
figure *factory() {
	switch(rand() % 3) {
		case 0: return new circle(10.0);
		case 1: return new triangle(10.1, 5.3);
		case 2: return new rectangle(4.3, 5.7);
	}
	return 0;
}

int main()
{
	figure *p; // Покажчик на базовий клас

	int t = 0, r = 0, c = 0;

	// Генеруємо і підраховуємо об'єкти
	for(int i=0; i<10; i++) {
```

```
            p = factory(); // Генеруємо об'єкт
            cout << "Об'єкт має тип " << typeid(*p).name();
            cout << ". ";

            // Враховуємо цей об'єкт
            if(typeid(*p) == typeid(triangle)) t++;
            if(typeid(*p) == typeid(rectangle)) r++;
            if(typeid(*p) == typeid(circle)) c++;

            // Відображаємо площу фігури
            cout << "Площа дорівнює " << p->area() << endl;
        }

        cout << endl;
        cout << "Згенеровано такі об'єкти:" << endl;
        cout << " трикутників: " << t << endl;
        cout << " прямокутників: " << r << endl;
        cout << " кругів: " << c << endl;

        getch(); return 0;
}
```

Можливий результат виконання цієї програми такий:

```
Об'єкт має тип class rectangle. Площа дорівнює 24.51
Об'єкт має тип class rectangle. Площа дорівнює 24.51
Об'єкт має тип class triangle. Площа дорівнює 26.765
Об'єкт має тип class triangle. Площа дорівнює 26.765
Об'єкт має тип class rectangle. Площа дорівнює 24.51
Об'єкт має тип class triangle. Площа дорівнює 26.765
Об'єкт має тип class circle. Площа дорівнює 314
Об'єкт має тип class circle. Площа дорівнює 314
Об'єкт має тип class triangle. Площа дорівнює 26.765
Об'єкт має тип class rectangle. Площа дорівнює 24.51

Згенеровано такі об'єкти:
 трикутників: 4
 прямокутників: 4
 кругів: 2
```

10.1.3. Застосування оператора typeid до шаблонних класів

Оператор **typeid** можна застосувати і до шаблонних класів. Тип об'єкта, який є примірником шаблонного класу, визначається частково на підставі того, які саме дані використовуються для його узагальнених даних під час реалізації об'єкта. Таким чином, два примірники одного і того ж шаблонного класу, які створюються з використанням різних даних, мають різний тип. Розглянемо простий приклад.

Код програми 10.5. Демонстрація механізму застосування оператора typeid до шаблонних класів

```
#include <iostream>          // Для потокового введення-виведення
#include <cstdlib>           // Для використання бібліотечних функцій
using namespace std;         // Використання стандартного простору імен
```

```
template <class myClass> class myClass {      // Оголошення класового типу
        myClass a;
public:
        myClass(myClass c) { a = c; }
        //. . .
};

int main()
{
        myClass<int> ObjA(10), ObjB(9);
        myClass<double> ObjC(7.2);

        cout << "Об'єкт ObjA має тип ";
        cout << typeid(ObjA).name() << endl;

        cout << "Об'єкт ObjB має тип ";
        cout << typeid(ObjB).name() << endl;

        cout << "Об'єкт ObjC має тип ";
        cout << typeid(ObjC).name() << endl;

        cout << endl;

        if(typeid(ObjA) == typeid(ObjB))
                cout << "Об'єкти ObjA і ObjB мають однаковий тип" << endl;
        if(typeid(ObjA) == typeid(ObjC))
                cout << "Помилка" << endl;
        else
                cout << "Об'єкти ObjA і ObjC мають різні типи" << endl;


        getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Об'єкт ObjA має тип class myClass<int>
Об'єкт ObjB має тип class myClass<int>
Об'єкт ObjC має тип class myClass<double>

Об'єкти ObjA і ObjB мають однаковий тип.
Об'єкти ObjA і ObjC мають різні типи.
```

Як бачите, незважаючи на те, що два об'єкти є примірниками одного і того ж шаблонного класу, якщо їх дані, що параметризуються, не збігаються, то вони не є еквівалентними за типом. У цьому коді програми об'єкт ObjA має тип myClass<**int**>, а об'єкт ObjC – тип myClass<**double**>. Таким чином, це об'єкти різного типу.

Розглянемо ще один приклад застосування оператора **typeid** до шаблонних класів, а саме модифіковану версію програми визначення геометричних фігур з попереднього підрозділу. Цього разу клас figure ми зробили шаблонним. Прозора структура коду програми дасть змогу читачу без проблем розібратися з основним принципом її роботи.

Код програми 10.6. Демонстрація механізму застосування оператора typeid до шаблонної версії figure-ієрархії класів

```
#include <iostream>              // Для потокового введення-виведення
#include <cstdlib>               // Для використання бібліотечних функцій
using namespace std;             // Використання стандартного простору імен

template <class myClass> class figure {
protected:
	myClass x, y;
public:
	figure(myClass _x, myClass _y) { x = _x; y = _y; }
	virtual myClass area() = 0;
};

template <class myClass> class triangle: public figure<myClass> {
public:
	triangle(myClass _x, myClass _y): figure<myClass>(_x, _y) {}
	myClass area() { return x * 0.5 * y; }
};

template <class myClass> class rectangle: public figure<myClass> {
public:
	rectangle(myClass _x, myClass _y): figure<myClass>(_x, _y) {}
	myClass area() { return x * y; }
};

template <class myClass> class circle: public figure<myClass> {
public:
	circle(myClass _x, myClass _y=0): figure<myClass>(_x, _y) {}
	myClass area() { return 3.14 * x * x; }
};

// Генератор об'єктів, що утворюються з класу figure.
figure<double> *generator()
{
	switch(rand() % 3) {
		case 0: return new circle<double>(10.0);
		case 1: return new triangle<double>(10.1, 5.3);
		case 2: return new rectangle<double>(4.3, 5.7);
	}
	return;
}

int main()
{
	figure<double> *p;
	int t = 0, c = 0, r = 0;

	// Генеруємо і підраховуємо об'єкти
	for(int i=0; i<10; i++) {
```

```
                p = generator();
                cout << "Об'єкт має тип " << typeid(*p).name();
                cout << ". ";

                // Враховуємо об'єкт
                if(typeid(*p) == typeid(triangle<double>)) t++;
                if(typeid(*p) == typeid(rectangle<double>)) r++;
                if(typeid(*p) == typeid(circle<double>)) c++;

                cout << "Площа дорівнює " << p->area() << endl;
        }

        cout << endl;
        cout << "Згенеровано такі об'єкти:" << endl;
        cout << " трикутників: " << t << endl;
        cout << " прямокутників: " << r << endl;
        cout << " кругів: " << c << endl;

        getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати::

```
Об'єкт має тип class rectangle<double>. Площа дорівнює 24.51
Об'єкт має тип class rectangle<double>. Площа дорівнює 24.51
Об'єкт має тип class triangle<double>. Площа дорівнює 26.765
Об'єкт має тип class triangle<double>. Площа дорівнює 26.765
Об'єкт має тип class rectangle<double>. Площа дорівнює 24.51
Об'єкт має тип class triangle<double>. Площа дорівнює 26.765
Об'єкт має тип class circle<double>. Площа дорівнює 314
Об'єкт має тип class circle<double>. Площа дорівнює 314
Об'єкт має тип class triangle<double>. Площа дорівнює 26.765
Об'єкт має тип class rectangle<double>. Площа дорівнює 24.51

Згенеровано такі об'єкти:
 трикутників: 4
 прямокутників: 4
 кругів: 2
```

Динамічна ідентифікація типів використовується не у кожній програмі. Але під час роботи з поліморфними типами цей засіб дає змогу дізнатися про тип об'єкта, який обробляється в будь-який довільний момент часу.

10.2. Поняття про оператори приведення типів

У мові програмування C++ визначено п'ять операторів приведення типів. *Перший оператор* (його було описано вище у цьому навчальному посібнику), вживається у звичайному (традиційному) стилі, був із самого початку вбудований у мові C++. Інші чотири (**dynamic_cast**, **const_cast**, **reinterpret_cast** і **static_cast**) були додані у мову всього декілька років тому. Ці оператори надають додаткові "важелі керування" характером виконання операцій приведення типу. Розглянемо кожний з них зокрема.

10.2.1. Оператор приведення поліморфних типів dynamic_cast

Можливо, найважливішим з нових операторів є оператор динамічного приведення типів **dynamic_cast**. У процесі виконання програми він перевіряє обґрунтованість пропонованої операції. Якщо у момент його виклику задана операція виявляється неприпустимою, то приведення типів не здійснюється. Загальний формат застосування оператора **dynamic_cast** є таким:

dynamic_cast<*type*> (*expr*)

У цьому записі елемент *type* означає новий тип, який є метою виконання цієї операції, а елемент *expr* – вираз, який приводиться до цього нового типу. Тип *type* має бути представлений покажчиком або посиланням, а вираз *expr* повинен приводитися до покажчика або посилання. Таким чином, оператор **dynamic_cast** можна використовувати для перетворення покажчика одного типу у покажчик іншого або посилання одного типу у посилання іншого.

Цей оператор в основному використовують для динамічного виконання операцій приведення типу серед поліморфних типів. Наприклад, якщо задано поліморфні класи B і D, причому клас D виведений з класу B, то за допомогою оператора **dynamic_cast** завжди можна перетворити покажчик D* у покажчик B*, оскільки покажчик на базовий клас завжди можна використовувати для вказівки на об'єкт класу, виведеного з базового. Проте оператор **dynamic_cast** може перетворити покажчик B* у покажчик D* тільки у тому випадку, якщо адресованим об'єктом дійсно є об'єкт класу D. І, взагалі, оператор **dynamic_cast** буде успішно виконаний тільки за умови, якщо дозволено поліморфне приведення типів, тобто якщо покажчик (або посилання), що приводиться до нового типу, може вказувати (або посилатися) на об'єкт цього нового типу або об'єкт, який виведено з нього. Інакше, тобто якщо задану операцію приведення типів виконати не можна, то результат дії оператора **dynamic_cast** оцінюється як нульовий, якщо у цій операції беруть участь покажчики[1].

Розглянемо простий приклад. Припустимо, що клас Base поліморфний, а клас Derived виведений з класу Base.

```
Base *bp, ObjB;
Derived *dp, ObjD;

bp = &ObjD;       // Присвоєння покажчику адреси об'єкта похідного класу
                  // Покажчик на базовий клас вказує на об'єкт класу Derived.
dp = dynamic_cast<Derived *> (bp);    // Приведення до покажчика на похідний клас дозволено.
if(dp) cout << "Приведення типу відбулося успішно!";
```

Тут приведення покажчика bp (на базовий клас) до покажчика dp (на похідний клас) успішно здійснюється, оскільки bp *дійсно* вказує на об'єкт класу Derived. Тому у процесі виконання цього фрагмента програми буде виведено повідомлення:

Приведення типу відбулося успішно!

---

[1] Якщо ж спроба виконати цю операцію виявилася невдалою за участі в ній посилань, генерується виняток типу bad_cast.

Але наведений нижче фрагмент коду програми демонструє невдалу спробу зробити операцію приведення типу, оскільки bp насправді вказує на об'єкт класу Base, і неправомірно приводити покажчик на базовий клас до типу покажчика на похідний, якщо адресований ним об'єкт не є насправді об'єктом похідного класу.

```
bp = &ObjB;     // Присвоєння покажчику адреси об'єкта базового класу
                // Покажчик на базовий клас посилається на об'єкт класу Base.
dp = dynamic_cast<Derived *> (bp);   // Помилка!
if(!dp) cout << "Приведення типу виконати не вдалося";
```

Оскільки спроба виконати операцію приведення типу виявилася невдалою, то у процесі виконання цього фрагмента програми буде виведено повідомлення:

```
Приведення типу виконати не вдалося.
```

У наведеному нижче коді програми демонструються різні ситуації застосування оператора **dynamic_cast**.

Код програми 10.7. Демонстрація різних ситуацій застосування використання оператора dynamic_cast

```
#include <iostream>          // Для потокового введення-виведення
using namespace std;         // Використання стандартного простору імен

class baseClass {       // Оголошення базового класу
public:
	virtual void Fun() { cout << "У класі Base" << endl; }
	//. . .
};

class Derived: public Base {
public:
	void Fun() { cout << "У класі Derived" << endl; }
};

int main()
{
	Base *bp, ObjB;
	Derived *dp, ObjD;

	dp = dynamic_cast<Derived *> (&ObjD);
	if(dp) {
		cout << "Приведення типів " << "(з Derived * у Derived *) реалізовано" << endl;
		dp->Fun();
	}
	else
		cout << "Помилка" << endl;
	cout << endl;

	bp = dynamic_cast<Base *> (&ObjD);
	if(bp) {
		cout << "Приведення типів " << "(з Derived * у Base *) реалізовано" << endl;
		bp->Fun();
	}
```

```
else
        cout << "Помилка" << endl;
cout << endl;

bp = dynamic_cast<Base *> (&ObjB);
if(bp) {
        cout << "Приведення типів " << "(з Base * у Base *) реалізовано" << endl;
        bp->Fun();
}
else
        cout << "Помилка" << endl;
cout << endl;

dp = dynamic_cast<Derived *> (&ObjB);
if(dp) cout << "Помилка" << endl;
else
        cout << "Приведення типів " << "(з Base * у Derived *) не реалізовано" << endl;
cout << endl;

bp = &ObjD; // Присвоєння покажчику адреси об'єкта похідного класу
                  // bp вказує на об'єкт класу Derived
dp = dynamic_cast<Derived *> (bp);
if(dp) {
        cout << "Приведення bp до типу Derived *\n" << "реалізовано, оскільки bp дійсно\n"
                << "вказує на об'єкт класу Derived" << endl;
        dp->Fun();
}
else
        cout << "Помилка" << endl;
cout << endl;

bp = &ObjB; // Присвоєння покажчику адреси об'єкта базового класу
// bp вказує на об'єкт класу Base
dp = dynamic_cast<Derived *> (bp);
if(dp)
        cout << "Помилка";
else {
        cout << "Тепер приведення bp до типу Derived *\n" << "не реалізовано, оскільки bp\n"
                << "насправді вказує на об'єкт класу Base" << endl;
}
cout << endl;

dp = &ObjD; // Присвоєння покажчику адреси об'єкта похідного класу
                  // dp вказує на об'єкт класу Derived
bp = dynamic_cast<Base *> (dp);
if(bp) {
        cout << "Приведення dp до типу Base * реалізовано" << endl;
        bp->Fun();
}
else
```

```
            cout << "Помилка" << endl;
        getch(); return 0;
    }
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Приведення типів (з Derived * у Derived *) реалізовано.
У класі Derived.

Приведення типів (з Derived * у Base *) реалізовано.
У класі Derived.

Приведення типів (з Base * у Base *) реалізовано.
У класі Base.

Приведення типів (з Base * у Derived *) не реалізовано.

Приведення bp до типу Derived *
реалізовано, оскільки bp дійсно
вказує на об'єкт класу Derived.
У класі Derived.

Тепер приведення bp до типу Derived *
не реалізовано, оскільки bp
насправді вказує на об'єкт класу Base.

Приведення dp до типу Base * реалізовано.
У класі Derived.
```

Оператор **dynamic_cast** можна іноді використовувати замість оператора **typeid**. Наприклад, припустимо, що клас Base – поліморфний і є базовим для класу Derived, тоді у процесі виконання такого фрагмента коду програми покажчику dp буде присвоєно адресу об'єкта, яка адресується покажчиком bp, але тільки у тому випадку, якщо цей об'єкт дійсно є об'єктом класу Derived:

```
Base *bp;
Derived *dp;
//...
if(typeid(*bp) == typeid(Derived)) dp = (Derived *) bp;
```

У цьому випадку використовується звичайна операція приведення типів. Тут це цілком безпечно, оскільки настанова **if** перевіряє законність операції приведення типів за допомогою оператора **typeid** до її реального виконання. Те ж саме можна зробити ефективніше, замінивши операторів **typeid** і настанову **if** оператором **dynamic_cast**:

```
dp = dynamic_cast<Derived *> (bp);
```

Оскільки оператор **dynamic_cast** успішно здійснюється тільки у тому випадку, якщо об'єкт, який піддається операції приведення до типу, вже є об'єктом або заданого типу, або типу, виведеного із заданого, то після завершення цієї настанови покажчик dp міститиме або нульове значення, або покажчик на об'єкт типу Derived. Окрім цього, оскільки оператор **dynamic_cast** успішно здійснюється тільки у тому випадку, якщо задана операція приведення типів правомірна, то у певних ситуаціях її логіку можна спростити. У наведеному нижче коді програми показано, як

оператора **typeid** можна замінити оператором **dynamic_cast**. Тут здійснюється один і той самий набір операцій двічі: спочатку з використанням оператора **typeid**, а потім – оператора **dynamic_cast**.

Код програми 10.8. Демонстрація механізму використання оператора dynamic_cast замість оператора typeid

```
#include <iostream>            // Для потокового введення-виведення
#include <typeinfo>            // Для динамічної ідентифікації типів
using namespace std;           // Використання стандартного простору імен

class baseClass {        // Оголошення базового класу
public:
	virtual void Fun() {}
};

class Derived: public Base {
public:
	void derivedOnly() { cout << "Це об'єкт класу Derived" << endl; }
};

int main()
{
	Base *bp, ObjB;
	Derived *dp, ObjD;

	// Використання оператора typeid
	bp = &ObjB; // Присвоєння покажчику адреси об'єкта базового класу

	if(typeid(*bp) == typeid(Derived)) {
		dp = (Derived *) bp;
		dp->derivedOnly();
	}
	else
		cout << "Операції приведення об'єкта типу Base до "
			<< " типу Derived не здійснено" << endl;

	bp = &ObjD; // Присвоєння покажчику адреси об'єкта похідного класу
	if(typeid(*bp) == typeid(Derived)) {
		dp = (Derived *) bp;
		dp->derivedOnly();
	}
	else
		cout << "Помилка, приведення типу має " << "бути реалізовано!" << endl;
	// Використання оператора dynamic_cast
	bp = &ObjB; // Присвоєння покажчику адреси об'єкта базового класу
	dp = dynamic_cast<Derived *> (bp);

	if(dp) dp->derivedOnly();
	else
		cout << "Операції приведення об'єкта типу Base до "
			<< " типу Derived не здійснено" << endl;
	bp = &ObjD; // Присвоєння покажчику адреси об'єкта похідного класу
	dp = dynamic_cast<Derived *> (bp);
```

```
        if(dp) dp->derivedOnly();
        else
                cout << "Помилка, приведення типу має " << "бути реалізовано!" << endl;
        getch(); return 0;
}
```

Як бачите, використання оператора **dynamic_cast** спрощує логіку, необхідну для перетворення покажчика на базовий клас у покажчик на похідний клас. Ось як виглядають результати виконання цієї програми:

```
Операції приведення об'єкта типу Base до типу Derived не здійснено.
Це об'єкт класу Derived. Операції приведення об'єкта типу Base до
типу Derived не здійснено. Це об'єкт класу Derived.
```

Нео! хідноапам'ятати! *Оператор* ***dynamic_cast*** *можна також використовувати стосовно шаблонних класів.*

10.2.2. Оператор перевизначення модифікаторів const_cast

Оператор **const_cast** використовують для безпосереднього перевизначення модифікаторів **const** і/або **volatile**. Новий тип повинен збігатися з початковим, за винятком його атрибутів **const** або **volatile**. Найчастіше оператор **const_cast** використовують для видалення ознаки постійності (атрибуту **const**). Його загальний формат має такий вигляд:

```
const_cast<type> (expr)
```

У цьому записі елемент *type* задає новий тип операції приведення, а елемент *expr* означає вираз, який приводиться до нового типу.

Використання оператора **const_cast** продемонстровано в такому коді програми.

Код програми 10.9. Демонстрація механізму використання оператора const_cast

```
#include <iostream>             // Для потокового введення-виведення
using namespace std;            // Використання стандартного простору імен

void Fun(const int *p)
{
        int *v;
        v = const_cast<int *> (p);          // Перевизначення const-атрибуту.
        *v = 100; // Тепер об'єкт можна модифікувати
}

int main()
{
        int x = 99;
        cout << "Значення x до виклику функції Fun(): " << x << endl;

        Fun(&x);
        cout << "Значення x після виклику функції Fun(): " << x << endl;
        getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Значення x до виклику функції Fun(): 99
Значення x після виклику функції Fun(): 100
```

Як бачите, змінна x була модифікована функцією Fun(), хоча параметр, який вона приймає, задається як **const**-покажчик.

Необхідно наголосити на тому, що використання оператора **const_cast** для видалення **const**-атрибуту є потенційно небезпечним засобом. Тому поводьтеся з ним дуже обережно.

Вартоа ' нати! *Видаляти* ***const****-атрибут здатний тільки оператор* ***const_cast****. Іншими словами, ні* ***dynamic_cast****, ні* ***static_cast****, ні* ***reinterpret_cast*** *не можна використовувати для зміни* ***const****-атрибуту об'єкта.*

10.2.3. Оператор неполіморфного приведення типів static_cast

Його можна використовувати для будь-якого стандартного перетворення. При цьому під час роботи коду програми ніяких перевірок на допустимість не здійснюється. Оператор **static_cast** має такий загальний формат запису:

static_cast<*type*> (*expr*)

У цьому записі елемент *type* задає новий тип операції приведення, а елемент *expr* означає вираз, який приводиться до цього нового типу.

Оператор **static_cast**, по суті, є замінником оригінального оператора приведення типів. Він тільки здійснює неполіморфне перетворення. Наприклад, у процесі виконання наведеної нижче програми змінна типу **float** приводиться до типу **int**.

Код програми 10.10. Демонстрація механізму використання оператора static_cast

```
#include <iostream>          // Для потокового введення-виведення
using namespace std;         // Використання стандартного простору імен

int main()
{
	int c;
	float f;
	f = 199.22F;
	c = static_cast<int> (f);

	cout << c;
	getch(); return 0;
}
```

10.2.4. Оператор перетворення типу reinterpret_cast

Оператор **reinterpret_cast** перетворить один тип у принципово інший. Наприклад, його можна використовувати для перетворення покажчика в ціле значення і ціле значення у покажчик. Його також можна використовувати для приведення спадково несумісних типів покажчиків. Цей оператор має такий загальний формат запису:

reinterpret_cast<*type*> (*expr*)

У цьому записі елемент *type* задає новий тип операції приведення, а елемент *expr* означає вираз, який приводиться до цього нового типу. Використання оператора **reinterpret_cast** продемонстровано в такому коді програми.

Код програми 10.11. Демонстрація механізму використання оператора reinterpret_cast

```
#include <iostream>            // Для потокового введення-виведення
using namespace std;           // Використання стандартного простору імен

int main()
{
	int c;
	char *p = "Це короткий рядок.";

	c = reinterpret_cast<int> (p); // Приводимо покажчик до типу int.

	cout << c;
	getch(); return 0;
}
```

У цьому коді програми оператор **reinterpret_cast** перетворить покажчик p у цілочисельне значення. Внаслідок такого перетворення відбувається фундаментальна зміну типу.

10.2.5. Порівняння звичайної операції приведення типів з новими чотирма cast-операторами

Можливо, комусь з Вас могло б видатися, що описані вище чотири **cast**-оператори повністю замінюють традиційну операцію приведення типів. На запитання: "Чи варто завжди замість звичайної операції приведення типів використовувати новіші засоби?". Відповімо – загального правила для всіх програмістів не існує. Оскільки нові оператори були створені для підвищення безпеки достатньо ризикованої операції приведення одного типу даних до іншого, багато C++-програмістів переконані у тому, що їх необхідно використовувати виключно з цією метою. І тут важко що-небудь заперечити. Інші ж програмісти вважають, що, оскільки традиційна операція приведення типів надійно слугувала їм протягом багатьох років, то від неї не варто так легко відмовлятися. Наприклад, для виконання простих і відносно безпечних операцій приведення типів (як ті, що потрібні під час виклику функцій введення-виведення **read**() і **write**(), описаних у попередньому розділі) "старий добрий" засіб цілком прийнятний.

Існує ще одна точка зору, з якою важко не погодитися: у процесі виконання операцій приведення поліморфних типів безумовно варто використовувати оператор **dynamic_cast**.

Розділ 11. ПОНЯТТЯ ПРО ПРОСТОРИ ІМЕН ТА ІНШІ ЕФЕКТИВНІ ПРОГРАМНІ ЗАСОБИ

У цьому розділі буде описано основні поняття про простори імен і такі ефективні програмні засоби, як **explicit**-конструктори, покажчики на функції, **static**-члени, **const**-функції-члени, альтернативний синтаксис ініціалізації членів-даних класу, оператори вказання на члени класу, ключове слово **asm**, специфікація компонування і функції перетворення.

11.1. Особливості організації простору імен

Простори імен стисло розглянуто у розд. 2 [9]. Там було сказано, що вони дають змогу локалізувати імена ідентифікаторів, щоб уникнути конфліктних ситуацій з ними. У С++-середовищі програмування використовується величезна кількість імен змінних, функцій та імен класів. До введення поняття простору імен всі ці імена конкурували між собою за пам'ять в глобальному просторі імен, що і було причиною виникнення багатьох конфліктів. Наприклад, якби у Вашій програмі було визначено функцію **toupper**(), то вона могла б (залежно від переліку параметрів) перевизначити стандартну бібліотечну функцію **toupper**(), оскільки обоє імен мали б зберігатися в глобальному просторі імен. Конфлікти з іменами виникали раніше також під час використання однією програмою декількох бібліотек сторонніх виробників. У цьому випадку ім'я, що є визначеним у одній бібліотеці, конфліктувало з таким самим іменем з іншої бібліотеки. Така ситуація особливо неприйнятна під час використання однойменних класів. Наприклад, якщо у Вашій програмі визначено клас VideoMode, і в бібліотеці, що використовуються Вашою програмою, визначено клас з таким самим іменем, то конфлікту не уникнути.

Простір імен визначає деяку декларативну область.

Для вирішення описаної проблеми було створено ключове слово **namespace**. Оскільки воно локалізує видимість оголошених у ньому імен, то це означає, що простір імен дає змогу використовувати одне і те саме ім'я в різних контекстах, не викликаючи при цьому конфлікту імен. Можливо, найбільше від нововведення "поталанило" С++-бібліотеці стандартних функцій. До появи ключового слова **namespace** (яке було, звичайно ж, єдиним) всю С++-бібліотеку було визначено в глобальному просторі імен. З настанням **namespace**-"ери" С++-бібліотека визначається у власному просторі імен, названий **std**, який значно знизив ймовірність виникнення конфліктів імен. У своїй програмі програміст повинен створювати власні простори імен, щоб локалізувати видимість тих імен, які, на його думку, можуть стати причиною конфлікту. Це особливо важливо, якщо Ви займаєтеся створенням бібліотек класів або функцій.

11.1.1. Поняття про простори імен

Ключове слово **namespace** дає змогу розділити глобальний простір імен шляхом створення певної декларативної області. По суті, простір імен визначає область видимості. Загальний формат задавання простору імен є таким:

```
namespace name {
    // Оголошення
}
```

Все, що визначено у межах настанови **namespace**, знаходиться в області видимості цього простору імен.

У наведеному нижче коді програми показано приклад використання **namespace**-настанови. Вона локалізує імена, що використовуються для виконання розрахунку у зворотному порядку. У створеному тут просторі імен визначається клас counter, який реалізує лічильник, і змінні upperbound і lowerbound, що містять значення верхньої і нижньої меж, які використовуються для всіх лічильників.

```
// Приклад використання namespace-настанови
namespace CounterNameSpace
{
    int upperbound;
    int lowerbound;

    class counter {
        int count;
    public:
        counter(int n) {
            if(n <= upperbound) count = n;
            else count = upperbound;
        }

        void Reset(int n) {
            if(n <= upperbound) count = n;
        }

        int Run() {
            if(count > lowerbound) return count--;
            else return lowerbound;
        }
    };
}
```

У цьому коді програми змінні upperbound і lowerbound, а також клас counter є частиною області видимості, визначеної простором імен CounterNameSpace.

У будь-якому просторі імен до ідентифікаторів, які у ньому оголошені, можна звертатися безпосередньо, тобто без вказання цього простору імен. Наприклад, у функції Run(), яка знаходиться у просторі імен CounterNameSpace, можна безпосередньо звертатися до змінної lowerbound:

```
if(count > lowerbound) return count—;
```

Але, оскільки настанова **namespace** визначає область видимості, то під час звернення до об'єктів, оголошених у просторі імен, ззовні цього простору необхідно використовувати оператор дозволу області видимості. Наприклад, щоб присво-

їти значення 10-ій змінній upperbound з коду програми, який є зовнішнім стосовно простору імен CounterNameSpace, потрібно використовувати таку настанову:

CounterNameSpace::upperbound = 10;

Щоб оголосити об'єкт типу counter поза простором імен CounterNameSpace, потрібно використати настанову, подібну до такої:

CounterNameSpace::counter obj;

У загальному випадку, щоб отримати доступ до деякого члена простору імен ззовні цього простору, необхідно, щоби імені цього члена передувало ім'я простору і розділити ці імена оператором дозволу області видимості.

Розглянемо невелику програму, у якій продемонстровано механізм використання простору імен CounterNameSpace.

Код програми 11.1. Демонстрація механізму використання простору імен CounterNameSpace для розрахунку у зворотному порядку

```
#include <vcl>
#include <iostream>            // Для потокового введення-виведення
#include <conio>               // Для консольного режиму роботи
using namespace std;           // Використання стандартного простору імен

// Приклад використання namespace-настанови
namespace CounterNameSpace
{
	int upperbound;
	int lowerbound;

	class counter {
		int count;
	public:
		counter(int n) {
			if(n <= upperbound) count = n;
			else count = upperbound;
		}

		void Reset(int n) {
			if(n <= upperbound) count = n;
		}

		int Run() {
			if(count > lowerbound) return count--;
			else return lowerbound;
		}
	};
}

int main()
{
	CounterNameSpace::upperbound = 100;
	CounterNameSpace::lowerbound = 0;

	CounterNameSpace::counter ObjA(10);
	CounterNameSpace::counter ObjB(20);
```

```
    int c;

    cout << "Розрахунок у зворотному порядку для об'єкта ObjA" << endl;
    do {
        c = ObjA.Run();
        cout << c << " ";
    } while(c > CounterNameSpace::lowerbound);
    cout << endl;

    cout << "Розрахунок у зворотному порядку для об'єкта ObjB" << endl;
    do {
        c = ObjB.Run();
        cout << c << " ";
    } while(c > CounterNameSpace::lowerbound);
    cout << endl;

    ObjB.Reset(100);
    CounterNameSpace::lowerbound = 80;

    cout << "Розрахунок у зворотному порядку для об'єкта ObjB" << endl;
    do {
        c = ObjB.Run();
        cout << c << " ";
    } while(c > CounterNameSpace::lowerbound);
    cout << endl;

    getch(); return 0;
}
```

Зверніть увагу на те, що при створенні об'єкта класу counter і при зверненні до змінних upperbound і lowerbound використовують ім'я простору імен CounterNameSpace. Але після створення об'єкта типу counter вже немає потреби у повній кваліфікації його самого або його членів. Оскільки простір імен однозначно визначено, то функцію Run() об'єкта ObjA можна викликати безпосередньо, тобто без вказання (як префікс) простору імен (ObjA.Run()).

Програма може містити декілька оголошень просторів імен з однаковими іменами. Це означає, що простір імен можна поділити на декілька файлів або на декілька частин у рамках одного файлу. Зробити це можна так:

```
namespace NS {
    int c;
}

//....

namespace NS {
    int d;
}
```

У цьому записі простір імен NS розділено на дві частини. Проте вміст кожної частини належить до одного і того ж простору імен NS.

Будь-який простір імен повинен бути оголошений поза всіма іншими областями видимості. Це означає, що не можна оголошувати простори імен, які локалізовані, наприклад, у межах функції. При цьому один простір імен може бути вкладено в інший.

11.1.2. Застосування настанови using

Якщо програма містить багато посилань на члени певного простору імен, то неважко уявити, що потреба вказувати ім'я цього простору імен при кожному зверненні до них, дуже скоро набридне Вам. Цю проблему дає змогу вирішити настанова **using**, яка застосовується у таких двох форматах:

```
using namespace ім'я;
using name::член;
```

*Настанова **using** робить заданий простір імен "видимим", тобто діючим.*

У першій формі елемент *ім'я* задає назву простору імен, до якого Ви зможете отримати доступ. Всі члени, визначені усередині заданого простору імен, потрапляють в "поле видимості", тобто стають частиною поточного простору імен і їх можна потім використовувати без кваліфікації (уточнення простору імен). У другий формі робиться "видимим" тільки вказаний член простору імен. Наприклад, вважаючи, що простір імен CounterNameSpace визначено (як це показано вище), то наступні настанови **using** і присвоєння будуть цілком законними:

```
using CounterNameSpace::lowerbound;         // Видимим став тільки член lowerbound.
lowerbound = 10; // Все гаразд, оскільки член lowerbound знаходиться в області видимості.

using namespace CounterNameSpace; // Всі члени видимі.
upperbound = 100; // Все гаразд, оскільки всі члени видимі.
```

Використання настанови **using** продемонстровано у наведеному нижче коді програми (яка є новим варіантом лічильника з попереднього розділу).

Код програми 11.2. Демонстрація механізму використання настанови using для виконання розрахунку у зворотному порядку

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен

namespace CounterNameSpace
{
	int upperbound;
	int lowerbound;

	class counter {
		int count;
	public:
		counter(int n) {
			if(n <= upperbound) count = n;
			else count = upperbound;
```

```
                }

                void Reset(int n) {
                        if(n <= upperbound) count = n;
                }

                int Run() {
                        if(count > lowerbound) return count--;
                        else return lowerbound;
                }
        };
}

int main()
{
        // Використовується тільки член upperbound з
        // простору імен CounterNameSpace.
        using CounterNameSpace::upperbound;

        // Тепер для встановлення значення змінній upperbound
        // не потрібно вказувати простір імен.
        upperbound = 100;

        // Але під час звернення до змінної lowerbound і до інших
        // об'єктів, як і раніше, необхідно вказувати простір імен.
        CounterNameSpace::lowerbound = 0;
        CounterNameSpace::counter ObjA (10);
        int c;

        cout << "Розрахунок у зворотному порядку для об'єкта ObjA" << endl;
        do {
                c = ObjA.Run();
                cout << c << " ";
        } while(c > CounterNameSpace::lowerbound);
        cout << endl;

        // Тепер використовуємо весь простір імен CounterNameSpace.
        using namespace CounterNameSpace;

        counter ObjB(20);

        cout << "Розрахунок у зворотному порядку для об'єкта ObjB" << endl;
        do {
                c = ObjB.Run();
                cout << c << " ";
        } while(c > lowerbound);
        cout << endl;

        ObjB.Reset(100);
        lowerbound = 80;
        cout << "Розрахунок у зворотному порядку для об'єкта ObjB" << endl;
        do {
                c = ObjB.Run();
```

```
            cout << c << " ";
        } while(c > lowerbound);
        cout << endl;

        getch(); return 0;
    }
```

Ця програма ілюструє ще один важливий момент: якщо певний простір імен стає "видимим", то це означає, що він просто додає свої імена до імен інших, вже діючих просторів. Тому до кінця цієї програми до глобального простору імен додалися і **std**, і CounterNameSpace.

11.1.3. Неіменовані простори імен

Існує *неіменований* простір імен спеціального типу, який дає змогу створювати ідентифікатори, унікальні для цього файлу. Загальний формат його оголошення має такий вигляд:

```
namespace {
    // Оголошення
}
```

Неіменовані простори імен дають змогу встановлювати унікальні ідентифікатори, які відомі тільки в області видимості одного файлу. Іншими словами, члени файлу, який містить неіменований простір імен, можна використовувати безпосередньо, без уточнюючого префікса. Але поза файлом ці ідентифікатори є невідомими.

Неіменований простір імен обмежує ідентифікатори рамками файлу, у якому їх оголошено.

Як уже зазначалося вище у цьому навчальному посібнику, використання модифікатора типу **static** також дає змогу обмежити область видимості глобального простору імен файлом, у якому він оголошений. Наприклад, розглянемо такі два файли, які є частиною однієї і тієї ж самої програми:

Файл One	Файл Two

static int f;
void Fun1() {
    f = 99; //OK
}
	extern int f;
	void Fun2() {
	    f = 10; // Помилка
	}

Оскільки змінну f визначено у файлі One, то її і можна використовувати у файлі One. У файлі Two змінну f визначено як зовнішню (**extern**-змінна), а це означає, що її ім'я і тип відомі, але саму змінну f насправді не визначено. Коли ці два файли будуть скомпоновані, спроба використовувати змінну f у файлі Two призведе до виникнення помилки, оскільки у ньому немає визначення для змінної f. Той факт, що f оголошена як **static**-змінна у файлі One, означає, що її область видимості обмежується цим файлом, і тому вона недоступна для файлу Two.

Незважаючи на те, що використання глобальних **static**-оголошень все ще дозволено стандартом мови програмування C++, проте для локалізації ідентифікатора у межах одного файлу краще використовувати неіменований простір імен. Розглянемо такий приклад:

Файл One	Файл Two

`namespace {`
`    int f;`
`}`
`static int f;`
`void Fun1() {`
`    f = 99; // OK`
`}`
	`extern int f;`
	`void Fun2() {`
	`    f = 10;        // Помилка`
	`}`

Тут змінну f також обмежено рамками файлу One. Для нових програм рекомендується використовувати замість модифікатора **static** неіменований простір імен.

Зазвичай для більшості коротких програм і програм середнього розміру немає потреби у створенні просторів імен. Але, формуючи бібліотеки багаторазово використовуваних функцій або класів, є сенс помістити свій програмний код (якщо хочете забезпечити його максимальну переносність) у власний простір імен.

11.1.4. Застосування простору імен std

Стандарт мови програмування C++ визначає всю свою бібліотеку у власному просторі імен, який іменується **std**. Саме з цієї причини більшість програм у цьому навчальному посібнику містять таку настанову:

```
using namespace std;            // Використання стандартного простору імен
```

У процесі виконання цієї настанови простір імен **std** стає поточним, що відкриває прямий доступ до імен функцій і класів, визначених у цій бібліотеці, тобто під час звернення до них відпадає необхідність у використанні префікса **std::**.

Простір імен ***std*** *використовується власною бібліотекою мови програмування C++.*

Звичайно, при бажанні можна безпосередньо кваліфікувати кожне бібліотечне ім'я префіксом **std::**. Наприклад, наведений нижче код програми не привносить бібліотеку в глобальний простір імен.

Код програми 11.3. Демонстрація механізму використання безпосередньо заданої кваліфікації бібліотечних імен префіксом std::

```
#include <iostream>             // Для потокового введення-виведення
int main()
{
    double n;
    std::cout << "Введіть число: "; std::cin >> n;

    std::cout << "Ви ввели число ";
```

```
    std::cout << n;
    getch(); return 0;
}
```

У цьому коді програми імена **cout** і **cin** безпосередньо доповнені іменами своїх просторів імен. Отже, щоб записати дані у стандартний вихідний потік, необхідно використовувати не просто ім'я потоку **cout**, а ім'я з префіксом **std::cout**, а щоб зчитати дані із стандартного вхідного потоку, потрібно застосувати "префіксне" ім'я **std::cin**.

Якщо Ваша програма використовує стандартну бібліотеку тільки в обмежених межах, то, можливо, її і не варто вносити в глобальний простір імен. Але, якщо Ваша програма містить сотні посилань на бібліотечні імена, то набагато простіше зробити простір імен **std** поточним, ніж повністю кваліфікувати кожне ім'я окремо.

Якщо Ви використовуєте тільки декілька імен із стандартної бібліотеки, то, ймовірно, є сенс використовувати настанову **using** для кожного з них окремо. Перевага цього підходу полягає у тому, що ці імена можна, як і раніше, використовувати без префікса **std::**, не вносячи при цьому всю бібліотеку стандартних функцій у глобальний простір імен. Розглянемо такий приклад.

Код програми 11.4. Демонстрація механізму внесення в глобальний простір імен декількох імен

```
#include <iostream>          // Для потокового введення-виведення
// Отримуємо доступ до імен потоків cout і cin
using std::cout;
using std::cin;

int main()
{
    double n;

    cout << "Введіть число: "; cin >> n;

    cout << "Ви ввели число ";
    cout << n;
    getch(); return 0;
}
```

У цьому коді програми імена потоків **cin** і **cout** можна використовувати безпосередньо, але інша частина простору імен **std** не внесена в область видимості.

Як уже зазначалося вище, початкова бібліотека мови програмування С++ була визначена в глобальному просторі імен. Якщо Вам доведеться модернізувати старі С++-програми, то програміст повинен або включити в них настанову **using namespace std**, або доповнити кожне звернення до члена бібліотеки префіксом **std::**. Це особливо важливо, якщо Вам доведеться замінювати старі заголовні *.h-файли сучасними заголовками.

Нео! хідноапам'ятати! *Старі заголовні *.h-файли поміщають свій вміст у глобальний простір імен, а сучасні заголовки – у простір імен* ***std***.

11.2. Застосування покажчиків на функції

Покажчик на функцію – це достатньо складний, але дуже потужний засіб С++-програмування. Незважаючи на те, що функція не є змінною, проте вона займає фізичну область пам'яті, певну адресу якої можна присвоїти покажчику. Адреса, що присвоюється покажчику, є вхідною точкою функції[1]. Якщо деякий покажчик посилається на функцію, то її (функцію) можна викликати за допомогою цього покажчика.

Покажчик на функцію посилається на вхідну точку цієї функції.

Покажчики на функції також дають змогу передавати функції як аргументи іншим функціям. Адресу функції можна отримати, використовуючи ім'я функції без круглих дужок і аргументів[2]. Якщо присвоїти адресу функції покажчику, то цю функцію можна викликати через покажчик. Щоб зрозуміти сказане, розглянемо наведену нижче програму. Вона містить дві функції – vLine() і hLine(), які малюють на екрані вертикальні та горизонтальні лінії заданої довжини.

Код програми 11.5. Демонстрація механізму застосування покажчиків на функції

```
#include <vcl>
#include <iostream>           // Для потокового введення-виведення
#include <conio>              // Для консольного режиму роботи
using namespace std;          // Використання стандартного простору імен

void vLine(int i), hLine(int i);

int main()
{
	void (*p)(int i);

	p = vLine; // Покажчик на функцію vLine()

	(*p)(4);	// Виклик функції vLine()

	p = hLine; // Покажчик на функцію hLine()

	(*p)(5);	// Виклик функції hLine()
	getch(); return 0;
}

void hLine(int i)
{
	for(; i; i--) cout << "-";
	cout << endl;
}
```

---

[1] Саме ця адреса використовується під час виклику функції.
[2] Цей процес подібний до процесу отримання адреси масиву, коли також використовується тільки його ім'я без індексу.

```
void vLine(int i)
{
    for(; i; i--) cout << "|" << endl;
}
```

Ось як виглядають результати виконання цієї програми:

```
|
|
|
|
____
```

Розглянемо цю програму у деталях. У першому рядку тіла функції **main**() визначається змінна p як покажчик на функцію, яка приймає один цілочисельний аргумент і не повертає ніякого значення. Це оголошення не визначає, про яку саме функцію йдеться. Воно тільки створює покажчик, який можна використовувати для адресації будь-якої функції цього типу. Необхідність круглих дужок, у які поміщено покажчик *p, випливає з С++-правил передування.

У наступному рядку покажчику p присвоюється адреса функції vLine(). Потім здійснюється виклик функції vLine() з аргументом 4. Після цього покажчику p присвоюється адреса функції hLine(), і за допомогою цього покажчика реалізується її виклик. У цьому коді програми під час виклику функцій за допомогою покажчика використовується такий формат:

```
(*p)(4);
```

Проте функцію, яка адресується покажчиком p, можна викликати з використанням дещо простішого синтаксису:

```
p(4);
```

Єдина причина, згідно з якою частіше використовується перший варіант виклику функції, полягає у тому, що всім, хто буде аналізувати Вашу програму, стане зрозуміло, що тут реалізовано виклик функції через покажчик p, а не виклик функції з іменем p. У всьому іншому ці варіанти є еквівалентними.

11.2.1. Передача покажчиком на функцію її адреси іншій функції

Незважаючи на те, що у наведеному вище прикладі покажчик на функцію використано тільки заради ілюстрації, часто таке його застосування має дуже важливе значення. Покажчик на функцію дає змогу передавати її адресу іншій функції. Як показовий приклад можна навести функцію **qsort**() із стандартної С++-бібліотеки. Функція **qsort**() – це функція швидкого сортування, що базується на алгоритмі Quicksort, який упорядковує вміст масиву. Її прототип має такий вигляд:

```
void qsort(void *start, size_t length size_t size,
int (*compare) (const void *, const void *));
```

Прототип функції **qsort**() "прописаний" у заголовку **<cstdlib>**, у якому також визначено тип **size_t** (як тип **unsigned int**). Щоб використовувати функцію **qsort**(), необхідно передати їй покажчик на початок масиву об'єктів, який Ви хочете відсортувати (параметр start), довжину цього масиву (параметр *length*), розмір у байтах кожного елемента (параметр size) і покажчик на функцію порівняння елементів масиву (параметр * ***compare***).

Функція порівняння, що використовується функцією **qsort**(), зіставляючи два елементи масиву, повинна повернути негативне значення, якщо її перший аргумент вказує на значення, яке є меншим від другого, нуль, якщо ці аргументи однакові, і позитивне значення, якщо перший аргумент вказує на значення, яке є більшим від другого.

Щоби зрозуміти, як можна використовувати функцію **qsort**(), розглянемо наведену нижче програму.

Код програми 11.6. Демонстрація механізму використання бібліотечної функції qsort() для сортування елементів текстового масиву

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
#include <cstdlib>           // Для використання бібліотечних функцій
#include <cstring>           // Для роботи з рядковими типами даних
using namespace std;         // Використання стандартного простору імен

int Comp(const void *a, const void *b);

int main()
{
	char str[] = "Покажчики на функції дають гнучкість.";

	qsort(str, strlen(str), 1, Comp);
	cout << "Відсортований рядок: " << str;
	cout << endl;
	getch(); return 0;
}

int Comp(const void *a, const void *b)
{
	return * (char *) a – * (char *) b;
}
```

Ось як виглядають результати виконання цієї програми.

```
Відсортований рядок: .Пааагджииіїїкккннносттууфцччьью
```

Ця програма сортує рядок str у зростаючому порядку. Оскільки функції **qsort**() передається вся необхідна їй інформація, в т.ч. покажчик на функцію порівняння, то її можна використовувати для сортування даних будь-якого типу. Наприклад, наведений нижче код програми сортує масив цілих чисел. Для гарантії переносності під час визначення розміру цілочисельного значення в ній використовують оператор **sizeof**.

Код програми 11.7. Демонстрація механізму використання бібліотечної функції qsort() для сортування елементів цілочисельного масиву

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
#include <cstdlib>           // Для використання бібліотечних функцій
using namespace std;         // Використання стандартного простору імен
```

```
int Comp(const void *a, const void *b);

int main()
{
	int num[] = {10, 4, 3, 6, 5, 7, 8};

	qsort(num, 7, sizeof(int), Comp);

	for(int i=0; i<7; i++)
	cout << num[i]<< " ";
	cout << endl;

	getch(); return 0;
}

int Comp(const void *a, const void *b)
{
	return * (char *) a – * (char *) b;
}
```

Не станемо заперечувати, що покажчики на функції є не простими для розуміння, але практика їх використання допоможе і "з ними знайти порозуміння". Відносно покажчиків на функції необхідно розглянути ще один аспект, що пов'язаний з перевизначеними функціями.

11.2.2. Пошук адреси перевизначеної функції

Отримати адресу перевизначеної функції трохи складніше, ніж знайти адресу звичайної "одиночної" функції. Якщо ж існує декілька версій перевизначеної функції, то повинен існувати механізм, який би визначав, адресу якої саме її версії ми отримуємо. Під час отримання адреси перевизначеної функції саме *спосіб оголошення покажчика* визначає, адресу якої її версії буде отримано. По суті, оголошення покажчика у цьому випадку порівнюється з відповідними оголошеннями покажчиків перевизначених функцій. Функція, оголошення якої виявить збіг, і буде тією функцією, адресу якої ми отримали.

У наведеному нижче прикладі коду програми міститься дві версії функції space(). Перша версія виводить на екран count пропусків, а друга – count символів, переданих як аргумент ch. У функції **main**() оголошуються два покажчики на функції. Перший заданий як покажчик на функцію з одним цілочисельним параметром, а другий – як покажчик на функцію з двома параметрами.

Код програми 11.8. Демонстрація механізму використання покажчиків на дві перевизначені функції для виведення на екран відповідно пропусків і символів

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен
```

```
// Введення на екран count пропусків.
void space(int count)
{
    for(; count; count--) cout << " ";
}

// Введення на екран count символів, переданих в ch.
void space(int count, char ch)
{
    for(; count; count--) cout << ch;
}

int main()
{
    // Створення покажчика на void-функцію з одним int-параметром.
    void (*fp1)(int);

    /* Створення покажчика на void-функцію з одним int-параметром
    і одним параметром типу char. */
    void (*fp2)(int, char);

    fp1 = space; // Отримуємо адресу функції space(int)

    fp2 = space; // Отримуємо адресу функції space(int, char)

    fp1(22);    // Виводимо 22 пропуски (цей виклик є аналогічним
                      // виклику (*fp1)(22)).
    cout << "|" << endl;

    fp2(30, 'x'); // Виводимо 30 символів "x" (цей виклик є
                      // аналогічним до виклику (*fp2)(30, 'x').
    cout << "|" << endl;

    getch(); return 0;
}
```

Ось як виглядають результати виконання цієї програми:

```
                      |
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx|
```

Як зазначено у коментарях до цієї програми, компілятор здатний визначити, адресу якої перевизначеної функції він отримує, на основі того, як оголошено покажчики fp1 і fp2.

Отже, коли адреса перевизначеної функції присвоюється покажчику на функцію, то саме це оголошення покажчика слугує основою для визначення того, адресу якої функції було присвоєно. При цьому оголошення покажчика на функцію повинно відповідати одній (і тільки одній) з перевизначених функцій. Інакше у програму вноситься неоднозначність, яка викличе помилку компілювання.

11.3. Поняття про статичні члени-даних класу

Ключове слово **static** можна застосовувати і до членів-даних класу. Оголошуючи член-даних класу статичним, ми тим самим повідомляємо компілятор про те, що, незалежно від того, скільки об'єктів цього класу буде створено, існує тільки одна копія цього **static**-члена. Іншими словами, **static**-член розділяється між всіма об'єктами класу. Всі статичні дані під час першого створення об'єкта ініціалізуються нульовими значеннями, якщо перед цим не представлено інших значень ініціалізації.

Під час оголошення статичного члена-даних у класі програміст не повинен його визначати. Необхідно забезпечити його глобальне визначення поза цим класом. Це реалізується шляхом повторного оголошення цієї статичної змінної за допомогою оператора дозволу області видимості, який дає змогу ідентифікувати, до якого класу вона належить. Тільки у цьому випадку для цієї статичної змінної буде виділено пам'ять.

Розглянемо приклад використання **static**-члена класу. Вивчіть код цієї програми і постарайтеся зрозуміти, як вона працює.

Код програми 11.9. Демонстрація механізму використання статичних членів-даних класу

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен

class ShareVar {
	static int n;
public:
	void Set(int c) { n = c; }
	void Show() { cout << n << " "; }
};

int ShareVar::n; // Визначаємо static-член n

int main()
{
	ShareVar ObjA, ObjB;

	ObjA.Show(); // Виводиться 0
	ObjB.Show(); // Виводиться 0
	ObjA.Set(10); // Встановлюємо static-члена n дорівнює 10
	ObjA.Show(); // Виводиться 10
	ObjB.Show(); // Також виводиться 10

	getch(); return 0;
}
```

Зверніть увагу на те, що статичний цілочисельний член n оголошений і у класі ShareVar, і його визначено як глобальна змінна. Як було сказано вище, необхід-

ність такого подвійного оголошення зумовлена тим, що під час оголошення члена n у класі ShareVar пам'ять для нього не виділяється. Компілятор C++ ініціалізував змінну n значенням 0, оскільки ніякої іншої ініціалізації у програмі немає. Тому внаслідок двох перших викликів функції Show() для об'єктів ObjA і ObjB відображається значення 0. Потім об'єкт ObjA встановлює члену n значення, яке дорівнює 10, після чого об'єкти ObjA і ObjB знову виводять на екран його значення за допомогою функції Show(). Але оскільки існує тільки одна копія змінної n, що розділяється об'єктами ObjA і ObjB, то значення 10 буде виведено під час виклику функції Show() для обох об'єктів.

Нео! хідноапам'ятати! *Під час оголошення члена класу статичним Ви забезпечуєте створення тільки однієї його копії, яка буде сумісно використовуватися всіма об'єктами цього класу.*

Якщо **static**-змінна є відкритою (тобто **public**-змінною), до неї можна звертатися безпосередньо через ім'я її класу, без посилання на будь-який конкретний об'єкт[1]. Розглянемо, наприклад, таку версію класу ShareVar:

```
class ShareVar {
public:
        static int n;
        void Set(int c) { n = c; };
        void Show() { cout << n << " "; }
};
```

У даній версії змінна n є public-членом даних класу. Це дає змогу нам звертатися до неї безпосередньо, як це показано в такій настанові:

```
ShareVar::n = 100;
```

У цьому записі значення змінної n встановлюється незалежно від об'єкта, а для звернення до неї достатньо використовувати ім'я класу і оператора дозволу області видимості. Більше того, ця настанова є правомірною навіть для створення яких-небудь об'єктів типу ShareVar. Таким чином, отримати або встановити значення **static**-члена класу можна до того, як будуть створені будь-які об'єкти.

І хоча Ви, можливо, поки що не відчули потреби в **static**-членах класу, проте, у міру набуття досвіду програмування мовою C++, Вам доведеться наштовхнутися на ситуацію, коли вони виявляться дуже корисними, тобто дадуть змогу уникнути застосування глобальних змінних.

Можна також оголосити статичною і функцію-члена класу, але це непоширена практика. До статичної функції-члена класу можуть отримати доступ тільки інші **static**-члени цього класу[2]. Статична функція-член не має покажчика **this**. Створення віртуальних статичних функцій-членів класу не дозволяється. Окрім цього, їх не можна оголошувати з модифікаторами **const** або **volatile**. Статичну функцію-члена можна викликати для об'єкта її класу або незалежно від будь-якого об'єкта, а для звернення до неї достатньо використовувати ім'я класу і оператор дозволу області видимості.

---

[1] Безумовно, звертатися можна також і через ім'я об'єкта.

[2] Звичайно ж, статична функція-член класу може отримувати доступ до нестатичних глобальних даних і функцій.

11.5. Застосування до функцій-членів класу модифікаторів const і mutable

Функції-члени класу можуть бути оголошені з використанням модифікатора **const**. Це означає, що з покажчиком **this** у цьому випадку необхідно звертатися як з **const**-покажчиком. Іншими словами, **const**-функція не може модифікувати об'єкт, для якого вона викликана. Окрім цього, **const**-об'єкт не може викликати не **const**-функцію-члена класу. Але **const**-функцію-члена можуть викликати як **const**-, так і не **const**-об'єкти.

Щоб визначити функцію як **const**-члена класу, використовується формат, який подано у наведеному нижче прикладі:

```
class aType {     // Оголошення класового типу
        int some_var;
public:
        int Fun1() const; // const-функція-член
};
```

Як бачите, модифікатор **const** розташовується після оголошення переліку параметрів функції.

Мета оголошення функції як **const**-члена класу – не допустити модифікацію об'єкта, який її викликає. Для розуміння сказаного розглянемо наведену нижче програму.

Код програми 11.10. Демонстрація механізму використання const-функцій-членів класу

```
// Ця програма не відкомпілюється.
#include <iostream>              // Для потокового введення-виведення
using namespace std;             // Використання стандартного простору імен

class Demo {
        int c;
public:
        int Put() const;
        return c; // Все гаразд
}

void Set(int x) const
{
        c = x;        // Помилка!
};

int main()
{
        Demo Obj;
        Obj.Set(1900);
        cout << Obj.Put();

        getch(); return 0;
}
```

Ця програма не відкомпілюється, оскільки функцію Set() оголошено як **const**-член. Це означає, що їй не дозволено модифікувати той об'єкт, який її викликає. Її спроба змінити вміст змінної c призводить до виникнення помилки. На відміну від функції Set(), функція Put() не намагається модифікувати змінну c, і тому вона абсолютно прийнятна.

Можливі ситуації, коли потрібно, щоб **const**-функція могла змінити один або декілька членів класу, але ніяк не могла вплинути на інші. Це можна реалізувати за допомогою модифікатора **mutable**, який перевизначає атрибут функції **const**. Іншими словами, **mutable**-член може бути модифікований **const**-функцією-членом. Розглянемо такий приклад.

Код програми 11.11. Демонстрація механізму використання модифікатора mutable

```
#include <iostream>              // Для потокового введення-виведення
using namespace std;             // Використання стандартного простору імен

class Demo {
	mutable int c;
	int d;
public:
	int Put() const { return c; }       // Все гаразд

	void Set(int x) const
	{
		c = x; // тепер все гаразд
	};

	void setJ(int x) const      // Наступна функція не відкомпілюється.
	{
		d = x; // Це, як і раніше, неправильно!
	};
}

int main()
{
	Demo Obj;
	Obj.Set(1900);
	cout << Obj.Put();

	getch(); return 0;
}
```

У цьому коді програми закритий член-даних c визначено з використанням модифікатора **mutable**, тому його можна змінити за допомогою функції Set(). Проте змінна d не є **mutable**-членом, тому функції Set() не дозволяється модифікувати його значення.

11.6. Використання explicit-конструкторів

У мові програмування C++ визначено ключове слово **explicit**, яке застосовується для оброблення спеціальних ситуацій, наприклад, при використанні конструкторів певних типів. Щоби зрозуміти призначення специфікатора **explicit**, спочатку розглянемо наведену нижче програму.

Код програми 11.12. Демонстрація механізму використання специфікатора explicit

```
#include <iostream>            // Для потокового введення-виведення
using namespace std;           // Використання стандартного простору імен

class myClass {                // Оголошення класового типу
	int a;
public:
	myClass(int x) { a = x; }
	int Put() { return a; }
};

int main()
{
	myClass ObjA(4);
	cout << ObjA.Put();
	getch(); return 0;
}
```

У цьому коді програми конструктор класу myClass приймає один параметр. Зверніть увагу на те, який оголошено об'єкт ObjA у функції **main**(). Значення 4, що задається у круглих дужках після імені ObjA, є аргументом, який передається параметру x конструктора myClass(), а параметр x, своєю чергою, використовується для ініціалізації члена a. Саме таким способом ми ініціалізуємо члени класу, як це було показано на початку цього навчального посібника. Проте існує і альтернативний варіант ініціалізації. Наприклад, у процесі виконання такої настанови член класу a також набуде значення 4.

```
myClass ObjA = 4; // Цей формат ініціалізації перетвориться у формат myClass(4).
```

Як зазначено у коментарі, цей формат ініціалізації автоматично перетвориться у виклик конструктора класу myClass, а число 4 використано як аргумент. Іншими словами, попередня настанова обробляється компілятором так, як вона була записана:

```
myClass ObjA(4);
```

У загальному випадку завжди, якщо у Вас є конструктор, який приймає тільки один аргумент, то для ініціалізації змінних об'єкта можна використовувати будь-який з форматів: або ObjA(x), або ObjA = x. Йдеться про те, що при створенні конструктора класу з одним аргументом Ви опосередковано створите перетворення з типу аргумента в тип цього класу.

*Для створення "неконвертованого" конструктора використовується специфікатор **explicit**.*

Якщо програмісту не потрібно, щоб таке неявне перетворення мало місце, то можна запобігти цьому за допомогою специфікатора **explicit**. Ключове слово **explicit** застосовується тільки до конструкторів. Конструктор, визначений за допомогою специфікатора **explicit**, буде задіяний тільки у тому випадку, якщо для ініціалізації членів-даних класу використовується звичайний синтаксис конструктора. Ніяких автоматичних перетворень виконано не буде. Наприклад, оголошуючи конструктор класу myClass з використанням специфікатора **explicit**, ми, тим самим, відміняємо підтримку автоматичного перетворення типів. У цьому варіанті визначення класу функція myClass() оголошується як **explicit**-конструктор.

```
#include <iostream>             // Для потокового введення-виведення
using namespace std;            // Використання стандартного простору імен

class myClass { // Оголошення класового типу
    int a;
public:
    explicit myClass(int x) { a = x; }
    int Put() { return a; }
};
```

Тепер буде дозволено до застосування тільки конструктори, які задаються в такому форматі:

```
myClass ObjA(110);
```

Потреба неявного перетворення конструктора. Автоматичне перетворення з типу аргумента конструктора у виклик конструктора саме по собі має цікаві наслідки. Розглянемо, наприклад, такий код програми.

Код програми 11.13. Демонстрація механізму використання неявного перетворення конструктора

```
#include <iostream>             // Для потокового введення-виведення
using namespace std;            // Використання стандартного простору імен

class myClass {                 // Оголошення класового типу
    int n;
public:
    myClass(int c) { n = c; }
    int Put() { return n; }
};

int main()
{
    myClass Obj(10);

    cout << Obj.Put() << endl; // Відображає 10
    // Тепер використовуємо неявне перетворення для присвоєння нового значення.
    Obj = 1000;
    cout << Obj.Put() << endl; // Відображає 1000
    getch(); return 0;
}
```

Зверніть увагу на те, що нове значення присвоюється об'єкту Obj за допомогою такої настанови:

```
Obj = 1000;
```

Використання такого формату можливе завдяки опосередкованому перетворенню з типу **int** в тип myClass, яке створюється конструктором myClass(). Звичайно ж, якби конструктор myClass() було оголошено за допомогою специфікатора **explicit**, то попередня настанова не могла б виконатися.

11.7. Синтаксис механізму ініціалізації членів-даних класу

У прикладах програм, наведених у попередніх розділах, члени-даних набували початкові значення у конструкторах своїх класів. Наприклад, наведений нижче код програми починається з клас myClass, який містить два члени-даних a і b. Ці члени ініціалізувалися у конструкторі myClass().

Код програми 11.14. Демонстрація механізму ініціалізації членів-даних з використанням конструкторів класу

```
#include <iostream>          // Для потокового введення-виведення
using namespace std;         // Використання стандартного простору імен

class myClass {              // Оголошення класового типу
    int a, b;
public:
    // Ініціалізація a і b у конструкторі myClass(), використовуючи звичайний синтаксис.
    myClass(int x, int y) { a = x; b = y; }
    int PutA() { return a; }
    int PutB() { return b; }
};

int main()
{
    myClass ObjA(7, 9), ObjB(5, 2);
    cout << "Значення об'єкта ObjA = " << ObjA.PutB() << " і " << ObjA.PutA() << endl;
    cout << "Значення об'єкта ObjB = " << ObjB.PutB() << " і " << ObjB.PutA() << endl;
    getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Значення змінних об'єкта ObjA дорівнює 9 і 7
Значення змінних об'єкта ObjB дорівнює 2 і 5
```

Присвоєння початкових значень членам-даних a і b у конструкторі, як це робиться у конструкторі myClass(), – звичайна практика, яка застосовується для багатьох класів. Але цей метод придатний не для всіх випадків. Наприклад, якби члени a і b були задані як **const**-змінні, тобто так:

```
class myClass { // Оголошення класового типу
    const int a; // const-член
    const int b; // const-член,
```

то їм не можна було б присвоїти значення за допомогою конструктора класу myClass, оскільки **const**-змінні повинні ініціалізуватися одноразово, після чого їм вже не можна надати інші значення. Такі проблеми виникають при використанні посилальних членів, які повинні ініціалізуватися, і при використанні членів класу, які не мають конструкторів за замовчуванням. Для вирішення проблем такого роду у мові програмування С++ передбачено підтримку альтернативного синтаксису ініціалізації членів-даних класу, який дає змогу присвоювати їм початкові значення при створенні об'єкта класу.

Синтаксис ініціалізації членів-даних класу аналогічний тому, який використовують для виклику конструктора базового класу. Ось як виглядає загальний формат такої ініціалізації:

```
constructor (перелік_аргументів): член1(ініціалізація),
                                  член2(ініціалізація),
                                  //...
                                  члениN (ініціалізація)
{
    // Тіло конструктора
}
```

Члени, що підлягають ініціалізації, вказуються після конструктора класу, відокремлюються від імені конструктора і переліку його аргументів двокрапкою. При цьому в одному і тому ж переліку можна змішувати звернення до конструкторів базового класу з ініціалізацією його членів.

Нижче представлена попередня програма, але перероблена так, щоби члени a і b були оголошені з використанням модифікатора **const**, і набували свої початкові значення через альтернативний синтаксис ініціалізації членів-даних класу.

Код програми 11.15. Демонстрація механізму оголошення членів-даних класу з використанням модифікатора const

```
#include <iostream>          // Для потокового введення-виведення
using namespace std;         // Використання стандартного простору імен

class myClass { // Оголошення класового типу
    const int a, b; // const-члени
public:
    // Ініціалізація членів a і b з використанням альтернативного синтаксису ініціалізації.
    myClass(int x, int y): a(x), b(y) { }
    int PutA() { return a; }
    int PutB() { return b; }
};

int main()
{
    myClass ObjA(7, 9), ObjB(5, 2);

    cout << "Значення ObjA = " << ObjA.PutA() << " i " << ObjA.PutB() << endl;
    cout << "Значення ObjB = " << ObjB.PutA() << " i " << ObjB.PutB() << endl;
    getch(); return 0;
}
```

Ця програма відображає на моніторі такі ж результати, як і її попередня версія. Проте зверніть увагу на те, як ініціалізовано члени a і b:

```
myClass(int x, int y): a(x), b(y) { }
```

У цьому записі член даних a ініціалізувався значенням, переданим у аргументі x, а член b – значенням, переданим у аргументі y. І хоча члени a і b зараз визначені як **const**-змінні, проте вони можуть набути свої початкові значення при створенні об'єкта класу myClass, оскільки тут використовується альтернативний синтаксис ініціалізації членів-даних класу.

11.8. Використання ключового слова asm

Незважаючи на те, що мова програмування С++ – всеосяжна і потужний засіб для створення сучасних програмних продуктів, проте трапляються ситуації, оброблення яких для неї виявляється дуже скрутним[1]. Щоб справитися з подібними спеціальними ситуаціями, мова С++ надає засіб, який дає змогу увійти до коду програми, написаного мовою Асемблер, абсолютно ігноруючи С++-компілятор. Цим засобом і є настанова **asm**, використовуючи яку можна вбудувати Асемблерний код програми безпосередньо у С++-програму. Цей програмний код відкомпілюється без будь-яких змін і стане частиною коду Вашої програми, починаючи з місця знаходження настанови **asm**.

---

*За допомогою ключового слова **asm** у С++-програму вбудовується програмний код, написаний мовою Асемблер.*

Загальний формат використання ключового слова **asm** має такий вигляд:

```
asm ("код");
```

У цьому записі елемент *код* означає настанову, написану мовою Асемблер, яка буде вбудована у С++-програму. При цьому деякі компілятори також дають змогу використовувати і інші формати запису настанови **asm**:

```
asm настанова;

asm настанова newline

asm {
    послідовність настанов
}
```

У цьому записі елемент *настанова* означає будь-яку допустиму настанову мови Асемблер. Оскільки використання настанови **asm** залежить від конкретної реалізації середовища програмування, то за подробицями реалізації потрібно звернутися до документації, що додається до Вашого компілятора.

На момент написання цього навчального посібника у середовищі Visual C++ (Microsoft) для вбудовування коду програми, написаного мовою Асемблер, пропонувалося використовувати настанову __asm. У всьому іншому цей формат аналогічний опису настанови **asm**.

---

[1] Наприклад, у мові програмування С++ не передбачено настанови, яка могла б заборонити переривання.

Вартоа' нати! *Для використання настанови* **asm** *необхідно володіти доскональними знаннями мови Асемблер. Якщо Ви не вважаєте себе фахівцем з цієї мови, то краще поки що уникати використання настанови* **asm**, *оскільки необережне її застосування може спричинити важкі наслідки для Вашої операційної системи.*

11.9. Специфікатор компонування функцій

У мові програмування С++ можна визначити, як функція зв'язується з Вашою програмою. За замовчуванням функції компонуються як С++-функції. Але, використовуючи специфікацію компонування, можна забезпечити компонування функцій, написаних іншими мовами програмування. Загальний формат специфікатора компонування має такий вигляд:

extern "мова" *прототип_функції*

У цьому записі елемент *мова* означає потрібну мову програмування. Всі С++-компілятори підтримують як С-, так і С++-компонування. Деякі компілятори також дають змогу використовувати специфікатори компонування для таких мов, як Fortran, Pascal або Visual Basic[1].

Специфікатор компонування дає змогу визначити спосіб компонування функції.

Наведений нижче код програми дає змогу скомпонувати функцію cFun() як С-функцію.

Код програми 11.16. Демонстрація механізму застосування специфікатора компонування функцій

```
#include <iostream>          // Для потокового введення-виведення
using namespace std;         // Використання стандартного простору імен

extern "C" void cFun();

int main()
{
	cFun();
	getch(); return 0;
}

// Ця функція буде скомпонована як С-функція.
void cFun()
{
	cout << "Ця функція скомпонована як С-функція" << endl;
}
```

Нео! хідноaпам'ятати! *Ключове слово* **extern** *– необхідна складова специфікації компонування. Понад це, специфікація компонування повинна бути глобальною; її не можна використовувати в тілі якої-небудь функції.*

[1] Цю інформацію необхідно уточнити у документації, яка додається до Вашого компілятора.

Використовуючи наступний формат специфікації компонування, можна задати не одну, а відразу декілька функцій.

```
extern "мова" {
	прототипи_функцій
}
```

Специфікації компонування використовуються досить рідко, і Вам, можливо, ніколи не доведеться їх застосовувати. Основне їх призначення – дати змогу застосування у С++-програмах кодів програм, написаних іншими організаціями мовами, відмінними від мови С++.

11.10. Оператори вказання на члени класу ".*" і "->"

У мові програмування С++ передбачено можливість згенерувати покажчик спеціального типу, який "посилається" не на конкретний примірник члена в об'єкті, а на члена класу взагалі. Покажчик такого типу називається *покажчиком на члена класу* (pointer-to-member). Це – не звичайний С++-покажчик. Цей спеціальний покажчик забезпечує тільки відповідний зсув у об'єкті, який дає змогу виявити потрібного члена класу. Оскільки покажчики на члени – не справжні покажчики, то до них не можна застосовувати оператори "." і "->". Для отримання доступу до члена класу через покажчик на член необхідно використовувати спеціальні оператори ".*" і "->".

Оператори вказання на члени класу дають змогу отримати доступ до члена класу через покажчик на цього члена.

Якщо ідея, яку викладено у попередньому абзаці, Вам видалася трохи "невиразною", то наведений нижче приклад допоможе її з'ясувати. У процесі виконання цієї програми відображається сума чисел від 1 до 7. Тут доступ до членів класу myClass(функції Sum() і змінної sum) реалізується шляхом використання покажчиків на члени.

Код програми 11.17. Демонстрація механізму використання покажчиків на члени класу (початкова версія)

```
#include <iostream>             // Для потокового введення-виведення
using namespace std;            // Використання стандартного простору імен

class myClass { // Оголошення класового типу
public:
	int sum;
	void myClass::Sum(int x);
};

void myClass::Sum(int x)
{
	sum = 0;
	for(int i=x; i; i--) sum += i;
}
```

```
int main()
{
	int myClass::*dp;             // Покажчик на int-члена класу
	void (myClass::*fp)(int x); // Покажчик на функцію-члена

	myClass ObjC;

	dp = myClass::sum;          // Отримуємо адресу члена даних
	fp = &myClass::Sum;         // Отримуємо адресу функції-члена класу

	(ObjC.*fp)(7);        // Обчислюємо суму чисел від 1 до 7
	cout << "Сума чисел від 1 до 7 дорівнює " << ObjC.*dp;

	getch(); return 0;
}
```

Результат виконання цієї програми є таким:

```
Сума чисел від 1 до 7 дорівнює 28
```

У функції **main**() створюється два члени-покажчики: dp (для вказівки на змінну sum) і fp (для вказівки на функцію Sum()). Зверніть увагу на синтаксис кожного оголошення. Для уточнення класу використовують оператор дозволу контексту (оператор дозволу області видимості). Програма також створює об'єкт типу myClass з іменем ObjC.

Потім програма отримує адреси змінної sum і функції Sum() і присвоює їх покажчикам відповідно dp і fp. Як уже зазначалося вище, ці адреси насправді є тільки зсувами в об'єкті типу myClass, згідно з якими можна знайти змінну sum і функцію Sum(). Потім програма використовує покажчик на функцію fp, щоб викликати функцію Sum() для об'єкта ObjC. Наявність додаткових круглих дужок пояснюється необхідністю коректного застосування оператора ".*". Нарешті, програма відображає значення суми чисел, отримуючи доступ до змінної sum об'єкта ObjC через покажчик dp.

Під час доступу до члена об'єкта за допомогою об'єкта або посилання на нього необхідно використовувати оператор ".*". Але, якщо для цього використовується покажчик на об'єкт, то потрібно використовувати оператор "->*", як показано у цій версії попереднього коду програми.

Код програми 11.18. Демонстрація механізму використання покажчиків на члени класу (модифікована версія)

```
#include <iostream>          // Для потокового введення-виведення
using namespace std;         // Використання стандартного простору імен

class myClass { // Оголошення класового типу
public:
	int sum;
	void myClass::Sum(int x);
};

void myClass::Sum(int x)
{
	sum = 0;
```

```
        for(int i=x; i; i--) sum += i;
    }

    int main()
    {
        int myClass::*dp;            // Покажчик на int-члена класу
        void (myClass::*fp)(int x); // Покажчик на функцію-члена
        myClass *c, ObjD;           // Член с зараз -- покажчик на об'єкт

        c = &ObjD;                  // Присвоюємо покажчику с адресу об'єкта
        dp = &myClass::sum;         // Отримуємо адресу члена даних sum
        fp = &myClass::Sum;         // Отримуємо адресу функції Sum()

        (c->*fp)(7);        // Тепер використовуємо оператор "->*" для виклику функції Sum().

        cout << "Сума чисел від 1 до 7 дорівнює " << c->*dp; // ->*
        getch(); return 0;
    }
```

У цій версії змінна c оголошується як покажчик на об'єкт типу myClass, а для доступу до члена даних sum і функції-члена Sum() використовують оператор "->*".

Нео! хідноапам'ятати! *Оператори вказання на члени класу призначені для спеціальних випадків, тому їх не варто використовувати для розв'язання звичайних повсякденних задач програмування.*

11.11. Створення функцій перетворення типів

Іноді виникає потреба в одному виразі об'єднати створений програмістом клас з даними інших типів. Хоча перевизначені операторні функції можуть забезпечити використання змішаних типів даних, у деяких випадках все ж таки можна обійтися простим перетворенням типів. І тоді, щоб перетворити клас у тип, сумісний з типом решти частини виразу, потрібно використовувати функцію перетворення типу. Загальний формат функції перетворення типу має такий вигляд:

```
operator type() {return value; }
```

У цьому записі елемент type – новий тип, який є метою нашого перетворення, а елемент *value* – значення після перетворення. Функція перетворення повинна бути членом класу, для якого вона визначається.

Функція перетворення автоматично перетворить тип класу в інший тип.

Щоб проілюструвати створення функцій перетворення, скористаємося класом kooClass ще раз. Припустимо, що нам потрібно мати засіб перетворення об'єкта типу kooClass у цілочисельне значення, яке можна використовувати в цілочисельному виразі. Понад це, таке перетворення повинно відбуватися з використанням добутку значень трьох координат. Для реалізації цього будемо використовувати функцію перетворення, яка має такий вигляд:

```
operator int() { return x * y * z; }
```

Тепер розглянемо код програми, яка ілюструє роботу функції перетворення.

Код програми 11.19. Демонстрація механізму використання функції перетворення типів

```
#include <iostream>           // Для потокового введення-виведення
using namespace std;          // Використання стандартного простору імен

class kooClass {// Оголошення класового типу
	int x, y, z; // Тривимірні координати
public:
	kooClass(int a, int b, int c) {x = a; y = b; z = c; }

	kooClass operator+(kooClass obj);
	friend ostream &operator<<(ostream &stream, kooClass &obj);
	operator int() {return x * y * z; }
};

// Відображення тривимірних координат x, y, z – функція виведення даних
// для класу kooClass.
ostream &operator<<(ostream &stream, kooClass &obj)
{
	stream << obj.x << ", ";
	stream << obj.y << ", ";
	stream << obj.z << endl;
	return stream; // Повертає посилання на параметр stream
}

kooClass kooClass::operator+(kooClass obj)
{
	kooClass tmp(0, 0, 0);

	tmp.x = x + obj.x;
	tmp.y = y + obj.y;
	tmp.z = z + obj.z;
	return tmp; // Повертає модифікований тимчасовий об'єкт
}

int main()
{
	kooClass ObjA(1, 2, 3), ObjA(2, 3, 4);

	cout << ObjA << ObjA;
	cout << ObjA+100;        // Відображає число 124, оскільки тут здійснюється
	                         // перетворення об'єкта класу у значення типу int.
	cout << endl;
	ObjA = ObjA + ObjA;      // Додавання двох об'єктів класу kooClass
	                         // виконується без перетворення типу.
	cout << ObjA;            // Відображає координати 3, 5, 7
	getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
1,2, 3
2, 3, 4
124
3, 5, 7
```

Як підтверджують результати виконання цієї програми, якщо в такому виразі цілочисельного типу, як **cout** << ObjA+100, використовується об'єкт типу kooClass, то до цього об'єкта застосовується функція перетворення. У цьому випадку функція перетворення повертає значення 24, яке потім бере участь в операції додавання з числом 100. Але коли у перетворенні немає потреби, наприклад, під час обчислення виразу ObjA = ObjA + ObjA, то функція перетворення не викликається.

Якщо функція перетворення створена, то вона викликатиметься скрізь, де потрібне перетворення, враховуючи ситуації, коли об'єкт передається функції як аргумент. Наприклад, якщо об'єкт класу kooClass передати стандартній функції **abs**(), то також буде викликано функцію, що здійснює перетворення об'єкта типу kooClass у значення типу **int**, оскільки функція **abs**() повинна приймати аргумент цілочисельного типу.

Нео! хідноапам'ятати!а*Для різних ситуацій можна створювати різні функції перетворення. Наприклад, можна визначити функції, які перетворять об'єкти типу* kooClass *у значення типу* ***double*** *або* ***long****, при цьому створені функції будуть застосовуватися автоматично. Функції перетворення дають змогу інтегрувати нові типи класів, що створюються програмістом, у С++-середовище програмування.*

Розділ 12. ВВЕДЕННЯ В СТАНДАРТНУ БІБЛІОТЕКУ ШАБЛОНІВ

Матеріал цього розділу містить інформацію про те, який за останні роки є найважливішим доповненням у мові програмування C++. Йдеться про стандартну бібліотеку шаблонів (Standard Template Library – STL). Саме внесення бібліотеки STL у мову програмування C++ було основною подією, яка наполегливо обговорювалася у період стандартизації мови C++. Бібліотека STL надає користувачу шаблонні класи і функції загального призначення, які реалізують багато популярних і часто використовуваних алгоритмів і структур даних. Наприклад, вона містить підтримку *векторів*, *списків*, *черг* і *стеків*, а також визначає різні функції, які забезпечують до них доступ. Оскільки бібліотека STL складається з шаблонних класів, то алгоритми і структури даних можна застосовувати до даних практично будь-якого типу.

Бібліотека STL – це результат розроблення стандартного програмного забезпечення, який увібрав у себе одні з найскладніших засобів мови програмування C++. Щоб розуміти вміст бібліотеки STL і уміти ним користуватися, необхідно досконало засвоїти весь матеріал, викладений у попередніх розділах. Особливо добре потрібно орієнтуватися в шаблонах. Відверто кажучи, шаблонний синтаксис, який використовує бібліотека STL, може спочатку налякати програмістів-початківців, хоча він виглядає дещо складнішим, ніж є насправді. Незважаючи на те, що матеріал цього розділу є не набагато важчим за будь-який інший, наведений у цьому навчальному посібнику, проте не варто засмучуватися від того, якщо щось на перший погляд Вам видасться незрозумілим. Трохи терпіння під час перегляду прикладів і певної уваги у процесі їх детального аналізу приведе до того, що незабаром Ви зрозумієте, як за незвичним синтаксисом ховається строга простота бібліотеки STL.

STL – достатньо велика бібліотека, і всі її засоби неможливо викласти в одному розділі. Повний опис бібліотеки STL зі всіма її особливостями і методами програмування займає цілу книгу. Мета представленого тут матеріалу – познайомити студентів з основними операціями, принципами проектування і основами STL-програмування. Засвоївши матеріал цього розділу, студент зможете самостійно легко вивчити іншу частину бібліотеки STL.

У цьому розділі також описано ще один важливий клас C++ – клас **string**. Він призначений для визначення рядкового типу даних, який дає змогу обробляти символьні рядки подібно до того, як обробляються дані інших типів за допомогою операторів. Клас **string** тісно пов'язаний з бібліотекою STL.

12.1. Огляд стандартної бібліотеки шаблонів

Незважаючи на великий розмір стандартної бібліотеки шаблонів (STL) та інколи лякливий синтаксис, насправді її засоби достатньо легко використовувати,

якщо зрозуміти, як вона побудована і з яких елементів складається. Тому, перш ніж переходити до перегляду прикладів, познайомимося з основними складовими бібліотеки STL.

Ядро стандартної бібліотеки шаблонів містить три основні елементи: *контейнери, алгоритми* і *ітератори.* Вони працюють спільно один з іншим, надаючи тим самим користувачу готові розв'язки різних задач програмування.

Контейнери *– це об'єкти, які містять інші об'єкти.*

Існує декілька різних типів контейнерів. Наприклад, клас **vector** визначає динамічний масив, клас queue створює двосторонню чергу, а клас **list** забезпечує роботу з лінійним списком. Ці контейнери називають *послідовними контейнерами* і є базовими в бібліотеці STL. Окрім базових, бібліотека STL визначає *асоціативні контейнери,* які дають змогу ефективно знаходити потрібні значення на основі заданих ключових значень (ключів). Наприклад, клас **map** забезпечує зберігання пар "ключ-значення" і надає змогу знаходити потрібне значення за заданим унікальним ключем.

Кожен контейнерний клас визначає набір функцій, які можна застосовувати для роботи з ним. Наприклад, контейнер списку містить функції, призначені для виконання вставлення, видалення і об'єднання елементів. А стек містить функції, які дають змогу поміщати значення у стек і витягувати їх із нього.

Алгоритми *обробляють вміст контейнерів.*

Їх можливості містять засоби ініціалізації, сортування, пошуку і перетворення вмісту контейнерів. Багато алгоритмів працюють із заданим *діапазоном* елементів контейнера.

Ітератори – це об'єкти, які тією чи іншою мірою діють подібно до покажчиків. Вони дають змогу циклічно здійснювати запит до вмісту контейнера практично так само, як це робиться за допомогою покажчика під час циклічного доступу до елементів масиву. Існує п'ять типів ітераторів.

Табл. 12.1. Типи ітераторів

Ітератори	Опис
Довільного доступу (**random access**)	Зберігають і зчитують значення; дають змогу організувати довільний доступ до елементів контейнера
Двонаправлені (**bi-directional**)	Зберігають і зчитують значення; забезпечують інкрементно-декрементне переміщення
Однонаправлені (**forward**)	Зберігають і зчитують значення; забезпечують тільки інкрементне переміщення елементами контейнера
Вхідні (**GetPut**)	Зчитують, але не записують значення; забезпечують тільки інкрементне переміщення елементами контейнера
Вихідні (**output**)	Записують, але не зчитують значення; забезпечують тільки інкрементне переміщення елементами контейнера

У загальному випадку ітератор з більшими можливостями доступу, можна використовувати замість ітератора з меншими можливостями. Наприклад, однонаправлений ітератор може замінити вхідний ітератор.

Ітератори обробляються аналогічно до покажчиків. Їх можна інкрементувати і декрементувати. До них можна застосовувати оператор перейменування адреси *. Ітератори оголошуються за допомогою типу **iterator**, який визначається різними контейнерами.

Бібліотека STL підтримує *реверсивні ітератори,* які є або двонаправленими, або ітераторами довільного доступу, даючи змогу переміщатися елементами послідовності у зворотному порядку. Отже, якщо реверсивний ітератор вказує на кінець послідовності, то після інкрементування він вказуватиме на елемент, розташований перед кінцем послідовності.

Під час посилання на різні типи ітераторів у описах шаблонів у цьому навчальному посібнику буде використано такі терміни:

Термін	Наданий ітератор
BiIter	Двонаправлений
ForIter	Однонаправлений
InIter	Вхідний
OutIter	Вихідний
RandIter	Довільного доступу

Бібліотека STL опирається не тільки на контейнери, алгоритми і ітератори, але і на інші стандартні компоненти. Основними з них є *розподільники пам'яті*, *предикати* і *функції порівняння*.

Ро' подільникапам'яті *керує виділенням пам'яті для контейнера.*

Кожен контейнер має свій *розподільник пам'яті* (***allocator***). Розподільники керують виділенням пам'яті під елементи контейнера. Стандартний розподільник – це об'єкт класу **allocator**, але у разі потреби (у спеціалізованих додатках) можна визначати власні розподільники. Здебільшого стандартного розподільника є цілком достатньо для реалізації програмних задумів.

Предикат *повертає як результат значення ІСТИНА/ФАЛЬШ.*

Деякі алгоритми і контейнери використовують спеціальний тип функції, так званий *предикат* (*predicate*). Існує два варіанти предикатів: унарний і бінарний. Унарний предикат приймає один аргумент, а бінарний – два. Ці функції повертають значення ІСТИНА/ФАЛЬШ, але точні умови, які примусять їх повернути дійсне або помилкове значення, визначаються програмістом. В іншій частині цього розділу, коли знадобиться унарна функція-предикат, на це буде вказано за допомогою типу **UnPred**. У разі потреби використання бінарного предиката застосовуватиметься тип **BinPred**. У бінарному предикаті аргументи завжди розташовані у порядку *перший, другий* стосовно функції, яка викликає цей предикат. Як для унарного, так і для бінарного предикатів аргументи повинні містити значення, тип яких збігається з типом об'єктів, які зберігає цей контейнер.

Функціїапорівняння *порівнюють два елементи послідовності.*

Деякі алгоритми і класи використовують спеціальний тип бінарного предиката, який порівнює два елементи. Функції порівняння повертають значення **true**, як-

що їх перший аргумент менший від другого. Функції порівняння ідентифікуються за допомогою типу **Comp**.

Крім заголовків, потрібних різним класам STL, стандартна бібліотека C++ містить заголовки **<utility>** і <**functional**>, які забезпечують підтримку STL. Наприклад, у заголовку **<utility>** визначається шаблонний клас **pair**, який може зберігати пару значень. Будемо використовувати клас **pair** нижче у цьому розділі.

Шаблони у заголовку <**functional**> дають змогу створювати об'єкти, які визначають функцію **operator**(), називаються *об'єктами-функціями*. Їх у багатьох випадках можна використовувати замість покажчиків на функції. Існує декілька вбудованих об'єктів-функцій, оголошених у заголовку <**functional**>. Нижче наведено деякі з них:

plus	minus	multiplies	divides	modulus
negate	equal_to	not_equal_to	greater	greater_equal
less	less_equal	logical_and	logical_or	logical_not

Можливо, найширше використовується об'єкт-функція **less**, який визначає, за яких умов один об'єкт менший від іншого. Об'єкти-функції можна використовувати замість реальних покажчиків на функції в алгоритмах бібліотеки STL, про які піде мова нижче. Використовуючи об'єкти-функції замість покажчиків на функції, бібліотека STL у деяких випадках генерує ефективніший програмний код.

Матеріал цього розділу не передбачає використання об'єктів-функцій, тому ми не застосовуватимемо їх безпосередньо. Детальний опис об'єктів-функцій можна знайти в книзі Герберта Шілдта "*Полный справочник по C++"*.

12.2. Поняття про контейнерні класи

Як уже зазначалося вище, контейнери є об'єктами бібліотеки STL, основне призначення яких – зберігання даних різних типів. Контейнери, які визначаються в бібліотеці STL, представлено в табл. 12.1. У ній також вказано заголовки, які необхідно помістити у програму під час використання кожного контейнера. Незважаючи на те, що клас **string** також є контейнером, який дає змогу зберігати і обробляти символьні рядки, однак він у цю таблицю не внесений і розглядається нижче у цьому розділі.

Табл. 12.1. Контейнери, визначені в бібліотеці STL

Контейнер	Опис	Заголовок
1	2	3
bitset	Набір бітів	**<bitset>**
deque	Дек (двобічна черга, або черга з двобічним доступом)	**<deque>**
list	Лінійний список	**<list>**
map	Відображення, зберігає пари "ключ-значення", в яких кожен ключ пов'язаний тільки з одним значенням	**<map>**
multimap	Мультивідображення, зберігає пари "ключ-значення", в яких кожен ключ може бути пов'язаний з двома або більше значеннями	**<map>**

1	2	3
multiset	Множина, у якій кожен елемент необов'язково унікальний (мультимножина)	**<set>**
priority_queue	Пріоритетна черга	**<queue>**
queue	Черга	**<queue>**
set	Множина, у якій кожен елемент унікальний	**<set>**
stack	Стек	**<stack>**
vector	Динамічний масив	**<vector>**

Оскільки імена типів у оголошеннях шаблонних класів довільні, контейнерні класи оголошують **typedef**-версії цих типів, що конкретизує імена типів. Деякі з найбільш популярних **typedef**-імен мають таке призначення:

size_type	Певний цілий тип, приблизно аналогічний типу **size_t**
reference	Посилання на елемент
const_reference	Константне (**const**-) посилання на елемент
iterator	Ітератор
const_iterator	Константний (**const**-) ітератор
reverse_iterator	Реверсивний ітератор
const_reverse_iterator	Константний реверсивний ітератор
value_type	Тип значення, яке зберігається у контейнері (те саме, що і узагальнений тип myType)
allocator_type	Тип розподільника (пам'яті)
key_type	Тип ключа
key_compare	Тип функції, яка порівнює два ключі
mapped_type	Тип значення, яке зберігається у відображенні (те саме, що і узагальнений тип myType)
value_compare	Тип функції, яка порівнює два значення

Оскільки в одному розділі неможливо розглянути всі контейнери, то в наступних розділах ми подамо тільки три з них: **vector**, **list** і **map**. Якщо Ви зрозумієте, як працюють ці три контейнери, у Вас не буде проблем з використанням інших.

12.3. Механізми роботи з векторами

Одним з найпоширеніших контейнерів загального призначення є вектор. Аналогічний клас **vector** підтримує динамічний масив, який у разі потреби може збільшувати (зменшувати) свій розмір. Як уже зазначалося вище, у мові програмування C++ розмір масиву фіксується при компілюванні. І хоча це найефективніший спосіб реалізації масивів, він водночас є і обмежувачем, оскільки розмір масиву не можна змінювати у процесі виконання програми. Ця проблема розв'язується за допомогою вектора, який у міру потреби забезпечує виділення додаткового об'єму пам'яті. Незважаючи на те, що вектор – це динамічний масив, проте, для доступу до його елементів можна використовувати стандартне позначення індексації елементів масивів.

Вектори *є динамічними масивами.*

Ось як виглядає шаблонна специфікація для класу **vector**:

```
template <class myType, class Allocator = allocator<myType>> class vector
```

У цьому записі myType – тип даних, який зберігається, а елемент **Allocator** означає розподільник пам'яті, який за замовчуванням використовує стандартний розподільник. Клас **vector** має такі конструктори:

```
explicit vector(const Allocator &a = Allocator());

explicit vector(size_type num, const myType &val = myType(), const Allocator &a = Allocator());

vector(const vector<myType, Allocator> &ob);

template <class InIter> vector(InIter start, InIter end, const Allocator &a = Allocator());
```

Перша форма конструктора призначена для створення порожнього вектора. Друга створює вектор, який містить *num* "елементів із значенням *val*", причому значення *val* може бути встановлено за замовчуванням. Третя форма дає змогу створити вектор, який містить ті самі елементи, що і заданий вектор *ob*. Четверта призначена для створення вектора, який містить елементи у діапазоні, заданому параметрами-ітераторами *start* і *end*.

Заради досягнення максимальної гнучкості та переносності будь-який об'єкт, який призначено для зберігання у векторі, повинен визначатися конструктором за замовчуванням. Окрім цього, він повинен визначати операції "<" і "==". Деякі компілятори можуть зажадати визначення і інших операторів порівняння. (З причини існування різних реалізацій для отримання точної інформації про вимоги, що пред'являються Вашим компілятором, необхідно звернутися до документації, що додається до нього.) Всі вбудовані типи даних автоматично задовольняють ці вимоги.

Незважаючи на те, що наведений вище синтаксис шаблону виглядає достатньо "масивним", у оголошенні вектора немає нічого складного. Розглянемо декілька прикладів:

```
vector<int> iv;            // Створення вектора нульової довжини для зберігання int-значень.
vector<char> cv(5);        // Створення 5-елементного вектора для зберігання char-значень.
vector<char> cv(5, 'x');   // Ініціалізація 5-елементного char-вектора.
vector<int> iv2(iv);       // Створення int-вектора на основі int-вектора iv.
```

Для класу **vector** визначено такі оператори порівняння: ==, <, <=, !=, > і >=.

Для вектора також визначено оператор індексації елементів масиву "[]", який дає змогу отримати доступ до своїх елементів за допомогою стандартного запису з використанням індексів. Функції-члени, визначені у класі **vector**, перераховані в табл. 12.2. Найважливішими з них є **size**(), **begin**(), **end**(), **push_back**(), **insert**() і **erase**(). Дуже корисна функція **size**(), яка повертає поточний розмір вектора, оскільки вона дає змогу визначити розмір вектора у процесі виконання програми.

Нео! хідноапам'ятати! *Вектори у разі потреби збільшують свій розмір, тому потрібно мати можливість визначати його величину під час роботи коду програми, а не тільки при компілюванні.*

Функція **begin**() повертає ітератор, який вказує на початок вектора. Функція **end**() повертає ітератор, який вказує на кінець вектора. Як уже пояснювалося вище, ітератори подібні до покажчиків, і за допомогою функцій **begin**() і **end**() можна отримати ітератори початку і кінця вектора відповідно.

Табл. 12.2. Функції-члени, визначені у класі vector

Функція-член	Опис
1	2
template <class InIter> void assign(InIter *start*, InIter *end*);	Поміщає у вектор послідовність, що визначається параметрами *start* і *end*
void assign(size_type *num*, const myType &*val*);	Поміщає у вектор *num* елементів із значенням *val*
reference at(size_type *i*); const_reference at(size_type *i*) const;	Повертає посилання на елементи, які задаються параметром *i*
reference back(); const_reference back() const;	Повертає посилання на останній елемент у векторі
iterator begin(); const_iterator begin() const;	Повертає ітератор для першого елемента у векторі
size_type capacity() const;	Повертає поточну місткість вектора, або кількість елементів, яка може зберігатися у векторі до того, як виникне потреба у виділенні додаткової пам'яті
void clear();	Видаляє всі елементи з вектора
bool empty() const;	Повертає істинне значення, якщо використовуваний вектор є порожнім, і помилкове значення – інакше
const_iterator end() const; iterator end();	Повертає ітератор для кінця вектора
iterator erase(iterator *i*);	Видаляє елемент, який адресується ітератором *i*; повертає ітератор для елемента, розташованого після видаленого
iterator erase(iterator *start*, iterator *end*);	Видаляє елементи у діапазоні, що задається параметрами *start* і *end*; повертає ітератор для елемента, розташованого за останнім видаленим елементом
reference front(); const_reference front() const;	Повертає посилання на перший елемент у векторі
allocator_type get_allocator() const;	Повертає розподільник пам'яті вектора
iterator insert(iterator *i*, const myType &*val* = myType());	Вставляє значення *val* безпосередньо перед елементом, заданим параметром *i*; повертає ітератор для цього елемента
void insert(iterator *i*, size_type *num*, const myType &*val*);	Вставляє *num* копій значення *val* безпосередньо перед елементом, заданим параметром *i*
template <class InIter> void insert(iterator *i*, InIter *start*, InIter *end*);	Вставляє у вектор послідовність елементів, що визначаються параметрами *start* і *end*, безпосередньо перед елементом, заданим параметром *i*
size_type max_size() const;	Повертає максимальну кількість елементів, яку може містити вектор
reference operator[](size_type *i*) const; const_reference operator[](size_type *i*) const;	Повертає посилання на елементи, які задаються параметром *i*
void pop_back();	Видаляє останній елемент у векторі
void push_back(const myType &*val*);	Додає у кінець вектора елемент, значення якого задане параметром *val*
reverse_iterator rbegin(); const_reverse_iterator rbegin() const;	Повертає реверсивний ітератор для кінця вектора
reverse_iterator rend();	Повертає реверсивний ітератор для початку век-

1	2
const_reverse_iterator rend() **const**;	тора
void reserve(**size_type** *num*);	Встановлює місткість вектора, що дорівнює значенню, не меншому від заданого *num*
void resize(**size_type** *num*, myType *val* = myType());	Встановлює розмір вектора, що дорівнює значенню, заданому параметром *num*. Якщо вектор для цього потрібно подовжити, то в його кінець додаються елементи із значенням, що задається параметром *val*
size_type size() **const**;	Повертає поточну кількість елементів у векторі
void swap(**deque**<myType, **Allocator**> &*ob*);	Здійснює обмін елементами викличного вектора і вектора *ob*

Функція **push_back**() поміщає задане значення у кінець вектора. У разі потреби довжина вектора збільшується так, щоб він міг прийняти новий елемент. За допомогою функції **insert**() можна додавати елементи у середину вектора. Окрім цього, елементи вектора можна ініціалізувати. У будь-якому випадку, якщо вектор містить елементи, то для доступу до них і їх модифікації можна використовувати засіб індексації елементів масивів. А за допомогою функції **erase**() можна видаляти елементи з вектора.

Розглянемо короткий приклад, який ілюструє базову поведінку вектора.

Код програми 12.1. Демонстрація механізму базової поведінки вектора

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
#include <vector>            // Для роботи з контейнерним класом "Вектор"
using namespace std;         // Використання стандартного простору імен

int main()
{
     vector<int> vek;  // Створення вектора нульової довжини

     // Відображаємо початковий розмір вектора vek.
     cout << "Розмір = " << vek.size() << endl;

     /* Поміщаємо значення у кінець вектора, і розмір
     вектора за потреби збільшуватиметься. */
     for(int i=0; i<10; i++) vek.push_back(i);

     // Відображаємо поточний розмір вектора vek.
     cout << "Новий розмір = " << vek.size() << endl;

     // Відображаємо вміст вектора.
     cout << "Поточний вміст:" << endl;
     for(int i=0; i<vek.size(); i++) cout << vek[i] << " ";
     cout << endl;

     /* Поміщаємо у кінець вектора нові значення, і розмір
     вектора за потреби збільшуватиметься. */
     for(int i=0; i<10; i++) vek.push_back(i + 10);
```

```
        // Відображаємо поточний розмір вектора vek.
        cout << "Новий розмір = " << vek.size() << endl;

        // Відображаємо новий вміст вектора.
        cout << "Новий вміст:" << endl;
        for(int i=0; i<vek.size(); i++) cout << vek[i] << " ";
        cout << endl;

        // Змінюємо вміст вектора.
        for(int i=0; i<vek.size(); i++) vek[i] = vek[i] + vek[i];

        // Відображаємо вміст вектора.
        cout << "Вміст подвоєно:" << endl;
        for(int i=0; i<vek.size(); i++) cout << vek[i] << " ";
        cout << endl;

        getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Розмір = 0
Новий розмір = 10
Поточний вміст:
0 1 2 3 4 5 6 7 8 9
Новий розмір = 20
Новий вміст:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Вміст подвоєно:
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38
```

Розглянемо уважно код цієї програми. У функції **main**() створюється вектор vek для зберігання **int**-елементів. Оскільки під час його створення не було передбачено ніякої ініціалізації, то вектор vek вийшов порожнім, а його місткість дорівнює нулю. Іншими словами, ми створили вектор нульової довжини. Це підтверджується викликом функції-члена класу **size**(). Потім, використовуючи функцію-члена **push_back**(), у кінець цього вектора ми поміщаємо 10 елементів, що примушує вектор збільшитися в розмірі, щоб розмістити нові елементи. Тепер розмір вектора дорівнює 10 елементам. Зверніть увагу на те, що для відображення вмісту вектора vek використовується стандартний запис індексації елементів масивів. Після цього у вектор додаються ще 10 елементів, і вектор vek автоматично збільшується в розмірі, щоб і їх прийняти на зберігання. Нарешті, використовуючи знову-таки стандартний запис індексації елементів масивів, ми змінюємо значення елементів вектора vek.

Зверніть також увагу на те, що для керування циклами, які використовуються для відображення вмісту вектора vek і його модифікації, як ознака їх завершення застосовується значення розміру вектора, що отримується за допомогою функції vek.**size**(). Одна з переваг векторів перед масивами полягає у тому, що у нас є змога дізнатися поточний розмір вектора, який у певних ситуаціях є дуже корисним засобом організації різних баз даних.

12.3.1. Доступ до елементів вектора за допомогою ітератора

Як уже зазначалося вище, масиви і покажчики у мові програмування С++ тісно пов'язані між собою. До елементів масиву можна отримати доступ як за допомогою індексу, так і за допомогою покажчика. У бібліотеці STL аналогічний зв'язок існує між векторами й ітераторами. Це означає, що до членів вектора можна звертатися як за допомогою індексу, так і за допомогою ітератора. Цю можливість продемонстровано у наведеному нижче коді програми.

Код програми 12.2. Демонстрація механізму доступу до елементів вектора за допомогою ітератора

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
#include <vector>            // Для роботи з контейнерним класом "Вектор"
using namespace std;         // Використання стандартного простору імен

int main()
{
	vector<char> vek; // Побудова вектора нульової довжини

	// Поміщаємо значення у вектор.
	for(int i=0; i<10; i++) vek.push_back('A' + i);

	// Отримуємо доступ до вмісту вектора за допомогою індексу.
	for(int i=0; i<10; i++) cout << vek[i] << " ";
	cout << endl;

	// Отримуємо доступ до вмісту вектора за допомогою ітератора.
	vector<char>::iterator p = vek.begin();
	while(p != vek.end()) {
		cout << *p << " ";
		p++;
	}

	getch(); return 0;
}
```

Ось як виглядають результати виконання цієї програми:

```
ABCDEFGHIJ
ABCDEFGHIJ
```

У цьому коді програми спочатку створюється вектор нульової довжини. Потім за допомогою функції **push_back**() у кінець вектора поміщаються символи, внаслідок чого розмір вектора відповідно збільшується.

Зверніть увагу на те, як оголошується ітератор p. Тип цього ітератора визначається контейнерними класами. Тому для отримання ітератора для конкретного контейнера використовуйте оголошення, аналогічне показаному у цьому прикладі: просто вкажіть для цього ітератора ім'я контейнера. У нашій програмі ітератор p ініціалізувався так, щоб він указував на початок вектора (за допомогою функції-члена класу **begin**()). Ітератор, який повертає ця функція, можна потім використовувати для поелементного доступу до вектора, інкрементуючи його відповідно.

Цей процес відбувається аналогічно тому, як можна використовувати покажчик для доступу до елементів масиву. Щоб визначити, коли буде досягнуто кінець вектора, використовується функція-член **end**(). Ця функція повертає ітератор, встановлено за останнім елементом вектора. Тому, якщо значення p дорівнює vek.**end**(), то кінець вектора досягнуто.

12.3.2. Вставлення та видалення елементів з вектора

Окрім занесення нових елементів у кінець вектора, у нас є можливість вставляти елементи у середину вектора, використовуючи функцію **insert**(). Видаляти елементи можна за допомогою функції **erase**(). Використання функцій **insert**() і **erase**() продемонстровано у наведеному нижче коді програми.

Код програми 12.3. Демонстрація механізму вставлення елементів у вектор і видалення їх з нього

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
#include <vector>            // Для роботи з контейнерним класом "Вектор"
using namespace std;         // Використання стандартного простору імен

int main()
{
    vector<char> vek; // Побудова вектора нульової довжини

    // Поміщаємо значення у вектор.
    for(int i=0; i<10; i++) vek.push_back('A' + i);

    // Відображаємо початковий вміст вектора.
    cout << "Розмір = " << vek.size() << endl;
    cout << "Початковий вміст вектора:" << endl;
    for(int i=0; i<vek.size(); i++) cout << vek[i] << " ";
    cout << endl;

    // Отримуємо доступ до вмісту вектора за допомогою ітератора.
    vector<char>::iterator p = vek.begin();
    p += 2; // Покажчик на 3-й елемент вектора

    // Вставляємо 10 символів 'x' у вектор vek.
    vek.insert(p, 10, 'x');

    // Відображаємо вміст вектора після вставлення символів.
    cout << "Розмір вектора після вставлення = " << vek.size() << endl;
    cout << "Вміст вектора після вставлення:" << endl;
    for(int i=0; i<vek.size(); i++) cout << vek[i] << " ";
    cout << endl;

    // Видалення вміщених елементів.
    p = vek.begin();
    p += 2; // Покажчик на 3-й елемент вектора
    vek.erase(p, p+10); // Видаляємо 10 елементів підряд.
```

```
    // Відображаємо вміст вектора після видалення символів.
    cout << "Розмір вектора після видалення символів = " << vek.size() << endl;
    cout << "Вміст вектора після видалення символів:" << endl;
    for(int i=0; i<vek.size(); i++) cout << vek[i] << " ";
    cout << endl;

    getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Розмір = 10
Початковий вміст вектора:
A B C D E F G H I J

Розмір вектора після вставлення = 20
Вміст вектора після вставлення:
A B x x x x x x x x x x C D E F G H I J

Розмір вектора після видалення символів = 10
Вміст вектора після видалення символів:
A B C D E F G H I J
```

12.3.3. Збереження у векторі об'єктів класу

У попередніх прикладах вектори слугували для зберігання значень тільки вбудованих типів, але цим їх можливості не обмежуються. У вектор можна поміщати об'єкти будь-якого типу, в т.ч. об'єкти класів, які створюються програмістом. Розглянемо приклад, у якому вектор використовують для зберігання об'єктів класу kooClass. Зверніть увагу на те, що у цьому класі визначаються конструктор за замовчуванням і перевизначені версії операторів "<" і "==". Майте на увазі, що, можливо, Вам доведеться визначити і інші оператори порівняння. Це залежить від того, як використовуваний Вами компілятор реалізує бібліотеку STL.

Код програми 12.4. Демонстрація механізму зберігання у векторі об'єктів класу

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
#include <vector>            // Для роботи з контейнерним класом "Вектор"
using namespace std;         // Використання стандартного простору імен

class kooClass {// Оголошення класового типу
    int x, y, z; // Тривимірні координати
public:
    kooClass() {x = y = z = 0; }
    kooClass(int a, int b, int c) {x = a; y = b; z = c; }
    // Повертає модифікований об'єкт, адресований покажчиком
    kooClass &operator+(int a) { x += a; y += a; z += a; return *this; }
    friend ostream &operator<<(ostream &stream, kooClass ObjA);
    friend bool operator<(kooClass ObjA, kooClass ObjB);
    friend bool operator==(kooClass ObjA, kooClass ObjB);
};

// Відображаємо координати x, y, z за допомогою оператора виведення для класу kooClass.
```

```
ostream &operator<<(ostream &stream, kooClass ObjA)
{
	stream << "x= " << ObjA.x << ", ";
	stream << "y= " << ObjA.y << ", ";
	stream << "z= " << ObjA.z << endl;
	return stream; // Повертає посилання на параметр stream
}

bool operator<(kooClass ObjA, kooClass ObjB)
{
	return (ObjA.x + ObjA.y + ObjA.z) < (ObjB.x + ObjB.y + ObjB.z);
}

bool operator==(kooClass ObjA, kooClass ObjB)
{
	return (ObjA.x + ObjA.y + ObjA.z) == (ObjB.x + ObjB.y + ObjB.z);
}

int main()
{
	vector<kooClass> vek; // Побудова вектора об'єктів нульової довжини

	// Додаємо у вектор об'єкти.
	for(int i=0; i<10; i++) vek.push_back(kooClass(i, i+2, i-3));

	cout << "Відображаємо вміст початкового вектора" << endl;
	for(int i=0; i<vek.size(); i++) cout << i << " ==> " << vek[i];
	cout << endl;

	// Модифікуємо об'єкти у векторі.
	for(int i=0; i<vek.size(); i++) vek[i] = vek[i] + 10;

	cout << "Відображаємо вміст модифікованого вектора" << endl;
	for(int i=0; i<vek.size(); i++) cout << i << " ==> " << vek[i];

	getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Відображаємо вміст початкового вектора.
0 ==> x= 0, y= 2, z= -3
1 ==> x= 1, y= 3, z= -2
2 ==> x= 2, y= 4, z= -1
3 ==> x= 3, y= 5, z= 0
4 ==> x= 4, y= 6, z= 1
5 ==> x= 5, y= 7, z= 2
6 ==> x= 6, y= 8, z= 3
7 ==> x= 7, y= 9, z= 4
8 ==> x= 8, y= 10, z= 5
9 ==> x= 9, y= 11, z= 6

Відображаємо вміст модифікованого вектора.
0 ==> x= 10, y= 12, z= 7
```

```
1 ==> x= 11, y= 13, z= 8
2 ==> x= 12, y= 14, z= 9
3 ==> x= 13, y= 15, z= 10
4 ==> x= 14, y= 16, z= 11
5 ==> x= 15, y= 17, z= 12
6 ==> x= 16, y= 18, z= 13
7 ==> x= 17, y= 19, z= 14
8 ==> x= 18, y= 20, z= 15
9 ==> x= 19, y= 21, z= 16
```

Вектори забезпечують надійне зберігання елементів, виявляючи при цьому надзвичайну потужність і гнучкість їх оброблення, але поступаються масивам у ефективності використання. Тому для більшості задач програмування частіше надається перевага звичайним масивам. Проте можливі ситуації, коли унікальні особливості векторів виявляються важливішими за витрати системних ресурсів, пов'язаних з їх використанням.

12.3.4. Доцільність використання ітераторів

Частково потужність бібліотеки STL зумовлена тим, що багато її функцій виконують ітератори. Такий механізм дає змогу виконувати операції з двома контейнерами одночасно. Розглянемо, наприклад, такий формат векторної функції **insert**():

```
template <class InIter> void insert(iterator i, InIter start, InIter end);
```

Ця функція вставляє початкову послідовність, яка визначається параметрами *start* і *end,* у приймальну послідовність, починаючи з позиції *i*. При цьому немає ніяких вимог, щоб ітератор *i* належав тому ж вектору, з яким пов'язані ітератори *start* і *end*. Таким чином, використовуючи цю версію функції **insert**(), можна елементи одного вектора вставити в інший. Розглянемо такий приклад.

Код програми 12.5. Демонстрація механізму використання ітераторів для вставлення елементів одного вектора у інший

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
#include <vector>            // Для роботи з контейнерним класом "Вектор"
using namespace std;         // Використання стандартного простору імен

int main()
{
	vector<char> vek, vek2; // Побудова векторів нульової довжини

	// Поміщаємо значення у вектор.
	for(int i=0; i<10; i++) vek.push_back('A' + i);

	// Відображаємо початковий вміст вектора.
	cout << "Початковий вміст вектора:" << endl;
	for(int i=0; i<vek.size(); i++) cout << vek[i] << " ";
	cout << endl << endl;
```

```
    // Ініціалізуємо другий вектор.
    char str[] = "-STL -- це сила!-";
    for(int i=0; str[i]; i++) vek2.push_back(str[i]);

    /* Отримуємо ітератори для середини вектора vek,
    а також початку і кінця вектора vek2. */
    vector<char>::iterator p = vek.begin()+5;
    vector<char>::iterator p2start = vek2.begin();
    vector<char>::iterator p2end = vek2.end();

    // Вставляємо вектор vek2 у вектор vek.
    vek.insert(p, p2start, p2end);

    // Відображаємо результат вставлення.
    cout << "Вміст вектора vek після вставлення: " << endl;
    for(int i=0; i<vek.size(); i++) cout << vek[i] << " ";
    cout << endl;

    getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Початковий вміст вектора:
A B C D E F G H I J

Вміст вектора vek після вставлення:
A B C D E – S T L - - ц е с и л а ! – F G H I J
```

Як бачите, вміст вектора **vek2** поміщено у середину вектора **vek**.

У міру подальшого вивчення можливостей, що надаються бібліотекою STL, можна дізнатися, що ітератори є зв'язними засобами, які роблять бібліотеку єдиним цілим. Вони дають змогу виконувати дії з двома (і більше) об'єктами бібліотеки STL одночасно, але особливо є корисними при використанні алгоритмів, описаних нижче у цьому розділі.

12.4. Механізми роботи зі списками

Одним з найпоширеніших контейнерів загального призначення є список. Аналогічний клас **list** підтримує функціонування двонаправленого лінійного списку. На відміну від вектора, у якому реалізовано підтримку довільного доступу, список дає змогу отримувати до своїх елементів тільки послідовний доступ. Двонаправленість списку означає, що доступ до його елементів можливий у двох напрямах: від початку до кінця і від кінця до початку.

Шаблонна специфікація класу **list** має такий вигляд.

```
template <class myType, class Allocator = allocator<myType> class list
```

У цьому записі myType – тип даних, який зберігаються у списку, а елемент **Allocator** означає розподільник пам'яті, який за замовчуванням використовує стандартний розподільник. У класі **list** визначено такі конструктори:

```
explicit list(const Allocator &a = Allocator());

explicit list(size_type num, const myType &val = myType (), const Allocator &a = Allocator());

list(const list<myType, Allocator> &ob);

template <class InIter>list(InIter start, InIter end, const Allocator &a = Allocator());
```

Конструктор, який представлено у першій формі, створює порожній список. Друга форма призначена для створення списку, який містить num елементів із значенням *val*. Третя форма створює список, який містить ті самі елементи, що і об'єкт *ob*. Четверта форма створює список, який містить елементи у діапазоні, заданому параметрами *start* і *end*.

Список *– це контейнер з двонаправленим послідовним доступом до його елементів.*

Для класу **list** визначено такі оператори порівняння: ==, <, <=, !=, > і >=.

12.4.1. Використання базових операцій для роботи зі списком

Функції-члени, визначені у класі **list**, подано у табл. 12.3. У кінець списку, як і у кінець вектора, елементи можна поміщати за допомогою функції **push_back**(), але за допомогою функції **push_front**() можна поміщати елементи у початок списку. Елемент можна також вставити і у середину списку, для цього використовується функція **insert**(). Один список можна помістити в інший, використовуючи функцію **splice**(). А за допомогою функції **merge**() два списки можна об'єднати і упорядкувати отриманий результат.

Табл. 12.3. Функції-члени класу list

Функція-член	Опис
1	2
template <class InIter> void assign(**InIter** *start*, **InIter** *end*);	Поміщає у список послідовність, що визначається параметрами *start* і *end*
void assign(**size_type** *num*, **const** myType &*val*);	Поміщає у список *num* елементів із значенням *val*
reference back(); **const_reference back**() **const**;	Повертає посилання на останній елемент у списку
iterator begin(); **const_iterator begin**() **const**;	Повертає ітератор для першого елемента у списку
void clear();	Видаляє всі елементи із списку
bool empty() **const**;	Повертає дійсне значення, якщо використовуваний список порожній, і помилкове – інакше
iterator end(); **const_iterator end**() **const**;	Повертає ітератор, який відповідає кінцю списку
iterator erase(**iterator** *i*);	Видаляє елемент, який адресується ітератором *i*; повертає ітератор, який вказує на елемент, розташований після видаленого
iterator erase(**iterator** *start*, **iterator** *end*);	Видаляє елементи у діапазоні, що задається параметрами *start* і *end*; повертає ітератор для елемента, розташованого за останнім видале-

1	2
	ним елементом
reference front(); **const_reference front**() **const**;	Повертає посилання на перший елемент у списку
allocator_type get_allocator() **const**;	Повертає розподільник пам'яті списку
iterator insert(**iterator** *i*, **const** myType &*val* = myType());	Вставляє значення *val* безпосередньо перед елементом, заданим параметром *i*; повертає ітератор для цього елемента
void insert(**iterator** *i*, **size_type** *num*, **const** myType &*val*);	Вставляє *num* копій значення *val* безпосередньо перед елементом, заданим параметром і
template <class InIter> void insert(**iterator** *i*, **InIter** *start*, **InIter** *end*);	Вставляє у список послідовність, що визначається параметрами *start* і *end*, безпосередньо перед елементом, заданим параметром *i*
size_type max_size() **const**;	Повертає максимальну кількість елементів, яке можу містити список
void merge(**list**<myType, **Allocator**> &*ob*>); **template <class Comp> void merge**(**list**<myType, **Allocator**> &*ob*, **Comp** *cmpfn*);	Об'єднує впорядкований список, що міститься в об'єкті *ob*, із впорядкованим списком, який його викликає. Результат також упорядковується. Після об'єднання список, що міститься в *ob*, залишається порожнім. У другому форматі може бути задана функція порівняння, яка визначає, за яких умов один елемент є меншим від іншого
void pop_back();	Видаляє останній елемент у списку
void pop_front();	Видаляє перший елемент у списку
void push_back(**const** myType &*val*);	Додає у кінець списку елемент із значенням, що задається параметром *val*
void push_front(**const** myType &*val*);	Додає у початок списку елемент із значенням, що задається параметром *val*
reverse_iterator rbegin(); **const_reverse_iterator rbegin**() **const**;	Повертає реверсивний ітератор для кінця списку
reverse_iterator rend(); **const_reverse_iterator rend**() **const**;	Повертає реверсивний ітератор для початку списку
void remove(**const** myType &*val*);	Видаляє із списку елементи із значенням *val*
template <class UnPred> void remove_if(**UnPred** *pr*);	Видаляє елементи, для яких унарний предикат *pr* дорівнює значенню true
void resize(**size_type** *num*, myType *val* = myType());	Встановлює розмір списку, що дорівнює значенню, заданому параметром *num*. Якщо список для цього потрібно подовжити, то у кінець списку додаються елементи із значенням, що задається параметром *val*
void reverse();	Реверсує список
size_type size() **const**;	Повертає поточну кількість елементів у списку
void sort(); **template <class Comp> void sort**(**Comp** *cmpfn*);	Сортує список. Друга форма сортує список за допомогою функції порівняння *cmpfn,* яка дає змогу визначати, за яких умов один елемент є меншим від іншого
void splice(**iterator** *i*, **list**<myType, **Allocator**> &*ob*);	Вставляє вміст списку *ob* у список, який його викликає, у позицію, вказану ітератором *i*. Після виконання цієї операції список *ob* зали-

1	2
	шається порожнім
void splice(**iterator** *i*, **list**<myType, **Allocator**> &*ob*>, **iterator** *el*);	Видаляє із списку *ob* елемент, який адресується ітератором *el*, і зберігає його у списку, який його викликає, у позицію, яка адресується ітератором *i*
void splice(**iterator** *i*, **list**<myType, **Allocator**> &*ob*, **iterator** *start*, **iterator** *end*);	Видаляє із списку *ob* діапазон, визначений параметрами *start* і end, і зберігає його у списку, який його викликає, починаючи з позиції, яка адресується ітератором *i*
void swap(**list**<myType, **Allocator**> &*ob*);	Здійснює обмін елементами списку, який його викликає, і списку *ob*
void unique(); **template <class BinPred> void unique**(**BinPred** *pr*);	Видаляє із списку елементи-дублікати. Друга форма для визначення унікальності використовує предикат *pr*

Щоб досягти максимальної гнучкості та переносності для будь-якого об'єкта, який підлягає зберіганню у списку, необхідно визначити конструктор за замовчуванням і оператор "<" (і бажано інші оператори порівняння). Точніші вимоги до об'єкта (як до потенційного елемента списку) необхідно погоджувати відповідно до документації на використовуваний Вами компілятор.

Розглянемо простий приклад використання списку.

Код програми 12.6. Демонстрація механізму використання базових операцій для роботи зі списком

```
#include <vcl>
#include <iostream>           // Для потокового введення-виведення
#include <conio>              // Для консольного режиму роботи
#include <list>               // Для роботи зі списками
using namespace std;          // Використання стандартного простору імен

int main()
{
    list<char> lst;   // Створення порожнього списку
    // Поміщаємо значення у список
    for(int i=0; i<10; i++) lst.push_back('A'+i);

    // Відображаємо початковий розміру списку
    cout << "Розмір = " << lst.size() << endl;

    // Відображаємо початковий вміст списку
    cout << "Вміст: ";
    list<char>::iterator p = lst.begin();
    // Отримуємо доступ до вмісту списку за допомогою ітератора.
    while(p != lst.end()) {
        cout << *p << " ";
        p++;
    }
    cout << endl;
    getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Розмір = 10
Вміст: A B C D E F G H I J
```

Внаслідок виконання ця програма створює список символів. Спочатку створюється порожній об'єкт списку. Потім у нього поміщається десять букв (від A до J). Заповнення списку реалізується шляхом використання функції **push_back**(), яка поміщає кожне нове значення у кінець наявного списку. Після цього відображається розмір списку і його вміст. Вміст списку виводиться на екран внаслідок виконання такого коду програми:

```
list<char>::iterator p = lst.begin();
while(p != lst.end()) {
    cout << *p << " ";
    p++;
}
```

Тут ітератор p ініціалізується так, що він вказує на початок списку. У процесі виконання чергового проходу циклу ітератор p інкрементується, внаслідок чого він вказує на наступний елемент списку. Цей цикл завершується тоді, коли ітератор p вказує на кінець списку. Застосування циклів з педедумовою – звичайна практика під час використання бібліотеки STL. Наприклад, аналогічний цикл ми застосували для відображення вмісту вектора у попередньому розділі.

Оскільки списки є двонаправленими, то заповнення їх елементами можна проводити з обох кінців. Наприклад, у процесі виконання наведеної нижче програми створюється два списки, причому елементи одного з них розташовані у порядку, зворотному стосовно іншого.

Код програми 12.7. Демонстрація механізму внесення елементів у список як з початку, так і з його кінця

```
#include <vcl>
#include <iostream>            // Для потокового введення-виведення
#include <conio>               // Для консольного режиму роботи
#include <list>                // Для роботи зі списками
using namespace std;           // Використання стандартного простору імен

int main()
{
    list<char> lst;     // Створення порожнього списку
    list<char> revlst; // Створення порожнього списку

    // Поміщаємо значення у список
    for(int i=0; i<10; i++) lst.push_back('A'+i);

    // Відображаємо початковий вміст списку
    cout << "Розмір списку lst = " << lst.size() << endl;
    cout << "Початковий вміст списку: ";
    list<char>::iterator p;

    /* Видаляємо елементи із списку lst і поміщаємо їх
    у список revlst у зворотному порядку. */
```

```
        while(!lst.empty()) {
                p = lst.begin();
                cout << *p << " ";
                revlst.push_front(*p);
                lst.pop_front();
        }
        cout << endl << endl;

        // Відображаємо реверсний вміст списку
        cout << "Розмір списку revlst = ";
        cout << revlst.size() << endl;
        cout << "Реверсний вміст списку: ";
        p = revlst.begin();
        while(p != revlst.end()) {
                cout << *p << " ";
                p++;
        }
        cout << endl;

        getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Розмір списку lst = 10
Початковий вміст списку: A B C D E F G H I J

Розмір списку revlst = 10
Реверсний вміст списку: J I H G F E D C B A
```

У цьому коді програми список реверсує шляхом видалення елементів з початку списку lst і занесення їх у початок списку revlst.

12.4.2. Особливості сортування списку

Список можна відсортувати за допомогою функції-члена класу **sort**(). У процесі виконання наведеної нижче програми створюється список випадково вибраних цілих чисел, який потім упорядковується за збільшенням їх значень.

Код програми 12.8. Демонстрація механізму сортування списку

```
#include <vcl>
#include <iostream>             // Для потокового введення-виведення
#include <conio>                // Для консольного режиму роботи
#include <list>                 // Для роботи зі списками
using namespace std;            // Використання стандартного простору імен

int main()
{
        list<int> lst;          // Створення порожнього списку

        // Створення списку випадково вибраних цілих чисел
        for(int i=0; i<10; i++) lst.push_back(rand());

        // Відображення початкового вмісту списку
```

```
        cout << "Початковий вміст списку:" << endl;
        list<int>::iterator p = lst.begin();
        while(p != lst.end()) {
                cout << *p << " ";
                p++;
        }
        cout << endl << endl;

        // Сортування списку.
        lst.sort();

        // Відображення відсортованого вмісту списку
        cout << "Відсортований вміст списку:" << endl;
        p = lst.begin();
        while(p != lst.end()) {
                cout << *p << " ";
                p++;
        }
        cout << endl;

        getch(); return 0;
}
```

Ось як може виглядати один з можливих варіантів виконання цієї програми:

```
Початковий вміст списку:
130 10982 1090 11656 7117 17595 6415 22948 31126 9004

Відсортований вміст списку:
130 1090 6415 7117 9004 10982 11656 17595 22948 31126
```

12.4.3. Об'єднання одного списку з іншим

Один впорядкований список можна об'єднати з іншим. Як результат ми отримаємо впорядкований список, який охоплює вміст двох початкових списків. Новий список залишається у списку, який його викликає, а другий список стає порожнім. У наведеному нижче прикладі здійснюється об'єднання двох списків. Перший список містить букви ACEGI, а другий – букви BDFHJ. Ці списки потім об'єднуються в один список, внаслідок чого утворюється впорядкована послідовність букв ABCDEFGHIJ.

Код програми 12.9. Демонстрація механізму об'єднання двох списків

```
#include <vcl>
#include <iostream>            // Для потокового введення-виведення
#include <conio>               // Для консольного режиму роботи
#include <list>                // Для роботи зі списками
using namespace std;           // Використання стандартного простору імен

int main()
{
        list<char> lst1, lst2; // Створення двох порожніх списків

        // Створення двох списків з випадково вибраними цілими числами
```

```
	for(int i=0; i<10; i+=2) lst1.push_back('A'+i);
	for(int i=1; i<11; i+=2) lst2.push_back('A'+i);

	// Відображення початкового вмісту першого списку
	cout << "Вміст списку lst1: ";
	list<char>::iterator p = lst1.begin();
	while(p != lst1.end()) {
		cout << *p;
		p++;
	}
	cout << endl << endl;

	// Відображення початкового вмісту другого списку
	cout << "Вміст списку lst2: ";
	p = lst2.begin();
	while(p != lst2.end()) {
		cout << *p;
		p++;
	}
	cout << endl << endl;

	// Тепер об'єднуємо ці два списки.
	lst1.merge(lst2);
	if(lst2.empty()) cout << "Список lst2 тепер порожній" << endl;

	// Відображення об'єднаного вмісту першого списку
	cout << "Вміст списку lst1 після об'єднання:" << endl;
	p = lst1.begin();
	while(p != lst1.end()) {
		cout << *p << " ";
		p++;
	}
	cout << endl;

	getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Вміст списку lst1: ACEGI
Вміст списку lst2: BDFHJ
Список lst2 тепер порожній.
Вміст списку lst1 після об'єднання:
A B C D E F G H I J
```

12.4.4. Зберігання у списку об'єктів класу

Розглянемо приклад, у якому список використовують для зберігання об'єктів типу myClass. Зверніть увагу на те, що для об'єктів типу myClass перевизначено оператори "<", ">", "!=" і "==". Для деяких компіляторів може виявитися зайвим визначення всіх цих операторів, або ж доведеться додати деякі інші з них. У бібліотеці STL ці функції використовуються для визначення впорядкування і рівності об'єктів у контейнері. Хоча список не є впорядкованим контейнером, необхідно мати

засіб порівняння елементів, який застосовується у процесі їх пошуку, сортування або об'єднання.

Код програми 12.10. Демонстрація механізму зберігання у списку об'єктів класу

```
#include <vcl>
#include <iostream>            // Для потокового введення-виведення
#include <conio>               // Для консольного режиму роботи
#include <list>                // Для роботи зі списками
using namespace std;           // Використання стандартного простору імен

class myClass { // Оголошення класового типу
	int a, b, sum;
public:
	myClass() { a = b = 0; }
	myClass(int c, int d) { a = c; b = d; sum = a + b; }
	int getSum() { return sum; }
	friend bool operator<(const myClass &ObjA, const myClass &ObjB);
	friend bool operator>(const myClass &ObjA, const myClass &ObjB);
	friend bool operator==(const myClass &ObjA, const myClass &ObjB);
	friend bool operator!=(const myClass &ObjA, const myClass &ObjB);
};

bool operator<(const myClass &ObjA, const myClass &ObjB)
	{ return ObjA.sum < ObjB.sum; }

bool operator>(const myClass &ObjA, const myClass &ObjB)
	{ return ObjA.sum > ObjB.sum; }

bool operator==(const myClass &ObjA, const myClass &ObjB)
	{ return ObjA.sum == ObjB.sum; }

bool operator!=(const myClass &ObjA, const myClass &ObjB)
	{ return ObjA.sum != ObjB.sum; }

int main()
{
	list<myClass> lst1;            // Створення першого списку.
	for(int i=0; i<10; i++) lst1.push_back(myClass(i, i));

	cout << "Перший список: ";
	list<myClass>::iterator p = lst1.begin();
	while(p != lst1.end()) {
		cout << p->getSum() << " ";
		p++;
	}
	cout << endl;

	list<myClass> lst2;            // Створення другого списку.
	for(int i=0; i<10; i++) lst2.push_back(myClass(i*2, i*3));

	cout << "Другий список: ";
```

```
	p = lst2.begin();
	while(p != lst2.end()) {
		cout << p->getSum() << " ";
		p++;
	}
	cout << endl;

	// Тепер об'єднуємо списки lst! i lst2.
	lst1.merge(lst2);

	// Відображаємо об'єднаний список.
	cout << "Об'єднаний список: "; p = lst1.begin();
	while(p != lst1.end()) {
		cout << p->getSum() << " ";
		p++;
	}
	cout << endl;
	getch(); return 0;
}
```

Ця програма створює два списки об'єктів типу myClass і відображає їх вміст. Потім здійснюється об'єднання цих двох списків з подальшим відображенням нового вмісту остаточного списку. Отже, Внаслідок виконання ця програма відображає на екрані такі результати:

```
Перший список: 0 2 4 6 8 10 12 14 16 18
Другий список: 0 5 10 15 20 25 30 35 40 45
Об'єднаний список: 0 0 2 4 5 6 8 10 10 12 14 15 16 18 20 25 30 35 40 45
```

12.5. Поняття про відображення – асоціативний контейнер

Клас **map** підтримує асоціативний контейнер, у якому унікальним ключам відповідають певні значення. По суті, ключ – це просте ім'я, яке присвоєне певному значенню. Після того, як значення збережено у контейнері, до нього можна отримати доступ за ключем. Таким чином, загалом відображення – це список пар "ключ-значення". Якщо нам відомий ключ, то ми можемо легко знайти значення. Наприклад, ми могли б визначити відображення, у якому як ключ використовують ім'я людини, а як значення – його телефонний номер. Асоціативні контейнери стають все більш популярними у програмуванні.

***Відо! раження** – це асоціативний контейнер.*

Як уже зазначалося вище, відображення може зберігати тільки унікальні ключі. Ключі-дублікати не дозволені. Щоб створити відображення, яке б давало змогу зберігати не унікальні ключі, використовується клас **multimap**.

Контейнер **map** має таку шаблонну специфікацію:

```
template<class Key, class myType, class Comp = less<Key>,
		class Allocator = allocator<pair<const Key myType>>> class map
```

Тут параметр Key – тип даних ключів, myType – тип значень, що зберігаються (що відображаються), а **Comp** – функція, яка порівнює два ключі. За замовчуванням як

функція порівняння використовується стандартна функція-об'єкт **less**. Елемент **Allocator** означає розподільник пам'яті, який за замовчуванням використовує стандартний розподільник **allocator**. Клас **map** має такі конструктори:

```
explicit map(const Comp &cmpfn = Comp(), const Allocator &a = Allocator());

map(const map<Key, myType, Comp Allocator> &ob>);

template<class InIter> map(InIter start, InIter end,
        const Comp &cmpfn = Comp(), const Allocator &a = Allocator());
```

Перша форма конструктора створює порожнє відображення. Друга призначена для створення відображення, яке містить ті ж елементи, що і відображення *ob*. Третя створює відображення, яке містить елементи у діапазоні, заданому ітераторами *start* і *end*. Функція, задана параметром *cmpfn* (якщо вона задана), визначає характер впорядкування відображення.

У загальному випадку будь-який об'єкт, який використовується як ключ, повинен визначати конструктор за замовчуванням і перевизначати оператор "<" (а також інші необхідні оператори порівняння). Ці вимоги для різних компіляторів є різними.

Для класу **map** визначено такі оператори порівняння: ==, <, <=, !=, > і >=.

Функції-члени, визначені для класу **map**, представлено в табл. 12.4. В їх описі під елементом **key_type** розуміють тип ключа, а під елементом **value_type** значення виразу **pair**<Key, myType>.

Табл. 12.4. Функції-члени, визначені у класі map

Функція-член класу	Призначення
1	2
iterator begin(); **const_iterator begin**() **const**;	Повертає ітератор для першого елемента у відображенні
void clear();	Видаляє всі елементи з відображення
size_type count(**const key_type** &*k*) **const**;	Повертає кількість входжень ключа *k* у відображенні (1 або 0)
bool empty() **const**;	Повертає значення **true**, якщо це відображення порожнє, і значення **false** – в іншому випадку
iterator end(); **const_iterator end**() **const**;	Повертає ітератор, який вказує на кінець відображення
pair<iterator, **iterator> equal_range**(**const key_type** &k); **pair<const_iterator**, **const_iterator> equal_range**(**const key_type** &k) **const**;	Повертає пару ітераторів, які вказують на перший і останній елементи у відображенні, що містять заданий ключ
void erase(**iterator** *i*);	Видаляє елемент, на який вказує ітератор *i*
void erase(**iterator** *start*, **iterator** *end*);	Видаляє елементи у діапазоні, що задаються параметрами *start* і *end*
size_type erase(**const key_type** &*k*);	Видаляє з відображення елементи, ключі яких мають значення *k*
iterator find(**const key_type** &*k*); **const_iterator find**(**const key_type** &k) **const**;	Повертає ітератор, який відповідає заданому ключу. Якщо такий ключ не виявлено, то повертає ітератор, який відповідає кінцю відображення
allocator_type get_allocator() **const**;	Повертає розподільник пам'яті відображення

1	2
iterator insert(**iterator** *i*, **const value_type** *&val*);	Вставляє значення *val* у позицію елемента (або після нього), що задається ітератором *i*. Повертає ітератор, який вказує на цей елемент
template<class InIter> void insert(**InIter** *start*, **InIter** *end*);	Вставляє елементи заданого діапазону
pair<iterator, **bool> insert**(**const value_type** *&val*);	Вставляє значення *val* у викликуване відображення. Повертає ітератор, який вказує на цей елемент. Елемент вставляється тільки у тому випадку, якщо його ще немає у відображенні. Якщо елемент було вставлено, то повертає значення **pair<iterator**, **true>**, інакше – значення **pair<iterator, false>**
key_compare key_Comp() **const**;	Повертає об'єкт-функцію, яка порівнює ключі
iterator lower_bound(**const key_type** *&k*); **const_iterator lower_bound**(**const key_type** *&k*) **const**;	Повертає ітератор для першого елемента у відображенні, ключ якого дорівнює значенню *k* або більше за це значення
size_type max_size() **const**;	Повертає максимальну кількість елементів, яку може містити це відображення
reference operator[](**const key_type** *&i*);	Повертає посилання на елемент, які задаються параметром *i*. Якщо цього елемента не існує, то вставляє його у відображення
reverse_iterator rbegin(); **const_reverse_iterator rbegin**() **const**;	Повертає реверсивний ітератор, який відповідає кінцю відображення
reverse_iterator rend(); **const_reverse_iterator rend**() **const**;	Повертає реверсивний ітератор, який відповідає початку відображення
size_type size() **const**;	Повертає поточну кількість елементів у відображенні
void swap(**map**<Key, myType, **Comp Allocator**> *&ob*);	Здійснює обмін елементами викликуваного відображення та відображення *ob*
iterator upper_bound(**const key_type** *&k*); **const_iterator upper_bound**(**const key_type** *&k*) **const**;	Повертає ітератор для першого елемента у відображенні, ключ якого є більшим від заданого значення k
value_compare value_Comp() **const**;	Повертає об'єкт-функцію, яка порівнює значення

Пари "ключ-значення" зберігаються у відображенні як об'єкти класу **pair**, який має таку шаблонну специфікацію:

```
template <class kType, class vType> struct pair {
    typedef kType first_type;        // Тип ключа
    typedef vType second_type;       // Тип значення
    kType first;                     // Містить ключ
    vType second;                    // Містить значення

    // Оголошення конструкторів
    pair();
    pair(const kType &k, const vType &v);
    template<class A, class B> pair(const<A, B> &ob);
}
```

Як зазначено у коментарях до цього фрагменту програми, член **first** містить ключ, а член **second** – значення, що відповідає цьому ключу.

12.5.1. Робота з відображеннями

Створити пару "ключ-значення" можна або за допомогою конструкторів класу **pair**, або шляхом виклику функції **make_pair**(), яка створює парний об'єкт на основі типів даних, які використовуються як параметри. Функція **make_pair**() – це узагальнена функція, прототип якої має такий вигляд:

```
template <class kType, class vType>
        pair<kType, vType> make_pair(const kType &k, const vType &v);
```

Як бачите, функція **make_pair**() повертає парний об'єкт, який складається із значень, типи яких задано параметрами *kType і vType*. Перевага використання функції **make_pair**() полягає у тому, що типи об'єктів, які об'єднуються у пару, визначаються автоматично компілятором, а не безпосередньо задаються програмістом.

Можливості використання відображення продемонстровано у наведеному нижче коді програми. У цьому випадку у відображенні зберігається 10 пар "ключ-значення". Ключем слугує символ, а значенням – ціле число. Пари "ключ-значення" зберігаються так:

A 0
B 1
C 2

і т.д. Після збереження пар у відображенні користувачу пропонується ввести ключ (тобто букву з діапазону A-J), після чого виводиться значення, пов'язане з цим ключем.

Код програми 12.11. Демонстрація механізму використання простого відображення

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
#include <map>               // Для роботи з асоціативними контейнерами
using namespace std;         // Використання стандартного простору імен

int main()
{
    map<char, int> h;        // Створення порожнього відображення.
    // Поміщаємо пари у відображення.
    for(int i=0; i<10; i++) {
        h.insert(pair<char, int>('A'+i, i));
    }
    char ch;
    cout << "Введіть ключ: "; cin >> ch;

    map<char, int>::iterator p;

    // Знаходимо значення за заданим ключем.
    p = h.find(ch);
    if(p != h.end()) cout << p->second;
    else cout << "Такого ключа у відображенні немає" << endl;
    getch(); return 0;
}
```

Зверніть увагу на використання шаблонного класу **pair** для побудови пар "ключ-значення". Типи даних, які задаються **pair**-виразом, повинні відповідати типам відображення, в які вставляються ці пари.

Після ініціалізації відображення ключами і значеннями можна виконувати пошук значення за заданим ключем, використовуючи функцію **find**(). Ця функція повертає ітератор, який вказує на потрібний елемент або на кінець відображення, якщо заданого ключа не було знайдено. Внаслідок виявлення збігу значення, пов'язаного з ключем, можна знайти в члені **second** парного об'єкта типу **pair**.

У наведеному вище прикладі пари "ключ-значення" створювалися безпосередньо за допомогою шаблону **pair<char**, **int>**. І хоча у цьому немає нічого неправильного, часто простіше використовувати з цією метою функцію **make_pair**(), яка створює **pair**-об'єкт на основі типів даних, які використовуються як параметри. Наприклад, цей рядок коду програми також дасть змогу вставити у відображення h пари "ключ-значення" (при використанні попереднього коду програми):

```
h.insert(make_pair((char)('A'+c), c));
```

У цьому записі, як бачите, здійснюється операція приведення до типу **char**, яка необхідна для перевизначення автоматичного перетворення в тип **int** результату додавання значення i з символом 'A'.

12.5.2. Зберігання у відображенні об'єктів класу

Подібно до всіх інших контейнерів, відображення можна використовувати для зберігання об'єктів, створюваних Вами типів. Наприклад, наведений нижче код програми створює простий словник на основі відображення слів з їх значеннями. Але спочатку вона створює два класи word і meaning. Оскільки відображення підтримує відсортований список ключів, програма також визначає для об'єктів типу word оператора "<". У загальному випадку оператор "<" необхідно визначати для будь-яких класів, які Ви плануєте використовувати як ключі. Деякі компілятори можуть зажадати визначення і інших операторів порівняння.

Код програми 12.12. Демонстрація механізму використання відображення для створення словника

```
#include <vcl>
#include <iostream>              // Для потокового введення-виведення
#include <conio>                 // Для консольного режиму роботи
#include <map>                   // Для роботи з асоціативними контейнерами
#include <cstring>               // Для роботи з рядковими типами даних
using namespace std;             // Використання стандартного простору імен

class word {
	char str[20];
public:
	word() { strcpy(str, ""); }
	word(char *s) { strcpy(str, s); }
	char *get() { return str; }
};

bool operator<(word ObjA, word ObjB)
```

```
{
	return strcmp(ObjA.get(), ObjB.get()) < 0;
}

class meaning {
	char str[80];
public:
	meaning() { strcmp(str, ""); }
	meaning(char *s) { strcpy(str, s); }
	char *get() { return str; }
};

int main()
{
	map<word, meaning> dictionary;

	// Поміщаємо у відображення об'єкти класів word і meaning.
	dictionary.insert(pair<word, meaning>(
		word("дім"), meaning("Місце мешкання.")));

	dictionary.insert(pair<word, meaning>(
		word("клавіатура"), meaning("Пристрій введення даних.")));

	dictionary.insert(pair<word, meaning>(
		word("програмування"), meaning("Процес розроблення програми.")));

	dictionary.insert(pair<word, meaning>(
		word("STL"), meaning("Standard Template Library")));

	// За заданим словом знаходимо його значення.
	char str[80];
	cout << "Введіть слово: "; cin >> str;

	map<word, meaning>::iterator p;
	p = dictionary.find(word(str));
	if(p != dictionary.end())
		cout << "Визначення: " << p->second.get();
	else
		cout << "Такого слова у словнику немає" << endl;

	getch(); return 0;
}
```

Один з можливих варіантів виконання цієї програми

```
Введіть слово: дім
Визначення: Місце мешкання.
```

У цьому коді програми кожен елемент відображення є символьним масивом, який містить рядок, який завершується нульовим символом, Нижче у цьому розділі розглядається простіший варіант побудови цієї програми, у якій використано стандартний тип **string**.

12.6. Алгоритми оброблення контейнерних даних

Алгоритми обробляють дані, які містяться у контейнерах. Незважаючи на те, що кожен контейнер забезпечує підтримку власних базових операцій, однак стандартні алгоритми дають змогу виконувати більш розширені або складніші дії. Вони також дають змогу виконувати дії з двома різними типами контейнерів одночасно. Для отримання доступу до алгоритмів бібліотеки STL необхідно приєднати до програми заголовок **<algorithm>**

У бібліотеці STL визначено багато алгоритмів, які описано в табл. 12.5. Всі ці алгоритми є шаблонними функціями. Це означає, що їх можна застосовувати до контейнера будь-якого типу.

Табл. 12.5. Алгоритми STL

Алгоритм	Призначення
1	2
adjacent_find	Здійснює пошук суміжних елементів, які збігаються усередині послідовності, і повертає ітератор для першого знайденого збігу
binary_search	Здійснює двійковий пошук заданого значення усередині впорядкованої послідовності
copy	Копіює послідовність
copy_backward	Аналогічний алгоритму **copy**, за винятком того, що копіювання відбувається у зворотному порядку, тобто спочатку переміщаються елементи, які знаходяться у кінці послідовності
count	Повертає кількість елементів із заданим значенням у послідовності
count_if	Повертає кількість елементів, які задовольняють заданий предикат
equal	Визначає, чи однакові два діапазони
equal_range	Повертає діапазон, у який можна вставити елемент, не порушуючи порядок певної послідовності
fill, fill_n	Заповнюють діапазон заданим значенням
find	У заданому діапазоні здійснює пошук заданого значення і повертає ітератор для першого входження знайденого елемента
find_end	У заданому діапазоні здійснює пошук заданої послідовності. Повертає ітератор, який відповідає кінцю шуканої послідовності
find_first_of	Здійснює пошук першого елемента усередині заданої послідовності, який збігається з будь-яким елементом із заданого діапазону
find_if	У заданому діапазоні здійснює пошук елемента, для якого визначений користувачем унарний предикат повертає значення true
for_each	Застосовує задану функцію до заданого діапазону елементів
generate, generate_n	Присвоюють значення, яке повертаються деякою функцією-генератором, елементам із заданого діапазону
includes	Встановлює факт внесення всіх елементів однієї заданої послідовності в іншу задану послідовність

1	2
inplace_merge	Об'єднує один заданий діапазон з іншим. Обидва діапазони мають бути відсортовані у порядку зростання. Після виконання алгоритму отримана послідовність сортується у порядку зростання
iter_swap	Змінює місцями значення, яке адресуються ітераторами, які передаються як параметри
lexicographical_compare	Порівнює одну задану послідовність з іншою в лексикографічному порядку
lower_bound	Здійснює пошук першого елемента у заданій послідовності, значення якого не менше від заданого значення
make_heap	Створює сукупність із заданої послідовності
max	Повертає максимальне з двох значень
max_element	Повертає ітератор для максимального елемента усередині заданого діапазону
merge	Об'єднує дві впорядковані послідовності, поміщаючи результат у третю послідовність
min	Повертає мінімальне з двох значень
min_element	Повертає ітератор для мінімального елемента усередині заданого діапазону
mismatch	Здійснює пошук першого не збігання елементів у двох послідовностях і повертає ітератори для цих двох елементів
next_permutation	Створює наступну перестановку заданої послідовності
nth_element	Упорядковує задану послідовність так, щоби всі елементи, значення яких менші від значення *E*, розміщувалися перед цим елементом, а всі елементи, значення яких є більшим від значення *E*, розміщувалися після нього
partial_sort	Сортує заданий діапазон
partial_sort_copy	Сортує заданий діапазон, а потім копіює стільки елементів, скільки може поміститися в остаточну послідовність
partition	Сортує задану послідовність так, щоб усі елементи, для яких заданий предикат повертає значення **true**, розміщувалися перед елементами, для яких цей предикат повертає значення **false**
pop_heap	Змінює місцями перший і передостанній елементи заданого діапазону, а потім перебудовує сукупність
prev_permutation	Створює попередню перестановку послідовності
push_heap	Поміщає елемент у кінець сукупності
random_shuffie	Додає випадковий характер заданій послідовності
remove, remove_if, remove_copy, remove_copy_if	Видаляють елементи із заданого діапазону
replace, replace_copy, replace_if, replace_copy_if	Замінюють задані елементи з діапазону іншими елементами
reverse, reverse_copy	Змінює порядок слідування елементів у заданому діапазоні на протилежний
rotate, rotate_copy	Здійснює циклічний зсув вліво елементів у заданому діапазоні
search	Здійснює пошук однієї послідовності усередині іншої
search_n	Усередині певної послідовності здійснює пошук заданої кількості подібних елементів
set_difference	Створює послідовність, яка містить різницю двох впорядкованих множин

1	2
set_intersection	Створює послідовність, яка містить перетин двох впорядкованих множин
set_symmetric_difference	Створює послідовність, яка містить симетричну різницю двох впорядкованих множин
set_union	Створює послідовність, яка містить об'єднання двох впорядкованих множин
sort	Сортує заданий діапазон
sort_heap	Сортує сукупність у заданому діапазоні
stable_partition	Упорядковує задану послідовність так, щоб усі елементи, для яких заданий предикат повертає значення **true**, розміщувалися перед елементами, для яких цей предикат повертає значення **false**. Таке розбиття є стабільним, що означає збереження відносного порядку послідовності
stable_sort	Здійснює стійке (стабільне) сортування заданого діапазону. Це означає, що однакові елементи не переставляються
swap	Змінює місцями задані два значення
swap_ranges	Здійснює обмін елементів у заданому діапазоні
transform	Застосовує функцію до заданого діапазону елементів і зберігає результат у новій послідовності
unique, unique_copy	Видаляє елементи, які повторюються, із заданого діапазону
upper_bound	Знаходить останній елемент у заданій послідовності, який не більший від заданого значення

12.6.1. Обчислення кількості елементів

Одна з найпопулярніших операцій, яку можна виконати для будь-якої послідовності елементів, – це обчислення їх кількості. Для цього можна використовувати один з алгоритмів: **count**() або **count_if**(). Загальний формат цих алгоритмів має такий вигляд:

```
template <class InIter, class myType>
    ptrdiff_t count(InIter start, InIter end, const myType &val);

template <class InIter, class UnPred>
    ptrdiff_t count_if(InIter start, InIter end, UnPred pfn);
```

Алгоритм **count**() повертає кількість елементів послідовності, які дорівнюють значенню *val*, межі якої задані параметрами *start і end.* Алгоритм **count_if**(), діючи у послідовності, межі якої задані параметрами *start* і *end*, повертає кількість елементів, для яких унарний предикат *pfn* повертає значення **true**. Тип **ptrdiff_t** визначається як певний різновид цілочисельного типу.

Механізм використання алгоритмів **count**() і **count_if**() продемонстровано у наведеному нижче коді програми.

Код програми 12.13. Демонстрація механізму використання алгоритмів count і count_if

```
#include <vcl>
#include <iostream>            // Для потокового введення-виведення
#include <conio>               // Для консольного режиму роботи
#include <vector>              // Для роботи з контейнерним класом "Вектор"
```

```
#include <algorithm>          // Для роботи з алгоритмами бібліотеки STL
#include <cctype>             // Для роботи з символьними аргументами
using namespace std;          // Використання стандартного простору імен

// Це унарний предикат, який визначає, чи представляє цей символ голосний звук.
// а б в г д е є ж з и і ї й к л м н о п р с т у ф х ц ч ш щ ю я ь
bool isvowel(char ch) {
	ch = tolower(ch);
	if(ch=='а' || ch=='е' || ch=='є' || ch=='и'
		|| ch=='і' || ch=='ї' || ch=='о' || ch=='у'
		|| ch=='ю' || ch=='я') return true;
	return false;
}

int main()
{
	char str[] = "STL-програмування -- це сила!";
	vector<char> vek;
	unsigned int i;

	for(int i=0; str[i]; i++) vek.push_back(str[i]);
	cout << "Послідовність: ";

	for(int i=0; i<vek.size(); i++) cout << vek[i];
	cout << endl;

	int n; char ch = 'н';
	n = count(vek.begin(), vek.end(), ch);
	cout << n << " символи '" << ch << "'" << endl;

	n = count_if(vek. begin(), vek.end(), isvowel);
	cout << n << " символів представляють голосні звуки" << endl;

	getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Послідовність: STL-програмування -- це сила!
2 символи н
8 символів представляють голосні звуки.
```

Програма починається із створення вектора, який містить рядок "STL-програмування – це сила!". Потім використовується алгоритм **count**() для підрахунку кількості букв 'н' у цьому векторі. Після цього викликається алгоритм **count_if**(), який підраховує кількість символів, що представляють голосні звуки з використанням як предикату функції **isvowel**(). Зверніть увагу на те, як закодований цей предикат. Всі унарні предикати отримують як параметр об'єкт, тип якого збігається з типом елементів, що зберігаються у контейнері, для якого і створюється цей предикат. Предикат повинен повертати значення ІСТИНА або ФАЛЬШ.

12.6.2. Видалення та заміна елементів

Іноді корисно згенерувати нову послідовність, яка складатиметься тільки з певних елементів початкової послідовності. Одним з алгоритмів, який може справитися з цим завданням, є **remove_copy**(). Його загальний формат має такий вигляд:

```
template <class ForIter, class OutIter, class myType>
    OutIter remove_copy(InIter start, InIter end,
                        OutIter result, const myType &val);
```

Алгоритм **remove_copy**() копіює з вилученням із заданого діапазону елементи, які дорівнюють значенню *val*, і поміщає результат у послідовність, яка адресується параметром *result*. Алгоритм повертає ітератор, який вказує на кінець результату. Контейнер-приймач повинен бути достатньо великим, щоби прийняти отриманий результат.

Щоб у процесі копіювання у послідовності один елемент замінити іншим, використовується алгоритм **replace_copy**(). Його загальний формат має такий вигляд:

```
template <class ForIter, class OutIter, class myType>
    OutIter replace_copy(InIter start, InIter end,
        OutIter result, const myType &old, Const myType &new);
```

Алгоритм **replace_copy**() копіює елементи із заданого діапазону у послідовність, яка адресується параметром *result*. У процесі копіювання відбувається заміна елементів, які мають значення *old,* елементами за значенням *new.* Алгоритм поміщає результат у послідовність, яка адресується параметром *result,* і повертає ітератор, який вказує на кінець цієї послідовності. Контейнер-приймач повинен бути достатньо великим, щоби прийняти отриманий результат.

У наведеному нижче коді програми продемонстровано механізм використання алгоритмів **remove_copy**() і **replace_copy**(). Під час її виконання створюється послідовність символів, з якої видаляються всі букви 'e'. Потім здійснюється заміна всіх букв 'e' буквами 'x'.

Код програми 12.14. Демонстрація механізму використання алгоритмів remove_copy і replace_copy

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
#include <vector>            // Для роботи з контейнерним класом "Вектор"
#include <algorithm>         // Для роботи з алгоритмами бібліотеки STL
using namespace std;         // Використання стандартного простору імен

int main()
{
    char str[] = "Це дуже простий тест.";
    vector<char> vek, vek2(30);

    for(int i=0; str[i]; i++) vek.push_back(str[i]);

    // **** Демонстрація алгоритму remove_copy ****

    cout << "Вхідна послідовність: ";
```

```
        for(int i=0; i<vek.size(); i++) cout << vek[i];
        cout << endl;

        // Видаляємо всі букви 'е'.
        remove_copy(vek.begin(), vek.end(), vek2.begin(), 'т');

        cout << "Після видалення букв 'т': ";
        for(int i=0; vek2[i]; i++) cout << vek2[i];
        cout << endl << endl;

        // **** Демонстрація алгоритму replace_copy ****

        cout << "Вхідна послідовність: ";
        for(int i=0; i<vek.size(); i++) cout << vek[i];
        cout << endl;

        // Замінюємо букви 'е' буквами 'Х'.
        replace_copy(vek.begin(), vek.end(), vek2.begin(), 'o', 'x');

        cout << "Після заміни букв 'е' буквами 'Х': ";
        for(int i=0; vek2[i]; i++) cout << vek2[i];
        cout << endl << endl;
        getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Вхідна послідовність: Це дуже простий тест.
Після видалення букв 'е': Ц дуж простий тст.

Вхідна послідовність: Це дуже простий тест.
Після заміни букв 'е' буквами 'Х': ЦХ дужХ простий тХст.
```

12.6.3. Реверсування послідовності

У програмах часто використовують алгоритм **reverse**(), який у діапазоні, заданому параметрами *start* і *end*, змінює порядок слідування елементів на протилежний. Його загальний формат має такий вигляд:

```
template <class Bilter> void reverse(Bilter start, Bilter end);
```

У наведеному нижче коді програми продемонстровано механізм використання цього алгоритму.

Код програми 12.15. Демонстрація механізму використання алгоритму reverse

```
#include <vcl>
#include <iostream>           // Для потокового введення-виведення
#include <conio>              // Для консольного режиму роботи
#include <vector>             // Для роботи з контейнерним класом "Вектор"
#include <algorithm>          // Для роботи з алгоритмами бібліотеки STL
using namespace std;          // Використання стандартного простору імен

int main()
{
        vector<int> vek;
        unsigned int i;
```

```
        for(int i=0; i<10; i++) vek.push_back(i);

        cout << "Початкова послідовність: ";
        for(int i=0; i<vek.size(); i++) cout << vek[i] << " ";
        cout << endl;

        reverse(vek. begin(), vek.end());

        cout << "Реверсована послідовність: ";
        for(int i=0; i<vek.size(); i++) cout << vek[i] << " ";
        getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Початкова послідовність: 0 1 2 3 4 5 6 7 8 9
Реверсована послідовність: 9 8 7 6 5 4 3 2 1 0
```

12.6.4. Перетворення послідовності

Одним з найцікавіших є алгоритм перетворення послідовності **transform**(), який дає змогу модифікувати кожен елемент у діапазоні відповідно до заданої функції. Алгоритм **transform**() використовується у двох загальних форматах:

```
template <class InIter, class OutIter, class Func>
        OutIter transform(InIter start, InIter end,
                OutIter result, Func unaryfunc);

template <class InIter1, class InIter2, class OutIter, class Func>
        OutIter transform(InIter1 start1, InIter1 end1,
                InIter2 start2, OutIter result, Func binaryfunc);
```

Алгоритм **transform**() застосовує функцію до діапазону елементів і зберігає результат у послідовності, яка задається параметром *result.* У першій формі діапазон задається параметрами *start* і *end.* Використовувана для перетворення функція задається параметром *unaryfunc.* Вона приймає значення елемента як параметр повертає перетворене значення. У другому форматі алгоритму перетворення здійснюється з використанням бінарної функції, яка приймає як перший параметр значення, призначеного для перетворення елемента з послідовності, а як другий параметр – елемент з другої послідовності. Обидві версії повертають ітератор, який вказує на кінець остаточної послідовності.

У наведеному нижче коді програми використовується проста функція перетворення **xform**(), яка підносить до квадрату кожен елемент списку. Зверніть увагу на те, що остаточна послідовність зберігається у тому ж списку, який містив початкову послідовність.

Код програми 12.16. Демонстрація механізму використання алгоритму transform()

```
#include <vcl>
#include <iostream>             // Для потокового введення-виведення
#include <conio>                // Для консольного режиму роботи
#include <list>                 // Для роботи зі списками
```

```
#include <algorithm>          // Для роботи з алгоритмами бібліотеки STL
using namespace std;          // Використання стандартного простору імен

// Проста функція перетворення.
int xform(int i) {
        return i * i; // Квадрат початкового значення
}

int main()
{
        list<int> xList;
        int i;

        // Поміщаємо значення у список.
        for(int i=0; i<10; i++) xList.push_back(i);

        cout << "Початковий список xList: ";
        list<int>::iterator p = xList.begin();
        while(p != xList.end()) {
                cout << *p << " ";
                p++;
        }
        cout << endl;

        // Перетворення списку xList.
        p = transform(xList.begin(), xList.end(), xList.begin(), xform);

        cout << "Перетворений список xList: ";
        p = xList.begin();
        while(p != xList.end()) {
                cout << *p << " ";
                p++;
        }

        getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Початковий список xList: 0 1 2 3 4 5 6 7 8 9
Перетворений список xList: 0 1 4 9 16 25 36 49 64 81
```

Як бачите, кожен елемент у списку xList тепер зведено у квадрат.

Описані вище алгоритми є тільки малою частиною всього вмісту бібліотеки STL. Зазвичай Вам потрібно самим досліджувати інші алгоритми. Заслуговують уваги, наприклад, такі алгоритми, як **set_union**() і **set_difference**(). Вони призначені для оброблення вмісту такого контейнера, як множина. Цікаві також алгоритми **next_permutation**() і **prev_permutation**(). Вони створюють наступну і попередню перестановки елементів заданої послідовності. Час, витрачений на вивчення алгоритмів бібліотеки STL, – це час, витрачений не дарма!

12.7. Особливості використання об'єктів класу string

Як уже зазначалося вище, мова програмування С++ не підтримує вбудованого рядкованого типу. Проте він надає два способи оброблення рядків. По-перше, для представлення рядків можна використовувати традиційний символьний масив, який завершується нулем. Рядки, що створюються у такий спосіб (Ви вже ознайомлені з ним), іноді називають *С-рядками.* По-друге, можна використовувати об'єкти класу **string**, і саме цей спосіб ми зараз розглянемо.

---

*Клас **string** забезпечує альтернативу для рядків, які мають завершальний нуль-символ.*

Насправді клас **string** є не чим іншим, як спеціалізація більш загального шаблонного класу **basic_string**. Існує дві спеціалізації типу **basic_string**: тип **string**, який підтримує 8-бітові символьні рядки, і тип **wstring**, який підтримує рядки, утворені двобайтовими символами. Найчастіше у звичайному програмуванні використовують рядкові об'єкти типу **string**. Для використання рядкових класів С++ необхідно приєднати до програми заголовок **<string>**.

12.7.1. Клас string – частина С++-бібліотеки

Перед тим, як розглядати особливості роботи з класом **string**, важливо зрозуміти, чому він є частиною С++-бібліотеки. Стандартні класи не відразу були додані у визначення мови програмування С++. Насправді кожному нововведенню передували серйозні дискусії та запеклі суперечки. При тому, що мова С++ вже містить підтримку рядків у вигляді масивів, що завершуються нулем, внесення класу **string** у мову програмування С++, на перший погляд, може видатися винятком з цього правила. Але це далеко не так. І ось чому: рядки з завершальним нуль-символом, не можна обробляти стандартними С++-операторами, і їх не можна використовувати у звичайних С++-виразах. Розглянемо, наприклад, такий фрагмент коду програми:

```
char s1[80], s2[80], s3[80];
s1 = "один";      // так робити не можна
s2 = "два";       // так робити не можна
s3 = s1 + s2;     // Помилка
```

Як зазначено у коментарях до цього коду програми, у мові програмування С++ неможливо використовувати оператор присвоєння для надання символьному масиву нового значення (за винятком настанови ініціалізації), а також не можна застосовувати операцію додавання "+" для конкатенації двох рядків. Ці операції можна виконати за допомогою бібліотечних функцій.

```
strcpy(s1, "один");
strcpy(s2, "два");
strcpy(s3, s1);
strcpy(s3, s2);
```

Оскільки символьний масив, який завершується нулем, формально не є самостійним типом даних, то до нього не можна застосувати С++-оператори. Це позбавляє "вишуканості" навіть найелементарніші операції з рядками. І саме нездат-

ність обробляти рядки з завершальним нуль-символом, за допомогою стандартних С++-операторів привела до розроблення стандартного рядкового класу. Пригадайте: створюючи клас у мові програмування С++, ми визначаємо новий тип даних, який можна повністю інтегрувати у С++-середовище. Це, звичайно ж, означає, що для нового класу можна перевизначати оператори. Таким чином, вводячи в мову стандартний рядковий клас, ми створюємо можливість для оброблення рядків так само, як і даних будь-якого іншого типу, а саме за допомогою операторів.

Проте існує ще одна причина, яка реабілітує створення стандартного класу **string**: безпека. Руками недосвідченого або необережного програміста дуже легко забезпечити вихід за межі масиву, який містить рядок з завершальним нулем. Розглянемо, наприклад, стандартну функцію копіювання **strcpy**(). Ця функція не передбачає механізм перевірки факту порушення меж масиву приймача. Якщо початковий масив містить більше символів, ніж може прийняти масив приймач, то внаслідок цієї помилки дуже вірогідна повна відмова системи. Як буде показано далі, стандартний клас **string** не допускає виникнення подібних помилок

Отже, існує три причини для внесення у мову програмування С++ стандартного класу **string**: несуперечність даних (рядок тепер визначається самостійним типом даних), зручність (програміст може використовувати стандартні С++-оператори) і безпечність (межі масивів відтепер не порушуватимуться). Необхідно мати на увазі, що все перераховане вище зовсім не означає, що програміст повинен відмовлятися від використання звичайних рядків, які мають завершальний нуль-символ. Вони, як і раніше, залишаються найбільш ефективним засобом реалізації рядків. Але, якщо швидкість виконання програми не є для Вас визначальним фактором, то використання нового класу **string** дасть Вам доступ до безпечного і повністю інтегрованого способу оброблення рядків.

І хоча клас **string** традиційно не сприймається як частина бібліотеки STL, проте він є ще одним контейнерним класом, який визначено у мові програмування С++. Це означає, що він підтримує алгоритми, описані у попередньому розділі. До того ж, рядки мають додаткові можливості. Для отримання доступу до класу **string** необхідно приєднати до програми заголовок **<string>**.

Клас **string** дуже великий, він містить багато конструкторів і функцій-членів. Окрім цього, багато функцій-членів класу мають декілька перевизначених форм. Оскільки в одному розділі неможливо розглянути весь вміст класу **string**, ми зверніть увагу тільки на найпопулярніших його засобах. Отримавши загальне уявлення про роботу класу **string**, Ви зможете легко розібратися в інших його можливостях самі.

Прототипи трьох найпоширеніших конструкторів класу **string** мають такий вигляд:

```
string();
string(const char *str);
string(const string &str);
```

Перша форма конструктора створює порожній об'єкт класу **string**. Друга форма створює **string**-об'єкт з рядка, який завершується нульовим символом, який адресується параметром *str*. Ця форма конструктора забезпечує перетворення рядка з

завершальним нуль-символом на об'єкт типу **string**. Третя форма створює **string**-об'єкт з іншого **string**-об'єкта.

Для об'єктів класу **string** визначено такі оператори:

Оператор	Опис
=	Присвоєння
+	Конкатенація
+=	Присвоєння з конкатенацією
==	Рівність
!=	Нерівність
<	Менше
<=	Менше або дорівнює
>	Більше
>=	Більше або дорівнює
[]	Індексація
<<	Введення
>>	Виведення

Ці оператори дають змогу використовувати об'єкти типу **string** у звичайних виразах і позбавляють програміста потреби викликати такі функції, як **strcpy**() або **strcpy**(). У загальному випадку у виразах можна змішувати **string**-об'єкти і рядки з завершальним нуль-символом. Наприклад, **string**-об'єкт можна присвоїти рядку, що завершується нулем.

Оператор додавання "+" можна використовувати для конкатенації одного **string**-об'єкта з іншим або **string**-об'єкта з рядком, створеним у С-стилі (С-рядком). Іншими словами, підтримуються такі операції:

string-об'єкт + **string**-об'єкт
string-об'єкт + С-рядок
С-рядок + **string**-об'єкт

Оператор додавання "+" дає змогу також додавати символ у кінець рядка.

У класі **string** визначено константу **npos**, яка дорівнює -1. Вона представляє розмір рядка максимально можливої довжини.

Рядковий клас у мові програмування C++ істотно полегшує оброблення рядків. Наприклад, використовуючи **string**-об'єкти, можна застосовувати оператор присвоєння для призначення **string**-об'єкту рядка в лапках, оператор конкатенації "+" – для конкатенації рядків і оператори порівняння – для порівняння рядків. Виконання цих операцій продемонстровано у наведеному нижче коді програми.

Код програми 12.17. Демонстрація механізму використання класу string для оброблення рядків

```
#include <vcl>
#include <iostream>          // Для потокового введення-виведення
#include <conio>             // Для консольного режиму роботи
#include <string>            // Для роботи з рядковими типами
using namespace std;         // Використання стандартного простору імен

int main()
{
    string str1("Клас string дає змогу ефективно ");
    string str2("обробляти рядки.");
```

```
        string str3;

        // Присвоєння string-об'єкта
        str3 = str1;
        cout << str1 << endl << str3 << endl;

        // Конкатенація двох string-об'єктів.
        str3 = str1 + str2;
        cout << str3 << endl;

        // Порівняння string-об'єктів.
        if(str3 > str1) cout << "str3 > str1" << endl;
        if(str3 == str1 + str2)
        cout << "str3 == str1 + str2" << endl;

        // Об'єкту класу string можна також присвоїти звичайний рядок.
        str1 = "Це рядок, який завершується нульовим символом" << endl;
        cout << str1;

        // Створення string-об'єкта за допомогою іншого string-об'єкта.
        string str4(str1);
        cout << str4;

        // Введення рядка.
        cout << "Введіть рядок: "; cin >> str4;
        cout << str4;

        getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Клас string дає змогу ефективно
Клас string дає змогу ефективно
Клас string дає змогу ефективно обробляти рядки.
str3 > str1
str3 == str1 + str2
Це рядок, який завершується нульовим символом.
Це рядок, який завершується нульовим символом.
Введіть рядок: Привіт
Привіт
```

Зверніть увагу на те, як легко тепер здійснюється оброблення рядків. Наприклад, оператор конкатенації "+" використовує для конкатенації рядків, а оператор ">" – для порівняння двох рядків. Для виконання цих операцій з використанням С-стилю оброблення рядків, тобто використання рядків, що мають завершальний нуль-символ, довелося б застосовувати менш зручні засоби, а саме викликати функції **streat**() і **strcmp**(). Оскільки С++-об'єкти типу **string** можна вільно змішувати з С-рядками, то їх (**string**-об'єкти) можна використовувати в будь-якій програмі не тільки без жодних втрат для ефективності її роботи, але навіть з помітним виграшем.

Важливо також відзначити, що у попередній програмі розмір рядків не задається. Об'єкти типу **string** автоматично отримують розмір, потрібний для зберігання заданого рядка. Таким чином, у процесі виконання операцій присвоєння або

конкатенації рядків рядок-приймач збільшиться у довжину настільки, наскільки це необхідно для зберігання нового вмісту рядка. Під час оброблення **string**-об'єктів неможливо вийти за межі рядка. Саме цей динамічний аспект **string**-об'єктів вигідно відрізняє їх від рядків, що мають завершальний нуль-символ (які часто страждають від порушення меж).

12.7.2. Огляд функцій-членів класу string

Якщо найпростіші операції з рядками можна реалізувати за допомогою операторів, то у процесі виконання дещо складніших операцій не можливо обійтися без функцій-членів класу **string**. Клас **string** містить дуже багато функцій-членів, ми ж розглянемо тут тільки найпоширеніші з поміж них.

Нео! хідноапам'ятати!аОскільки клас ***string****- контейнер, він підтримує такі звичайні контейнерні функції, як* ***begin(), end()*** *і* ***size()****.*

Основні маніпуляції над рядками. Щоб присвоїти один рядок іншому, найчастіше використовують функцію **assign**(). Ось як виглядають два можливі формати її реалізації:

string &assign(**const string** &*strob*, **size_type** *start* **size_type** *num*);

string &assign(**const char** **str,* **size_type** *num*);

Перший формат дає змогу присвоїти викликуваному об'єкту *num* символів з рядка, який задано параметром *strob*, починаючи з індексу *start*. Під час використання другого формату викликуваного об'єкта присвоюються перші *num* символів рядка, який завершується нульовим символом, які задаються параметром *str*. У кожному випадку повертається посилання на викликуваний об'єкт. Звичайно, набагато простіше для присвоєння одного повного рядка іншому використовувати оператор присвоєння "=". Про функцію-члена **assign**() згадують, в основному, тоді, коли потрібно присвоїти тільки частину рядка.

За допомогою функції-члена класу **append**() можна частину одного рядка приєднати до кінця іншого. Два можливі формати її реалізації мають такий вигляд:

string &append(**const string** *&strob,* **size_type** *start* **size_type** *num*);

string &append(**const char** **str*, **size_type** *num*);

У цих записах при використанні першого формату *num* символів з рядка, який задано параметром *strob*, починаючи з індексу *start*, буде приєднано до кінця викликуваного об'єкта. Другий формат дає змогу приєднати до кінця викликуваного об'єкта, перші *num* символів рядка, який завершується нульовим символом, який задається параметром *str*. У кожному випадку повертається посилання на викликуваний об'єкт. Звичайно, набагато простіше для приєднання одного повного рядка до кінця іншого використовувати оператор конкатенації "+". Функція ж **append**() застосовується тоді, коли необхідно приєднати до кінця викликуваного об'єкта, тільки частину рядка.

Вставлення або заміну символів у рядку можна виконувати за допомогою функцій-членів класу **insert**() і **replace**(). Ось як виглядають прототипи їх найбільш використовуваних форматів:

string &**insert**(**size_type** *start*, **const string** &*strob*);

string &**insert**(**size_type** *start*, **const string** &*strob*, **size_type** *insStart,* **size_type** *num*);

string &**replace**(**size_type** *start,* **size_type** *num,* **const string** &*strob*);

string &**replace**(**size_type** *start,* **size_type** *orgNum,* **const string** &*strob*,
size_type *replaceStart*, **size_type** *replaceNum*);

Перший формат функції **insert**() дає змогу вставити рядок, який задається параметром *strob,* у позицію викликуваного рядка, який задано параметром *start.* Другий формат функції **insert**() призначений для вставлення *num* символів з рядка, який задано параметром *strob,* починаючи з індексу *insStart,* у позицію викликуваного рядка, який задано параметром *start.*

Перший формат функції **replace**() слугує для заміни *num* символів у викликуваному рядку, починаючи з індексу *start*, рядком, який задано параметром *strob.* Другий формат дає змогу замінити *orgNum* символів у викликуваному рядку, починаючи з індексу *start, replaceNum* символами рядка, який задано параметром *strob,* починаючи з індексу *replaceStart*. У кожному випадку повертається посилання на викликуваний об'єкт.

Видалити символи з рядка можна за допомогою функції **erase**(). Один з її форматів має такий вигляд:

string &**erase**(**size_type** *start* = 0 **size_type** *num* = **npos**);

Ця функція видаляє *num* символів із викликуваного рядка, починаючи з індексу *start.* Функція повертає посилання на викликуваний об'єкт.

Використання функцій **insert**(), **erase**() і **replace**() продемонстровано у наведеному нижче коді програми.

Код програми 12.18. Демонстрація механізму використання функцій insert(), erase() і replace()

```
#include <vcl>
#include <iostream>              // Для потокового введення-виведення
#include <conio>                 // Для консольного режиму роботи
#include <string>                // Для роботи з рядковими типами
using namespace std;             // Використання стандартного простору імен

int main()
{
    string str1 ("Це простий тест.");
    string str2("ABCDEFG");
    cout << "Початкові рядки:" << endl;
    cout << "str1: " << str1 << endl;
    cout << "str2: " << str2 << endl << endl;

    // Демонструємо механізм використання функції insert().
    cout << "Вставляємо рядок str2 у рядок str1:" << endl;
    str1.insert(5, str2);
    cout << str1 << endl << endl;

    // Демонструємо механізм використання функції erase().
```

```
        cout << "Видаляємо 7 символів з рядка str1:" << endl;
        str1.erase(5, 7);
        cout << str1 <<endl << endl;

        // Демонструємо механізм використання функції replace().
        cout << "Замінюємо 2 символи в str1 рядком str2:" << endl;
        str1.replace(5, 2, str2);
        cout << str1 << endl;

        getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Початкові рядки:
str1: Це простий тест.
str2: ABCDEFG

Вставляємо рядок str2 у рядок str1:
Це пABCDEFGростий тест.

Видаляємо 7 символів з рядка str1:
Це простий тест.

Замінюємо 2 символи в str1 рядком str2:
Це пABCDEFGстий тест.
```

Пошук у рядку. У класі **string** передбачено декілька функцій-членів класу, які здійснюють пошук. Це, наприклад, такі функції, як **find**() і **rfind**(). Розглянемо прототипи найбільш використовуваних версій цих функцій:

```
size_type find(const string &strob, size_type start = 0) const;

size_type rfind(const string &strob, size_type start = npos) const;
```

Функція **find**(), починаючи з позиції *start,* проглядає викликуваний рядок на предмет пошуку першого входження рядка, який задано параметром *strob.* Якщо пошук відбувся успішно, то функція **find**() повертає індекс, за яким у викликуваному рядку було виявлено збіг. Якщо збігу не виявлено, то повертається значення **npos**. Функція **rfind**() здійснює ту ж саму дію, але з кінця. Починаючи з позиції *start,* вона проглядає викликуваний рядок у зворотному порядку на предмет пошуку першого входження рядка, який задано параметром *strob* (тобто вона знаходить у викликуваному рядку останнє входження рядка, який задано параметром *strob*). Якщо пошук відбувся вдало, то функція **find**() повертає індекс, за яким у викликуваному рядку був виявлений збіг. Якщо збігу не виявлено, то повертається значення **npos**.

Розглянемо короткий приклад використання функції **find**().

Код програми 12.19. Демонстрація механізму використання функції find()

```
#include <vcl>
#include <iostream>         // Для потокового введення-виведення
#include <conio>            // Для консольного режиму роботи
#include <string>           // Для роботи з рядковими типами
using namespace std;        // Використання стандартного простору імен
```

```
int main()
{
        int c;
        string s1 = "Клас string полегшує оброблення рядків.";
        string s2;

        c = s1.find("string");
        if(c != string::npos) {
                cout << "Збіг виявлений у позиції " << c << endl;
                cout << "Залишок рядка такий: ";
                s2.assign(s1, c, s1.size());
                cout << s2;
        }

        getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Збіг виявлений у позиції 6
Залишок рядка такий: string полегшує оброблення рядків.
```

Порівняння рядків. Щоб порівняти повний вміст одного **string**-об'єкта з іншим, зазвичай використовуються описані вище перевизначені оператори відношення. Але, якщо потрібно порівняти частину одного рядка з іншим, Вам доведеться використовувати функцію-члена **compare**():

```
int compare(size_type start size_type num, const string &strob) const;
```

Функція **compare**() порівнює із викликуваним рядком *num* символів рядка, який задано параметром *strob,* починаючи з індексу *start.* Якщо викликуваний рядок менший від рядка *strob,* то функція **compare**() поверне негативне значення. Якщо викликуваний рядок більший від рядка *strob,* то вона поверне позитивне значення. Якщо рядок *strob* дорівнює викликуваному рядку, то функція **compare**() поверне нуль.

Отримання рядка, який завершується нульовим символом. Незважаючи на незаперечну корисність об'єктів типу **string**, можливі ситуації, коли Вам доведеться отримувати з такого об'єкта символьний масив, який завершується нулем, тобто його версію С-рядка. Наприклад, Ви могли б використовувати **string**-об'єкт для створення імені файлу. Але, відкриваючи файл, Вам потрібно задати покажчик на стандартний рядок, який завершується нульовим символом. Для вирішення цього питання і використовується функція-член **c_str**(). Її прототип має такий вигляд:

```
const char *c_str() const;
```

Ця функція повертає покажчик на С-версію рядка (тобто на рядок, який завершується нульовим символом), який міститься у викликуваному об'єкті типу **string**. Отриманий рядок, який завершується нульовим символом, зміні не підлягає. Окрім цього, після виконання інших операцій над цим **string**-об'єктом допустимість застосування отриманого С-рядка не гарантується.

12.7.3. Зберігання рядків у інших контейнерах

Оскільки клас **string** визначає тип даних, то можна створити контейнери, які міститимуть об'єкти типу **string**. Розглянемо, наприклад, вдаліший варіант програми-словника, яку було показано вище.

Код програми 12.20. Демонстрація механізму використання відображення string-об'єктів для створення словника

```
#include <vcl>
#include <iostream>            // Для потокового введення-виведення
#include <conio>               // Для консольного режиму роботи
#include <map>                 // Для роботи з асоціативними контейнерами
#include <string>              // Для роботи з рядковими типами
using namespace std;           // Використання стандартного простору імен

int main()
{
	map<string, string> dictionary;

	dictionary.insert(pair<string, string>("дім", "Місце мешкання."));
	dictionary.insert(pair<string, string>("клавіатура", "Пристрій введення даних."));
	dictionary.insert(pair<string, string>("програмування", "Процес розроблення програми."));
	dictionary.insert(pair<string, string>("STL", "Standard Template Library."));

	string s;
	cout << "Введіть слово: "; cin >> s;

	map<string, string>::iterator p;

	p = dictionary.find(s);
	if(p != dictionary.end())
		cout << "Визначення: " << p->second << endl;
	else
		cout << "Такого слова у словнику немає." << endl;

	getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Введіть слово: дім
Визначення: Місце мешкання.

Введіть слово: хата
Такого слова у словнику немає.

Введіть слово: програмування
Визначення: Процес розроблення програми.

Введіть слово: STL
Визначення: Standard Template Library.
```

Розділ 13. ОСОБЛИВОСТІ РОБОТИ ПРЕПРОЦЕСОРА С++

Цей розділ навчального посібника присвячено вивченню особливостей роботи препроцесора С++. Препроцесор мови програмування С++ – це частина компілятора, яка піддає Вашу програму різним текстовим перетворенням до реальної трансляції початкового коду програми в об'єктний. Програміст може давати препроцесору команди, що задаються *директивами препроцесора* (preprocessor directives), які, не будучи формальною частиною мови програмування С++, здатні розширити область дії його середовища програмування.

Директиви препроцесора С++

#define	**#error**	**#include**
#if	**#else**	**#elif**
#endif	**#ifdef**	**#ifndef**
#undef	**#line**	**#pragma**

Як бачите, всі директиви препроцесора починаються з символу "#". Тепер розглянемо кожну з них окремо.

Нео! хідноапам'ятати! *Препроцесор мови програмування С++ є прямим нащадком препроцесора мови С, і деякі його засоби виявилися надлишковими після введення у мові програмування С++ нових елементів. Проте він, як і раніше, є важливою частиною С++-середовища програмування.*

13.1. Поняття про директиви препроцесора С++

Директива #define. Цю директиву використовують для визначення ідентифікатора та імені символьної послідовності, яка буде підставлена замість ідентифікатора скрізь, де він траплятиметься у початковому коді програми. Цей ідентифікатор називають *макроіменем,* а процес заміни – *макропідстановкою* (реалізацією макророзширення). Загальний формат використання цієї директиви має такий вигляд:

#define *макроім'я послідовність_символів*

Зверніть увагу на те, що у цьому записі немає крапки з комою. Задана *послідовність_символів* завершується тільки символом кінця рядка. Між елементами *ім'я_макроса і послідовність_символів* може бути будь-яка кількість пропусків.

---

*Директива **#define** визначає ім'я макросу.*

Отже, після внесення цієї директиви кожне входження текстового фрагмента, що є визначеним як *макроім'я,* замінюють заданим елементом *послідовність_символів.* Наприклад, якщо виникає бажання використовувати слово UP як значення 1 і слово DOWN як значення 0, оголосіть такі директиви **#define**:

```
#define UP 1
#define DOWN 0
```

Дані директиви змусять компілятор підставляти 1 або 0 кожного разу, коли у файлі початкового коду програми трапиться слово UP або DOWN відповідно. Наприклад, у процесі виконання такої настанови

```
cout << UP << " " << DOWN << " " << UP + UP;
```

на екран буде виведено таке:

```
1 0 2
```

Після визначення імені макросу його можна використовувати як частину визначення інших макроімен. Наприклад, такий програмний код визначає імена ONE, TWO і THREE і відповідні їм значення:

```
#define ONE 1
#define TWO ONE+ONE
#define THREE ONE+TWO
```

Важливо розуміти, що макропідстановка – це просто заміна ідентифікатора відповідним рядком. Отже, якщо виникає потреба визначити стандартне повідомлення, то використовують програмний код, подібний до цього:

```
#define GETFILE "Введіть ім'я файлу"
//...
```

Препроцесор замінить рядком "Введіть ім'я файлу" кожне входження ідентифікатора GETFILE. Для компілятора ця **cout**-настанова

```
cout << GETFILE;
```

насправді виглядає так:

```
cout << "Введіть ім'я файлу";
```

Ніякої текстової заміни не відбудеться, якщо ідентифікатор знаходиться в рядку, поміщеному в лапки. Наприклад, у процесі виконання такого коду програми

```
#define GETFILE "Введіть ім'я файлу"
//...
cout << "GETFILE – це макроім'я" << endl;
```

на екрані буде відображена ця інформація

```
GETFILE – це макроім'я
```

а не така:

```
Введіть ім'я файлу – це макроім'я
```

Якщо текстова послідовність не поміщається в рядку, то її можна продовжити в наступному, поставивши зворотну косу риску у кінці рядка, як це показано у такому прикладі:

```
#define LONG_STRING "Це дуже довга послідовність, \
яка використовується як приклад."
```

Серед С++-програмістів прийнято використовувати для макроімен прописні букви. Ця домовленість дає змогу з першого погляду зрозуміти, що тут використовується макропідстановка. Окрім цього, найкраще помістити всі директиви **#define** у початок файлу або включити в окремий файл, щоб не шукати їх потім у всій програмі.

Макропідстановки часто використовують для визначення "магічних чисел" програми. Наприклад, у Вас є програма, яка визначає певний масив, і ряд функцій, які отримують доступ до нього. Замість "жорсткого" кодування розміру масиву за допомогою константи краще визначити ім'я, яке б представляло розмір, а потім використовувати це ім'я скрізь, де повинен знаходитися розмір масиву. Тоді, якщо цей розмір доведеться змінити, Вам достатньо буде внести тільки одну зміну, а потім перекомпілювати програму. Розглянемо такий приклад:

```
#define Max_size 100
//...
float balance[Max_size];
double index[Max_size];
int num_emp[Max_size];
```

Нео! хідноа пам'ятати!а *У мові програмування С++ передбачено ще один спосіб визначення констант, який полягає у використанні специфікатора* **const**. *Проте багато програмістів "прийшли" у мову програмування С++ з С-середовища, де для цих потреб зазвичай використовувалася директива* **#define**. *Тому Вам ще часто доведеться з нею мати справу у С++-коді програми.*

Макровизначення, що діють як функції. Директива **#define** має ще одне призначення: макроім'я може використовуватися з аргументами. При кожному входженні макроімені пов'язані з ним аргументи замінюються реальними аргументами, вказаними у коді програми. Такі макровизначення діють подібно до функцій. Розглянемо такий приклад.

Код програми 13.1. Демонстрація механізму використання "функціональних" макровизначень

```
#include <iostream>          // Для потокового введення-виведення
using namespace std;         // Використання стандартного простору імен

#define MIN(a, b) (((a)<(b))? a: b)

int main()
{
	int x = 10; y = 20;
	cout << "Мінімум дорівнює: " << MIN(x, y);

	getch(); return 0;
}
```

У процесі компілювання цієї програми вираз, визначений ідентифікатором MIN(a, b), буде замінено, але x і y розглядатимуться як операнди. Це означає, що з **out**-настанова після компілювання виглядатиме так:

```
cout << "Мінімум дорівнює: " << (((x)<(y))? x: y);
```

По суті, таке макровизначення є способом визначити функцію, яка замість виклику дає змогу розкрити свій код у рядку.

Макровизначення, що діють як функції, – це макровизначення, які приймають аргументи. Круглі дужки, що здаються надлишковими, в які поміщено макровизначення MIN, необхідні для того, щоби гарантувати правильне сприйняття компілятором замінюваного виразу. Насправді додаткові круглі дужки повинні

застосовуватися практично до всіх макровизначень, що діють подібно до цієї функцій. Потрібно завжди дуже уважно ставитися до визначення таких макросів; інакше можливе отримання несподіваних результатів. Розглянемо, наприклад, цю навчальну програму, яка використовує макрос для визначення парності значення.

Код програми 13.2. Демонстрація неправильної роботи коду програми

```
#include <iostream>              // Для потокового введення-виведення
using namespace std;             // Використання стандартного простору імен

#define EVEN(a) a%2==0? 1 : 0

int main()
{
        if(EVEN(9 + 1)) cout << "парне число";
        else cout << "непарне число ";

        getch(); return 0;
}
```

Ця програма не працюватиме коректно, оскільки не забезпечена правильна підстановка значень. У процесі компілювання вираз EVEN(9 + 1) буде замінений таким чином:

```
9+1 %2==0 ? 1 : 0
```

Нагадаю, що оператор ділення за модулем "%" має вищий пріоритет, ніж оператор додавання "+". Це означає, що спочатку виконається операція ділення за модулем (%) для числа 1, а потім її результат буде складний з числом 9, що (звичайно ж) не дорівнює 0. Щоб виправити помилку, достатньо помістити у круглі дужки аргумент, а в макровизначенні EVEN, як це показано в наступній (виправлений) версії тієї ж самої програми.

Код програми 13.3. Демонстрація коректної роботи коду програми

```
#include <iostream>              // Для потокового введення-виведення
using namespace std;             // Використання стандартного простору імен

#define EVEN(a) (a)%2==0? 1 : 0

int main()
{
        if(EVEN(9 + 1)) cout << "парне число";
        else cout << "непарне число";

        getch(); return 0;
}
```

Тепер сума 9+1 обчислюється до виконання операції ділення за модулем. У загальному випадку краще завжди брати параметри макровизначення у круглі дужки, щоб уникнути непередбачених результатів, подібних описаному вище.

Використання макровизначень замість справжніх функцій має одну істотну перевагу: оскільки програмний код макровизначення розширюється в рядку, і немає ніяких витрат системних ресурсів на виклик функції, то швидкість роботи Вашої програми буде вищою порівняно із застосуванням звичайної функції. Але під-

вищення швидкості є платою за збільшення розміру програми (через дублювання коду функції).

*Нео! хідноа пам'ятати!а Незважаючи на те, що макровизначення все ще трапляється у С++-коді програми, макроси, що діють подібно до функцій, можна замінити специфікатором **inline**, який справляється з тією ж функцією краще і безпечніше. (Пригадайте: специфікатор **inline** забезпечує замість виклику функції розширення її тіла в рядку). Окрім цього, **inline**-функції не вимагають додаткових круглих дужок, без яких не можуть обійтися макровизначення. Проте макроси, що діють подібно до функцій, все ще залишаються частиною С++-програм, оскільки багато хто з С/С++-програмістів продовжує використовувати їх за звичкою.*

Директива #error. Ця директива дає вказівку компіляторові зупинити компілювання. Вона використовується здебільшого для відлагодження. Загальний формат її запису є таким:

#error *повідомлення*

Зверніть увагу на те, що елемент *повідомлення* не поміщений у подвійні лапки. При потраплянні на директиву **#error** відображається задане *повідомлення* та інша інформація (вона залежить від конкретної реалізації робочого середовища), після чого компілювання припиняється. Щоб дізнатися, яку інформацію відображає у цьому випадку компілятор, достатньо провести експеримент.

*Директива **#error** відображає повідомлення про помилку.*

Директива препроцесора #include. Ця директива зобов'язує компілятор включити або стандартний заголовок, або інший початковий файл, ім'я якого вказане у директиві **#include**. Ім'я стандартних заголовків береться у кутові дужки, як це показано у прикладах, наведених у цьому навчальному посібнику. Наприклад, ця директива

```
#include <vector>          // Для роботи з контейнерним класом "Вектор"
```

містить стандартний заголовок для векторів.

*Директива **#include** містить заголовок або інший початковий файл.*

Під час включення іншого початкового файлу його ім'я може бути вказане у подвійних лапках або кутових дужках. Наприклад, наступні дві директиви зобов'язують компілятор С++ прочитати і скомпілювати файл з іменем sample.h:

```
#include <sample.h>
#include "sample.h"
```

Якщо ім'я файлу поміщене у кутові дужки, то пошук файлу здійснюватиметься в одному або декількох спеціальних каталогах, визначених конкретною реалізацією.

Якщо ж ім'я файлу поміщене в лапки, пошук файлу здійснюється, як правило, у поточному каталозі (що також визначено конкретною реалізацією). У багатьох випадках це означає пошук поточного робочого каталога. Якщо заданий файл не знайдено, то пошук повторюється з використанням першого способу (неначебто ім'я файлу було поміщено у кутові дужки). Щоб ознайомитися з подроби-

цями, пов'язаними з різною обробкою директиви **#include** у разі використання кутових дужок і подвійних лапок, зверніться до настанови користувача, яка додається до Вашого компілятора. Настанови **#include** можуть бути вкладеними в інші поміщені файли.

13.2. Директиви умовного компілювання

Існують директиви, які дають змогу вибірково компілювати частини початкового коду програми. Цей процес, названий *умовною компіляцією,* широко використовують комерційні фірми з розроблення програмного забезпечення, які створюють і підтримують багато різних версій однієї програми.

Директиви #if, #else, #elif і #endif. Головна ідея полягає у тому, що коли вираз, який знаходиться після директиви **#if** виявляється істинним, то буде скомпільований програмний код, розташований між нею і директивою **#endif**; інакше цей програмний код буде опущений. Директиву **#endif** використовують для позначення кінця блоку **#if**.

---

*Директиви **#if**, **#ifdef**, #ifndef, **#else**, **#elif** і **#endif** – це директиви умовного компілювання.*

Загальна форма запису директиви **#if** має такий вигляд:

```
#if константний_вираз
    послідовність настанов
#endif
```

Якщо *константний_вираз* є істинним, то програмний код, розташований безпосередньо за цією директивою, буде скомпільований. Розглянемо такий приклад.

Код програми 13.4. Демонстрація механізму використання директиви #if

```
#include <iostream>          // Для потокового введення-виведення
using namespace std;         // Використання стандартного простору імен

#define MAX 100

int main()
{
    #if MAX>10
        cout << "Потрібна додаткова память" << endl;
    #endif
    //...

    getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані повідомлення

```
Потрібна додаткова пам'ять
```

оскільки, як визначено у програмі, значення константи MAX більше 10. Цей приклад ілюструє важливий момент: вираз, який знаходиться після директиви **#if**, обчислюється при компілюванні. Отже, воно повинно містити тільки ідентифікато-

ри, які були заздалегідь визначені, або константи. Використання ж змінних тут виключене.

Поведінка директиви **#else** багато в чому подібна до поведінки настанови **else**, яка є частиною мови програмування С++: вона визначає альтернативу на випадок невиконання директиви **#if**. Щоб показати, як працює директива **#else**, скористаємося попереднім прикладом, дещо його розширивши.

Код програми 13.5. Демонстрація механізму використання директив #if / #else

```
#include <iostream>          // Для потокового введення-виведення
using namespace std;         // Використання стандартного простору імен

#define MAX 6

int main()
{
    #if MAX>10
        cout << "Потрібна додаткова пам'ять" << endl;);
    #else
        cout << "Достатньо наявної пам'яті" << endl;
    #endif
    //...

    getch(); return 0;
}
```

У цьому коді програми для імені MAX визначено значення, яке менше 10, тому **#if**-гілка коду програми не відкомпілюється, проте відкомпілюється альтернативна **#else**-гілка. Як наслідок, відобразиться повідомлення:

```
Достатньо наявної пам'яті.
```

Зверніть увагу на те, що директиву **#else** використовують для індикації одночасно як кінця **#if**-блоку, так і початку #else-блоку. У цьому є логічна необхідність, оскільки тільки одна директива **#endif** може бути пов'язана з директивою **#if**.

Директива **#elif** еквівалентна зв'язці настанов **else-if** і використовують для формування багатоланкової схеми **if-else-if**, яка представляє декілька варіантів компілювання. Після директиви **#elif** повинен знаходитися константний вираз. Якщо цей вираз істинний, то наступний блок коду програми відкомпілюється, і ніякі інші **#elif**-вирази не тестуватимуться або не компілюватимуться. Інакше буде перевірений наступний по черзі **#elif**-вираз. Ось як виглядає загальний формат використання директиви **#elif**:

```
#if вираз
    послідовність настанов
#elif вираз 1
    послідовність настанов
#elif вираз 2
    послідовність настанов
#elif вираз 3
    послідовність настанов
```

```
    //...
#elif вираз N
    послідовність настанов
#endif
```

Наприклад, наведений нижче фрагмент коду програми використовує ідентифікатор COMPILED_BY, який дає змогу визначити, хто компілює програму:

```
#define JOHN 0
#define BOB 1
#define TOM 2

#define COMPILED_BY JOHN

#if COMPILED_BY == JOHN
    char Show[] = "John";
#elif COMPILED_BY == BOB
    char Show[] = "Bob";
#else
    char Show[] = "Tom";
#endif
```

Директиви **#if** і **#elif** можуть бути вкладеними. У цьому випадку директива **#endif**, **#else** або **#elif** зв'язується з найближчою директивою **#if** або **#elif**. Наприклад, такий фрагмент коду програми є абсолютно допустимим:

```
#if COMPILED_BY == BOB
    #if DEBUG == FULL
        int port = 198;
    #elif DEBUG == PARTIAL
        int port = 200;
    #endif
#else
    cout << "Борис повинен скомпілювати код"
        << "для відлагодження виведення даних" << endl;
#endif
```

Директиви #ifdef і #ifndef. Ці директиви пропонують ще два варіанти умовного компілювання, які можна виразити як "якщо визначено" і "якщо не визначено" відповідно. Загальний формат використання директиви **#ifdef** такий:

```
#ifdef макроім'я
    послідовність настанов
#endif
```

Якщо *макроім'я* заздалегідь визначено за допомогою якої-небудь директиви **#define**, то *послідовність настанов,* розташована між директивами **#ifdef** і **#endif**, буде скомпільована.

Загальний формат використання директиви **#ifndef** такий:

```
#ifndef макроім'я
    послідовність настанов
#endif
```

Якщо *макроім'я* не визначене за допомогою якої-небудь директиви **#define**, то *послідовність настанов,* розташована між директивами **#ifdef** і **#endif**, буде скомпільована.

Як директива **#ifdef**, так і директива **#ifndef** може мати директиву **#else** або **#elif**. Розглянемо такий приклад.

```
#include <iostream>          // Для потокового введення-виведення
using namespace std;         // Використання стандартного простору імен

#define TOM

int main()
{
	#ifdef TOM
		cout << "Програміст Том" << endl;
	#else
		cout << "Програміст невідомий" << endl;
	#endif

	#ifndef RALPH
		cout << "Ім'я RALPH не визначене" << endl;
	#endif

	getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Програміст Том.
Ім'я RALPH не визначене.
```

Але якби ідентифікатор TOM не було визначено, то результат виконання цієї програми виглядав би так:

```
Програміст невідомий.
Ім'я RALPH не визначене.
```

Варто знати! *Директиви* ***#ifdef*** *і* ***#ifndef*** *можна вкладати так само, як і директиви* ***#if****.*

Директива #undef. Цю директиву використовують для видалення попереднього визначення певного макроімені. Її загальний формат є таким:

#undef *макроім'я*

Розглянемо такий приклад:

```
#define TIMEOUT 100
#define WAIT 0
//...
#undef TIMEOUT
#undef WAIT
```

У цих записах імена TIMEOUT і WAIT визначені доти, доки не виконається директива **#undef**.

Основне призначення директиви **#undef** – дати змогу локалізувати макроімена для тих частин коду програми, в яких вони потрібні.

Використання оператора defined. Окрім директиви **#ifdef**, існує ще один спосіб з'ясувати, чи визначене у програмі певне макроім'я. Для цього можна використовувати директиву **#if** у поєднанні з оператором часу компілювання **defined**. Наприклад, щоб дізнатися, чи визначено макроім'я MYFILE, можна використовувати одну з таких команд препроцесорного оброблення:

```
#if defined MYFILE
```

або

```
#ifdef MYFILE
```

У разі потреби, щоб реверсувати умову перевірки, операторові **defined** може передувати символ "!". Наприклад, такий фрагмент коду програми відкомпілюється тільки у тому випадку, якщо макроім'я DEBUG не визначене:

```
#if !defined DEBUG
    cout << "Остаточна версія!" << endl;
#endif
```

Про значення препроцесора. Як ми уже зазначали раніше, препроцесор мови програмування С++ – прямий спадкоємець препроцесора мови С, до того ж без жодних удосконалень. Проте його вагомість у мові програмування С++ набагато менша, ніж препроцесора у мові С. Йдеться про те, що багато завдань, які виконує препроцесор у мові С, реалізовані мовою С++ у вигляді елементів мови. Страуструп тим самим виразив свій намір зробити функції препроцесора непотрібними, щоб врешті-решт від нього можна було зовсім звільнити мову.

На цьому етапі препроцесор вже частково надлишковий. Наприклад, дві найбільш використовувані властивості директиви **#define** було замінено настановами мови С++. Зокрема, її здатність створювати константне значення і визначати макровизначення, що діє подібно до функцій, тепер абсолютно надлишкова. У мові програмування С++ є ефективніші засоби для виконання цих завдань. Для створення константи достатньо визначити **const**-змінну. А із створенням вбудованої (що підставляється) функції легко справляється специфікатор **inline**. Обидва ці засоби краще працюють, ніж відповідні механізми директиви **#define**.

Наведемо ще один приклад заміни елементів препроцесора елементами мови. Він пов'язаний з використанням однорядкового коментарю. Одна з причин його створення – дозволити "перетворення" коду програми у коментар. Як уже зазначалося вище, коментар, що використовує /*...*/-стиль, не може бути вкладеним. Це означає, що фрагменти коду програми, що містять /*...*/-коментарі, одним махом "перетворити на коментар" не можна. Але це можна зробити з //-коментарями, оточивши їх /*...*/-символами коментарю. Можливість "перетворення" коду програми у коментар робить використання таких директив умовного компілювання, як **#ifdef**, частково надлишковим.

Директива #line. Цю директиву використовують для зміни вмісту псевдозмінних **__LINE__** і **__FILE__**, які є зарезервованими ідентифікаторами (макроіменами). Псевдозмінна **__LINE__** містить номер скомпільованого рядка, а псевдозмінна **__FILE__** – ім'я компільованого файлу. Базова форма запису цієї команди має такий вигляд:

#line *номер "ім'я_файлу"*

У цьому записі *номер* – це будь-яке позитивне ціле число, а *ім'я_файлу* – будь-який допустимий ідентифікатор файлу. Значення елемента номер стає номером поточного початкового рядка, а значення елемента *ім'я_файлу* – іменем початкового файлу. Ім'я файлу – елемент необов'язковий. Директива **#line** використовується, в основному, з метою відлагодження і у спеціальних додатках.

---

*Директива **#line** змінює вміст псевдозмінних **__LINE__** і **__FILE__**.*

Наприклад, наведений нижче код програми зобов'язує починати рахунок рядків з позиції 200. Настанова **cout** відображає номер 202, оскільки це третій рядок у програмі після директивної настанови **#line** 200.

```
#include <iostream>           // Для потокового введення-виведення
using namespace std;          // Використання стандартного простору імен

#line 200 // Встановлюємо лічильник рядків, що дорівнює 200.

int main()          // Цей рядок зараз має номер 200.
{                   // Номер цього рядка дорівнює 201.
    cout << __LINE__; // Тут виводиться номер 202.

    getch(); return 0;
}
```

Директива #pragma. Робота цієї директиви залежить від конкретної реалізації компілятора. Вона дає змогу видавати компіляторові різні настанови, передбачені творцем компілятора. Загальний формат його використання такий:

#pragma *ім'я*

У цьому записі елемент *ім'я* представляє ім'я бажаної **#pragma**-настанови. Якщо вказане ім'я не розпізнається компілятором, директива **#pragma** просто ігнорується без повідомлення про помилку.

---

*Директива **#pragma** залежить від конкретної реалізації компілятора.*

Нео! хідноапам'ятати!а*Для отримання детальної інформації про можливі варіанти використання директиви **#**pragma варто звернутися до системної документації з використовуваного Вами компілятора. Ви можете знайти для себе дуже корисну інформацію. Звичайно **#**pragma-настанови дають змогу визначити, які застережні повідомлення видає компілятор, як генерується програмний код і які бібліотеки компонуються з Вашими програмами.*

13.3. Оператори препроцесора "#" і "##"

У мові програмування С++ передбачено підтримку двох операторів препроцесора: "#" і "##". Ці оператори використовують спільно з директивою **#define**. Оператор "#" перетворить наступний за ним аргумент у рядок, поміщений у лапки. Розглянемо, наприклад, такий код програми:

```
#include <iostream>           // Для потокового введення-виведення
using namespace std;          // Використання стандартного простору імен

#define mkstr(s) # s
```

```
int main()
{
        cout << mkstr(Я у захопленні від мови програмування С++);

        getch(); return 0;
}
```

Препроцесор мови програмування С++ перетворить рядок

```
cout << mkstr(I like C++);
```

у рядок

```
cout << "Я у захопленні від мови програмування С++";
```

Оператор "##" використовують для конкатенації двох лексем. Розглянемо такий приклад:

```
#include <iostream>              // Для потокового введення-виведення
using namespace std;             // Використання стандартного простору імен

#define concat(a, b) a ## b

int main()
{
        int xy = 10;
        cout << concat(x, y);

        getch(); return 0;
}
```

Препроцесор перетворить рядок

```
cout << concat(x, y);
```

у рядок

```
cout << xy;
```

Якщо ці оператори Вам здаються дивними, потрібно пам'ятати, що вони не є операторами "повсякденного попиту" і рідко використовуються у програмах. Їх основне призначення – дати змогу препроцесору обробляти деякі спеціальні ситуації.

13.4. Зарезервовані макроімена

У мові С++ визначено шість вбудованих макроімен:

__LINE__
__FILE__
__DATE__
__TIME__
__STDC__
__cplusplus

Розглянемо кожне з них окремо.

Макроси **__LINE__** і **__FILE__** описані під час розгляду директиви **#line** вище у цьому розділі. Вони містять номер поточного рядка та ім'я файлу компільованої програми.

Макрос **__DATE__** – це рядок у форматі *місяць/день/рік,* який означає дату трансляції початкового файлу в об'єктний код.

Час трансляції початкового файлу в об'єктний код міститься у вигляді рядка в макросі **__TIME__**. Формат цього рядка такий: *години:хвилини:секунди.*

Точне призначення макросу **__STDC__** залежить від конкретної реалізації компілятора. Як правило, якщо макрос **__STDC__** визначено, то компілятор прийме тільки стандартний С/С++-код, який не містить ніяких нестандартних розширень.

Компілятор, що відповідає ANSI/ISO-стандарту мови програмування С++, визначає макрос **__cplusplus** як значення, що містить принаймні шість цифр. "Нестандартні" компілятори повинні використовувати значення, що містить п'ять (або навіть менше) цифр.

* * *

Ми подолали чималий шлях – завдовжки в цілу книгу. Якщо Ви уважно вивчили всі наведені тут навчальні приклади, то можете сміливо назвати себе фахівцем з програмування мовою С++. Подібно до багатьох інших наук, процес програмування найкраще освоювати на практиці, тому тепер Вам потрібно писати якомога більше програм. Корисно також розібратися у С++-програмах, написаних іншими (причому різними) професійними програмістами. Важливо звертати увагу також на те, як програма оформлена і реалізована. Постарайтеся знайти в них як переваги, так і недоліки. Це розширить діапазон Ваших уявлень про різні стилі програмування. Подумайте також над тим, як можна поліпшити розглянутий Вами код будь-якої програми, застосувавши контейнери і алгоритми бібліотеки STL. Ці засоби, як правило, дають змогу покращити читабельність і підтримку великих програм. Нарешті, чим більше Ви експериментуєте, тим більше у Вас виникає власних ідей реалізації тієї чи іншої програми. Дайте волю своїй фантазії і незабаром Ви відчуєте себе справжнім С++-програмістом.

Для продовження теоретичного засвоєння мови програмування С++ пропонуємо звернутися до перекладу книги Герберта Шілдта "*Полный справочник по С++"* [27]. Вона містить детальний опис елементів мови програмування С++ і бібліотек. Також можна використати й інші книги, які так чи інакше наблизять Вас до засвоєння вершин мови програмування С++ (див. рекомендований перелік літературних джерел).

Розділ 14. ФОРМАЛІЗАЦІЯ ПРОЦЕСУ РОЗРОБЛЕННЯ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Приклади кодів програм, які було наведено в цьому навчальному посібнику, настільки малі за обсягом, що не вимагають якої-небудь формалізації процесу їх розроблення. Проте ситуація різко змінюється, коли ми маємо справу із справжньою, масштабною програмою, над створенням якої трудяться десятки або сотні програмістів і в якій містяться мільйони рядків початкового коду програми. У таких випадках дуже важливо чітко дотримуватися певної концепції розроблення сучасного програмного забезпечення (ПЗ). У цьому розділі ми достатньо стисло розглянемо приклад процесу розроблення програми, а потім продемонструємо, як цю технологію застосовують до справжніх програм.

У цьому навчальному посібнику є багато прикладів діаграм UML. Проте UML призначено не для створення програми; це всього тільки мова візуального моделювання її структури. Але UML, як ми побачимо далі, може мати ключове значення в процесі роботи над великим проектом.

14.1. Удосконалення процесу розроблення сучасного програмного забезпечення

Ідея формалізації процесу розроблення програмного забезпечення розвивалася протягом десятиліть. У цьому навчальному посібнику спробуємо тільки стисло розглянути основні віхи історії.

Безпосередній процес розроблення ПЗ. Ще на початку удосконалення обчислювальної техніки ніяких формальностей процесу розроблення ПЗ не було. Здебільшого програміст обговорював фізичний зміст технічного завдання з конкретним замовником чи потенційними користувачами і відразу ж брався писати код програми. Це було, втім, цілком прийнятно навіть для великих, як на той час, програм.

Каскадний процес розроблення ПЗ. Коли програмісти стали більш кваліфікованими фахівцями, то вони почали ділити процес розроблення ПЗ на декілька етапів, що мали б виконуватися послідовно. Цю ідею насправді було запозичено з виробничих процесів. Етапи були такі: аналіз, планування, кодування і впровадження. Така послідовність часто називалася *каскадною моделлю,* оскільки процес завжди йшов у одному напрямку від аналізу до впровадження, як показано на рис. 14.1. Для реалізації кожного з етапів почали залучати окремі групи програмістів, і кожна з них передавала результати своєї праці наступній.

Набутий досвід показав, що каскадна модель має дуже багато недоліків. Адже малося на увазі, що кожний з етапів здійснюється без помилок або з мінімальною їх кількістю. Так, зазвичай, в повсякденній роботі програмістів майже не бу-

ває. Кожен етап привносив свої помилки, їхня кількість від етапу до етапу наростала, як снігова лавина, роблячи всю програму суцільною великою помилкою.

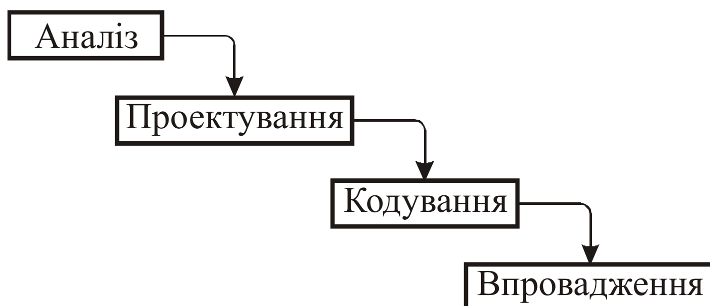

Рис. 14.1. Каскадна модель розроблення ПЗ

До того ж, у процесі розроблення ПЗ замовник міг змінити свої вимоги, а після закінчення етапу планування вже складно було знову повернутися до нього. У підсумку виявлялося, що до моменту завершення процесу написання програма вже просто ставала застарілою.

Використання ООП. Як уже було сказано в розд. 1, ООП створювалося для вирішення деяких проблем, що є притаманними процесу удосконалення методики розроблення великих програм. Зрозуміло, процес планування під час використання ООП різко спрощується, оскільки об'єкти програми відповідають об'єктам реального світу.

Але саме ООП не вказує нам, що повинна робити програма; воно придатне тільки після того, як визначено мету і завдання проекту. Початковим, як і завжди, є етап "ініціалізації", коли потрібно з'ясувати вимоги замовника і чітко уявити собі потреби потенційних користувачів. Тільки після цього можна починати планування об'єктно-орієнтованої програми. Але як же нам зробити цей перший крок?

Сучасні підходи до розроблення ПЗ. За останні роки з'явилося безліч нових концепцій розроблення ПЗ. Вони визначають певну послідовність дій і способи взаємодії замовників, постановників завдань, розробників і програмістів. На сьогодні жодна мова моделювання не володіє тією універсальністю, яка властива UML. Багато експертів дотепер не можуть повірити, що один і той же підхід до розроблення ПЗ може бути застосований до створення проектів будь-яких видів. Навіть коли вже вибраний якийсь процес, то може з часом виникнути нагальна потреба, залежно від застосування програми, достатньо серйозно його змінити. Як приклад сучасної методики розроблення ПЗ, розглянемо основні ознаки підходу, якому ми дамо назву *уніфікований процес*.

Уніфікований процес було розроблено тими ж фахівцями, які створили UML: Граді Бучем[1] (Grady Booch), Айваром Джекобсоном (Ivar Jacobson) і Джеймсом Рамбо[2] (James Rumbaugh) [33, 34]. Іноді цю концепцію називають *раціональним уніфікованим процесом* (Rational Unified Process, за назвою фірми, в якій його було розроблено) або *уніфікованим процесом розроблення ПЗ*.

---

[1] Граді Буч (Grady Booch; *27 лютого 1955 р., Техас) – американський вчений в галузі інформаційних технологій і програмування. Автор класичних праць з об'єктно-орієнтованого аналізу. Один з творців UML.

[2] Джеймс Рамбо (англ. James Rumbaugh, народ. 1947 р.) – американський учений в області інформатики і об'єктної методології, найбільш відомий за своїми роботами над створенням технології об'єктного моделювання (OMT) і мови моделювання UML.

Уніфікований процес розроблення ПЗ поділяється на чотири етапи: початок; удосконалення; побудова; передача.

На початковому етапі виявляються загальні можливості роботи майбутньої програми і здійсненність її функцій. Етап закінчується схваленням проекту. На етапі удосконалення планується загальна архітектура системи. Саме тут визначають вимоги користувачів. На етапі побудови здійснюють планування окремих деталей системи і пишуть власне код програми. На етапі впровадження система представляється кінцевим користувачам, тестується та впроваджується.

Всі чотири етапи можуть бути розбиті на ряд так званих *ітерацій.* Зокрема, етап побудови складається з декількох ітерацій. Кожну з них є підмножиною всієї системи і відповідає певним завданням, поставленим замовником[1]. На рис. 14.2 показано етапи уніфікованого процесу.

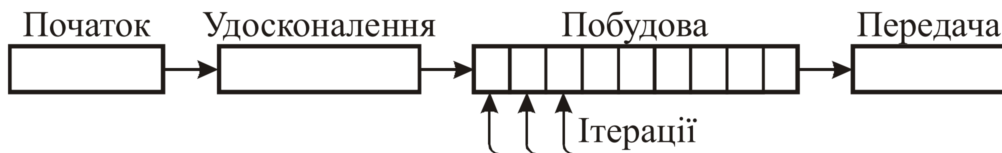

Рис. 14.2. Етапи уніфікованого процесу розроблення ПЗ

Кожна ітерація включає свою власну послідовність етапів аналізу, планування, реалізації і тестування. Ітерації можуть повторюватися кілька разів. Метою ітерації є створення діючої частини системи.

На відміну від каскадного процесу розроблення ПЗ, уніфікований процес дає змогу повернутися на попередні етапи розроблення. Наприклад, зауваження, зроблені користувачами на етапі передачі, повинні бути враховані на попередніх етапах, що приводить до перегляду етапу побудови і, можливо, етапу удосконалення.

Необхідно зазначити, що розглядувана концепція уніфікованого процесу розроблення ПЗ може бути застосована до будь-яких типів програмної архітектури, а не тільки до проектів, у яких використовуються об'єктно-орієнтовані мови. Насправді, напевно, якраз слабкою стороною цього підходу є не дуже активне використання ООП.

Етап удосконалення зазвичай за основу бере технологію *варіантів використання* (**use case**). Це відправна точка детального розроблення системи. З цієї причини уніфікований процес розроблення ПЗ називають *прецедентним*. У наступному підрозділі розглядатимемо цю технологію, а потім застосуємо її до проекту, узятого нами як приклад.

14.2. Моделювання варіантів використання

Моделювання варіантів використання дає змогу майбутнім користувачам найактивніше брати участь в розробленні ПЗ. При цьому підході за основу береться термінологія користувача, а не програміста. Це гарантує взаєморозуміння між замовниками і системними інженерами. У моделюванні варіантів використання застосовуються дві основні сутності: діючі суб'єкти і варіанти використання. Давайте з'ясуємо ці нюанси.

---

[1] Як ми побачимо далі, ітерації зазвичай відповідають варіантам використання.

Поняття про діючі суб'єкти. Діючий суб'єкт – це в більшості випадків просто той користувач, який буде реально працювати із створюваною нами системою. Наприклад, діючим суб'єктом є покупець, що користується торговим автоматом. Астроном, що вводить координати зірки у програму автоматизації телескопа, – також діючий суб'єкт. Продавець у книжковому магазині, який перевіряє по базі даних наявність навчального посібника, також може виступати як діючий суб'єкт. Зазвичай цей користувач ініціює якусь подію у програмі, яку-небудь операцію, але може бути і "приймачем" інформації, що видається програмою. Окрім цього, він може супроводжувати і контролювати проведення операції.

Насправді більш відповідною назвою, ніж "діючий суб'єкт" або "актор", є, можливо, "роль". Тому що один користувач може в різних життєвих ситуаціях грати різні ролі. Приватний підприємець Петренко може зранку бути продавцем у своєму маленькому магазині, а увечері – бухгалтером, що вводить дані про продаж за день. Навпаки, один діючий суб'єкт може представлятися різними людьми. Протягом робочого дня Василь і Василина Петренки є продавцями у своєму магазинчику.

Системи, що взаємодіють з нашою, наприклад інший комп'ютер локальної мережі або web-сервер, можуть бути діючими суб'єктами. Наприклад, комп'ютерна система магазину книжкової торговельної мережі може бути пов'язана з видаленою системою в головному офісі. Остання є діючим суб'єктом стосовно першої.

Під час розроблення великого проекту складно буває визначити, які саме діючі суб'єкти можуть знадобитися. Розробник повинен розглядати кандидатів на ці ролі з погляду їх взаємодії з системою:

- чи вводять вони дані;
- чи очікують приходу інформації від системи;
- чи допомагають іншим діючим суб'єктам.

Поняття про варіанти використання. *Варіант використання* – це спеціальне завдання, які зазвичай ініціює діючий суб'єкт. Він описує єдину мету, яку необхідно на поточний момент досягти. Прикладами можуть слугувати такі операції, як зняття грошей з внеску клієнтом банку, націлювання телескопа астрономом, з'ясування інформації про наявність навчального посібника у книжковому магазині його продавцем.

У більшості ситуацій, як ми вже сказали раніше, варіанти використання генеруються діючими суб'єктами, але іноді їх ініціює сама система. Наприклад, комп'ютерна система ЛьвівЕнергоЗбуту може прислати на Вашу адресу нагадування про те, що пора заплатити за використовування електроенергії. Електронна система (після вбудовування її у Ваш "Запорожець") може повідомити про перегрів двигуна, наприклад, запаленням контрольної лампи на панелі приладів.

У цілому все, що повинна робити система, повинно бути описано за допомогою варіантів використання на етапі її розроблення.

Поняття про сценарії. Варіант використання складається в більшості випадків з набору *сценаріїв.* Тоді як варіант використання визначає мету операції, сценарій описує спосіб досягнення цієї мети. Припустимо, варіант використання полягає в тому, що службовець книжкового магазину запитує у системи, де саме на

складі знаходиться конкретний навчальний посібник. Існує декілька варіантів вирішення цієї задачі (декілька сценаріїв):

- книга є на складі; комп'ютер виводить на екран номер полиці, на якій вона стоїть;
- книги немає на складі, але система дає клієнту змогу замовити її з видавництва;
- навчального посібника не тільки немає на складі, його немає взагалі. Система інформує клієнта про те, що йому не поталанило.

Якщо відповідально підходити до процесу розроблення комп'ютерної системи, то кожен сценарій повинен супроводжуватися своєю документацією, в якій детально описано всякі події.

Застосування діаграм варіантів використання. За допомогою UML можна будувати діаграми варіантів використання. Діючі суб'єкти зображаються чоловічками, варіанти використання – еліпсами. Прямокутна рамка оточує всі варіанти використання, залишаючи за своїми межами діючі суб'єкти. Цей прямокутник називають *межею системи.* Те, що знаходиться всередині, – програмне забезпечення, яке розробник намагається створити. На рис. 14.3 показано діаграму варіантів використання для комп'ютерної системи книжкового магазину.

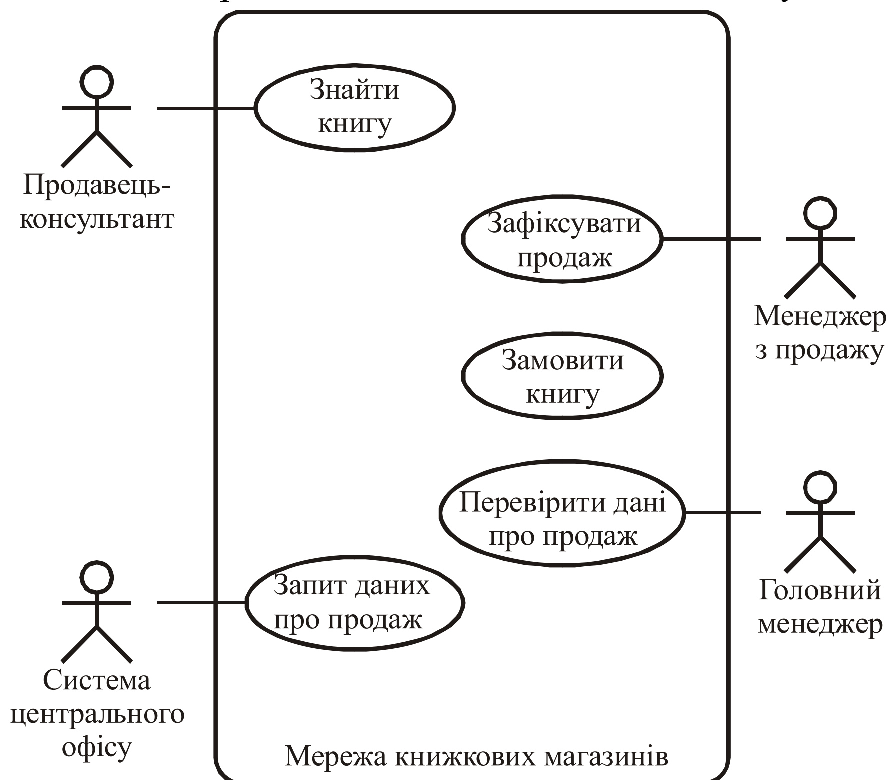

Рис. 14.3. Діаграма варіантів використання для книжкового магазину

На цій діаграмі лінії, які називають *асоціаціями,* з'єднують діючі суб'єкти з їх варіантами використання. У загальному випадку асоціації не є спрямованими, і на лініях немає стрілок, але можна їх вставити для того, щоби наочно показати той діючий суб'єкт, який є ініціатором варіанта використання.

Будемо вважати, що книжковий магазин – це частина торговельної мережі, і бухгалтерія, і подібні функції виконуються в центральному офісі. Службовці магазину записують покупку кожного навчального посібника і запитують інформацію про наявність продукції і його місцезнаходження. Менеджер може проглянути дані про те, які навчальні посібники продано, і замовити у видавництві ще певну кількість їх екземплярів. Діючими суб'єктами системи є продавець, консуль-

тант, менеджер і система центрального офісу. Варіантами використання є реєстрація продажу, пошук навчального посібника, замовлення навчального посібника, перегляд і запит даних про реалізацію книг.

Описи варіантів використання. На діаграмі варіантів використання немає місця для розміщення детального опису усіх варіантів використання, тому доводиться виносити описи за її межі. Для створення цих описів можна використовувати різні рівні формалізації, залежно від масштабів проекту і принципів, якими керуються розробники. У більшості випадків потрібен детальний опис усіх сценаріїв у варіанті використання. Простою реалізацією опису діаграми варіантів використання є один-два абзаци тексту. Іноді використовують таблицю, що складається з двох колонок: діяльність діючого суб'єкта і реакція на неї системи. Більш формалізований варіант може включати такі деталі, як передумову, умову поста, детальний опис послідовності кроків. Діаграма UML, що називається *діаграмою дій,* яка є не чим іншим як різновидом блок-схеми, іноді використовується якраз для того, щоби графічно зображати послідовність кроків у варіанті використання.

Діаграми варіантів використання і їх описи використовують, передусім, при початковому плануванні системи для забезпечення якнайкращого взаєморозуміння між замовниками і розробниками. Недаремно використовуються такі наочні значки у вигляді чоловічків, адже прості геометричні фігури – це етап розроблення програми "на серветці", тобто той момент, коли користувачі та розробники ще можуть спілкуватися один з одним за допомогою олівця і шматка паперу. Але варіанти використання, їх діаграми і описи корисні і під час розроблення програми. З ними можна звірятися для того, щоби упевнитися, що програмуються саме ті дії, які потрібні, понад це, вони можуть стати основою для тестування і написання документації.

Перехід від варіантів використання до класів. Коли визначено всі діючі суб'єкти і варіанти використання, то процес розроблення плавно переходить з етапу удосконалення у етап побудови програми. Це означає, що визначилася тенденція якогось зрушення у бік розробників від користувачів. З цього моменту їх тісна співпраця припиняється, оскільки вони починають розмовляти різними мовами. Першою проблемою, яку нам потрібно вирішити, є створення і удосконалення класів, які входитимуть у програму.

Одним з підходів до іменування класів є використання імен іменників, що трапляються в коротких описах варіантів використання. Ми хочемо, щоб об'єкти класів програми відповідали об'єктам реального світу, і ці іменники якраз є іменами тієї суті, яку вказав нам замовник. Вони є кандидатами в класи, але, мабуть, не з усіх іменників можуть вийти прийнятні класи. Потрібно виключити дуже загальні, тривіальні іменники, а також ті, які краще подати у вигляді атрибутів (простих змінних).

Після визначення декількох кандидатів у класи можна починати думати про те, як вони працюватимуть. Для цього варто подивитися на дієслова описів варіантів використання. Частенько буває так, що дієслово стає тим повідомленням, яке передається від одного об'єкта до іншого, або передує тим діям, які виникають між класами.

Діаграму класів UML (вона обговорювалася в попередніх розділах) можна використовувати для того, щоби показати класи і їх взаємини. Варіант використання реалізується послідовністю повідомлень, що посилають одні об'єкти іншим. Можна використовувати ще одну діаграму UML, що називається *діаграмою взаємодії,* для детальнішого представлення цих послідовностей. Насправді для кожного сценарію варіанта використання застосовують свою діаграму взаємодії. У подальших підрозділах розглядатимемо приклади *діаграм послідовностей*, що є різновидом діаграми взаємодій.

Процес розроблення програм краще описувати з використанням будь-якого конкретного прикладу, тому зараз ми звернемося до розгляду процесу розроблення реальної програми. Але Ви все-таки читаєте навчальний посібник, а це не компакт-диск, тому програму довелося зробити настільки стислою, що, власне кажучи, викликає сумнів доцільність формалізації процесу її розроблення. Та все ж навіть такий приклад повинен допомогти зрозуміти суть понять, які ми ввели вище.

14.3. Предметна область програмування

Програма, яку ми створюватимемо як приклад, називається landlord – домовласник. Ви можете поважати або не поважати свого домовласника, але необхідно цілком уявляти собі ті дані, з якими йому доводиться працювати: плата за житло і витрати. Ось такий нехитрий бізнес. Його ми і описуватимемо в нашій програмі.

Уявіть собі, що Ви є незалежним експертом з ведення домашнього господарства і з мови програмування С++, і до Вас зайшов замовник на ім'я Степан Полатайко. Полатайко – дрібний господар, у його власності перебуває будівля по вул. Левандівка, що складається з 12 кімнат, які він здає в оренду студентам. Він хоче, щоб Ви написали програму, яка спростила б його марудну працю з реєстрації даних і друкування звітів про свою фінансову діяльність. Якщо Ви зможете домовитися із Степаном Полатайком про ціну, терміни і загальне призначення програми, то можете вважати, що Ви задіяні до процесу розроблення ПЗ.

Рукописні форми. На цей момент Степан Полатайко записує всю інформацію про свій дім уручну в старомодний гросбух. Він показує Вам, які форми використовуються для ведення справ: перелік мешканців; доходи від оренди; витрати; річний звіт.

У *переліку мешканців* містяться номери кімнат й імена винаймачів, що проживають в них. Таким чином, це таблиця з двох стовпців і 12 рядків.

Доходи від оренди. Тут зберігаються записи про платежі винаймачів. У цій таблиці міститься 12 стовпців (за кількістю місяців) і по одному рядку на кожну орендовану кімнату. Щоразу, отримуючи гроші від мешканців, Степан Полатайко записує заплачену суму у відповідний елемент таблиці, яку показано на табл. 14.1. Така таблиця наочно показує, які саме суми і ким вони були внесені.

У таблиці *Поточні витрати* записано обов'язкові та поточні платежі. Вона нагадує чекову книжку і містить такі стовпці: дата, отримувач (компанія або користувач, на чиє ім'я виписується чек) і сума платежу. Окрім цього, є стовпець, в який Степан Полатайко вносить види або категорії платежів: Сплата за заставу,

ремонт, комунальні послуги, податки, страхування і т.ін. Результати поточних витрат показано в табл. 14.2.

Табл. 14.1. Місячний дохід від оренди приміщень, грн.

Номер кімнати	Січень	Лютий	Березень	Квітень	Травень	Червень	Липень	Серпень
101	696	695	695	695	695			
102	595	595	595	595	595			
103	810	810	825	825	825			
104	720	720	720	720	720			
201	680	680	680	680	680			
202	510	510	510	530	530			
203	790	790	790	790	790			
204	495	495	495	495	495			
301	585	585	585	585	585			
302	530	530	530	530	550			
303	810	810	810	810	810			
304	745	745	745	745	745			

Табл. 14.2. Поточні витрати

Дата	Отримувач	Сума	Категорія
1/3	Перший Megabank	5 187.30	Застава
1/8	Міська Вода	963.10	Зиск
1/9	Стійка Держава	4 840.00	Страхування
1/15	P.G. & E.	727.23	Зиск
1/22	Технічні Засоби Сема	54.81	Постачання
1/25	Ерні Glotz	150.00	Ремонти
2/3	Перший Megabank	5 187.30	Застава
2/7	Міська Вода	845.93	Зиск
2/15	P.G. & E.	754.20	Зиск
2/18	Plotx & Skeems	1 200.00	Юридичні грошові збори
3/2	Перший Megabank	5 187.30	Застава
3/7	Міська Вода	890.27	Зиск
3/10	Країна Springfield	9 427.00	Властивість Texes
3/14	P.G. & E.	778.38	Зиск
3/20	Кур'єр Gotham	26.40	Рекламування
3/25	Ерні Glotz	450.00	Ремонти
3/27	Живопис вищої Крапки	600.00	Обслуговування
4/3	Перший Megabank	5 187.30	Застава

У річному звіті (табл. 14.3) використовується інформація як з таблиці доходів, так і з таблиці витрат для підрахунку сум, що прийшли за рік від клієнтів і заплачених у процесі ведення бізнесу. Підсумовують всі прибутки від усіх мешканців за всі місяці. Також підсумовуються всі витрати і записують відповідно до бюджетних категорій. Нарешті, з доходів віднімаються витрати, внаслідок чого виходить значення чистого річного прибутку (або збитку).

Річний звіт Степан Полатайко складає тільки в кінці року, коли всі доходи і витрати грудня вже відомі. Наша комп'ютерна система виводитиме частковий річний звіт у будь-який час дня і ночі за період, що відбувся з початку року.

Табл. 14.3. Річний звіт

1	ПРИБУТОК	
2	Орендна плата	102 264.00
3	ПОВНИЙ ПРИБУТОК	102 264.00
4		
5	ВИТРАТИ	
6	Застава	62 247.60
7	Податки на нерухоме майно	9 427,00
8	Страхування	4 840.00
9	Зиск	18 326,76
10	Постачання	1 129,23
11	Ремонти	4 274.50
12	Обслуговування	2 609.42
13	Юридичні грошові збори	1 200,00
14	Створення штучного ландшафту	900.00
15	Рекламування	79,64
16		
17	ЗАГАЛЬНІ ВИТРАТИ	105 034,15
18		
19	ЧИСТИЙ ДОХІД АБО (ВТРАТА)	(2 700.15)

Степан Полатайко – особа поважного віку і достатньо консервативна, тому він, зрозуміло, просить зробити програму так, щоб зовнішній вигляд форм якомога точніше копіював таблички з його гросбухів. Власне кажучи, основне завдання програми можна визначити як введення даних і виведення різних звітів.

Прийняття допущень і спрощень. Звичайно, ми вже зробили декілька допущень і спрощень. Є ще велика кількість даних, пов'язаних з веденням справ зі здачі в оренду приміщень, таких, як застава за збиток, амортизація, іпотечна вигода, доходи від запізнілих внесків (з нарахуванням пені) і прокату пральних машин. Але під час розгляду цього прикладу ми не вдаватимемося в ці подробиці, тобто приймемо деякі допущення і спрощення.

Водночас, є ще декілька звітів, які хазяйновитий Степан Полатайко хотів би бачити у своїй програмі. Наприклад, звіт про вартість господарських витрат за деякий період чи чистий дохід. Звичайно, можна було б зробити таку програму, яка і перераховувала б суми по поточних рахунках, і могла б працювати як онлайновий магазин, але в цьому випадку з боку замовника було безрозсудно звернутися до послуг приватного програміста, не зовсім компетентного у виконанні таких послуг. Окрім цього, взагалі-то є і комерційні програми для домовласників, і бухгалтерські системи, які, при бажанні, можна було б придбати. Загалом, усякі претензії Степана Полатайка до неповноти нашої програми на цьому етапі ми спробуємо проігнорувати.

14.4. Програма landlord: етап удосконалення

Під час проходження етапу удосконалення повинні відбуватися зустрічі потенційних користувачів і реальних розробників для з'ясування того, що повинна

робити програма. У нашому прикладі Степан Полатайко є замовником і кінцевим користувачем системи, а ми з Вами – тим експертом, який буде і розробляти, і кодувати програму.

Встановлення діючих суб'єктів. Спочатку потрібно визначити, хто буде діючими суб'єктами? Хто вводитиме інформацію? Хто запрошуватиме? Чи буде хто-небудь ще взаємодіяти з програмою? Чи буде сама програма взаємодіяти з іншими?

У нашому прикладі з програмою landlord має працювати тільки один користувач – домовласник. Таким чином, один і той самий користувач вводить інформацію, здійснює розрахунки і переглядає їх у різних видах.

Навіть у такому невеликому проекті можна було б уявити собі ще декілька будь-яких дійових осіб. Це може бути, наприклад, податківець, а сама наша програма може стати діючим суб'єктом комп'ютерної системи податкової служби. Для спрощення пояснень ми не включатимемо усі ці та багато інших додаткових можливостей у наш проект.

З'ясування варіантів використання. Наступне, що потрібно з'ясувати, це групу дій, які ініціюватиме діючий суб'єкт. У реальному проекті це може стати достатньо об'ємним завданням, що вимагає тривалого з'ясування і ретельного уточнення деталей. У нашому ж випадку все це зробити не так складно, тобто можемо практично відразу скласти перелік варіантів використання, які можуть виникнути під час роботи нашої програми. Почнемо з того, що у нашому випадку домовласнику потрібно буде виконувати такі дії:

- почати роботу з програмою;
- додати нового мешканця в перелік;
- ввести значення орендної плати в таблицю доходів від оренди;
- ввести значення в таблицю витрат;
- вивести перелік мешканців;
- вивести таблицю доходів від оренди;
- вивести таблицю витрат;
- вивести річний звіт.

Таким чином, діаграму варіантів використання, яка отримується внаслідок наведеного переліку дій, можна подати так, як це показано на рис. 14.4.

Опис варіантів використання. Тепер непогано було б детально описати всі варіанти використання. Як було уже наголошено вище, описи можуть бути достатньо формалізованими і складними. Але наш приклад є дещо простим, тому все, що нам потрібно, – це невеликі описи у прозаїчній формі.

Почати програму. Ця дія, здавалося б, є дуже очевидною для того, щоби про неї зовсім не згадувати, але все ж таки... Коли запускається програма, на екран повинно виводитися меню, з якого користувач може вибрати потрібну дію. Це може називатися екраном інтерфейсу користувача.

Додати нового мешканця: сценарій 1. На екрані має відобразитися повідомлення, у якому програма просить користувача ввести ім'я мешканця і номер кімнати. Ця інформація повинна заноситися в таблицю. Перелік автоматично сортується за номерами кімнат.

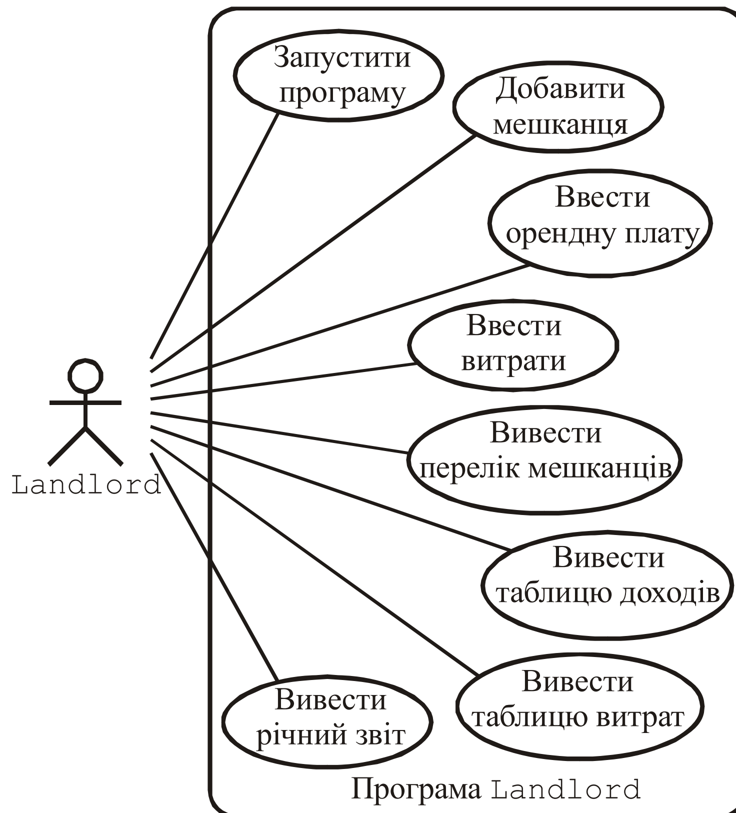

Рис. 14.4. Діаграма варіантів використання для програми landlord

Ввести орендну плату: сценарій 1. Екран введення орендної плати має містити повідомлення, з якого користувач зрозуміє, що йому необхідно ввести ім'я мешканця, місяць оплати, а також отриману суму грошей. Програма проглядає перелік мешканців і за номером кімнати знаходить відповідний запис в таблиці доходів від оренди. Якщо мешканець вперше вносить плату, в цій таблиці створюється новий рядок і вказана сума заноситься в стовпець того місяця, за який здійснюється оплата. Інакше значення вноситься в наявний рядок.

Ввести витрату. Екран введення витрати повинен містити запрошення користувачу на введення імені отримувача (або назви організації), суми оплати, дня і місяця, в який здійснюється оплата, бюджетної категорії. Потім програма створює новий рядок, що містить цю інформацію, і вставляє її в таблицю витрат.

Вивести перелік мешканців. Програма має виводити на екран перелік мешканців, кожен рядок переліку складається з двох полів: номера кімнати і імені мешканця.

Вивести таблицю доходів від оренди. Кожен рядок таблиці, яку виводить програма, складається з номера кімнати і значення щомісячної оплати.

Вивести таблицю витрат. Кожен рядок таблиці, яку має виводити програма, складається із значень місяця, дня, отримувача, суми і бюджетної категорії платежу.

Вивести річний звіт. Програма виводить річний звіт, що складається з:

- загальної орендної плати за минулий рік;
- переліку усіх витрат по кожній бюджетній категорії;
- підсумкового річного балансу (доходи/збитки).

Передбачення додаткових сценаріїв. Ми вже згадували про те, що варіант використання може складатися з декількох альтернативних сценаріїв. Вище було

описано тільки основний сценарій для кожного варіанта використання. Це сценарій безпомилкової роботи, коли все йде гладко, і мета операції досягається так само, як це вимагається в ідеалі. Проте необхідно передбачити більш загальні варіанти удосконалення подій у програмі. Як приклад можна навести випадок спроби запису користувачем у таблицю мешканців другого мешканця в зайняту кімнату.

Додати нового мешканця: сценарій 2. На екрані з'являється екран введення нового мешканця. Введений номер кімнати вже зайнятий деяким іншим мешканцем. Користувачу виводиться повідомлення про помилку.

А ось ще один приклад другого сценарію для варіанта використання. Тут користувач намагається ввести значення орендної плати для не наявного мешканця.

Ввести орендну плату: сценарій 2. Під час введення даних про орендну плату користувач повинен ввести ім'я мешканця, місяць оплати і її суму. Програма проглядає перелік мешканців, але не знаходить введене прізвище. Виводиться повідомлення про помилку.

З метою спрощення програми ми не розвиватимемо далі ці альтернативні сценарії, хоча в реальних проектах кожен додатковий сценарій повинен бути розроблений з тією ж ретельністю, що і основний. Тільки так можна домогтися того, щоби програма дійсно була застосовна в реальних умовах її експлуатації.

Використання діаграм дій UML. Діаграми дій UML використовують для моделювання варіантів використання. Цей тип діаграм демонструє керівні потоки від одних дій до інших. Він нагадує блок-схеми, які існували з найперших днів удосконалення технологій програмування. Але діаграми дій повністю формалізовані і мають додаткові можливості.

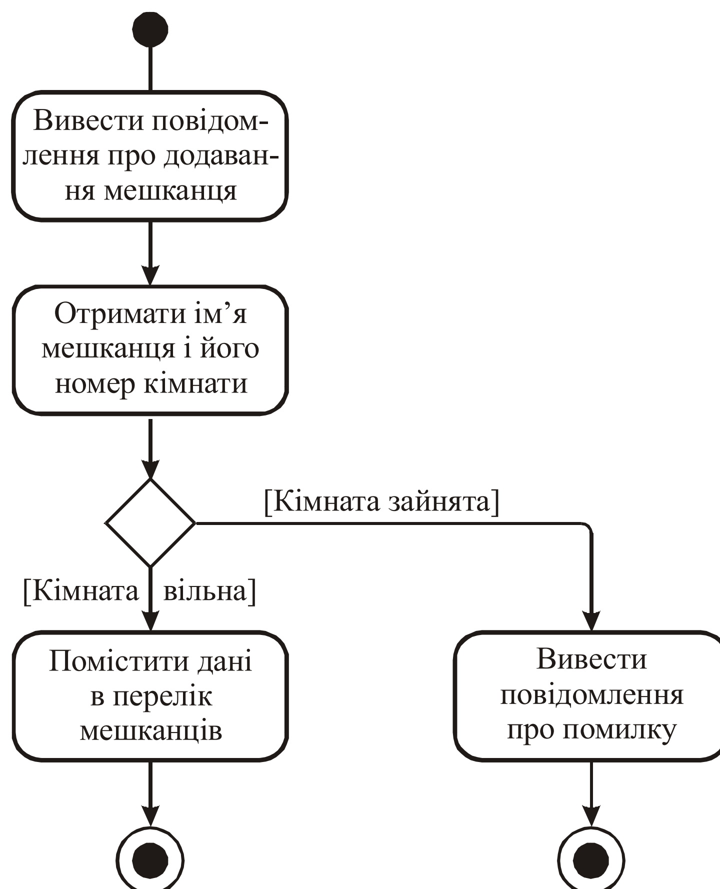

Рис. 14.5. Діаграма дій UML

Дії показують на діаграмах ромбоподібними контурами. Лінії, які з'єднують дії, є переходами від одних дій до інших. Розгалуження показане за допомогою ромбів з одним входом і двома або більш виходами. Як і на діаграмі станів, можна поставити елементи, призначені для вибору одного з рішень. Так само, як і там, можна задати початковий і кінцевий стани, що позначаються, відповідно, кружечком і кружечком у кільці.

На рис. 14.5 показано варіант використанням функції меню "Додати нового мешканця", що включає обидва сценарії. Вибір гілки діаграми залежить від того, зайнята чи ні введена користувачем кімната. Якщо вона вже зайнята, то виводиться повідомлення про помилку.

Діаграми дій можуть також використовуватися для представлення складних алгоритмів, що трапляються в коді програми. В цьому випадку вони практично ідентичні блок-схемам. Вони мають певні додаткові можливості, які ми тут не розглядаємо. Серед них, наприклад, представлення декількох конкуруючих дій.

14.5. Перехід від варіантів використання до класів

Етап побудови програмного забезпечення починається тоді, коли ми переходимо до планування структури програми. Насамперед спробуємо передбачити класи, з яких буде складатися наша програма. Для цього спочатку проаналізуємо перелік іменників з опису варіантів використання, потім внесемо деякі їх уточнення, визначимо їх атрибути, перейдемо від дієслів до повідомлень, і, на завершення, розробимо діаграми класів і послідовностей

Аналіз переліку іменників з опису варіантів використання. Розглянемо перелік усіх іменників, які візьмемо з опису варіантів використання:

1. Вікно інтерфейсу користувача.
2. Мешканець.
3. Вікно введення мешканців.
4. Ім'я мешканця.
5. Номер кімнати.
6. Рядок мешканця.
7. Перелік мешканців.
8. Орендна плата.
9. Вікно введення орендної плати.
10. Місяць.
11. Сума орендної плати.
12. Таблиця доходів від оренди.
13. Рядок орендної плати.
14. Витрата.
15. Вікно введення витрат.
16. Отримувач.
17. Обсяг платежу.
18. День.
19. Бюджетна категорія.
20. Рядок у таблиці витрат.
21. Таблиця витрат.
22. Річний звіт.
23. Загальна орендна плата.
24. Загальні витрати за категоріями.
25. Баланс.

Уточнення переліку іменників. З різних причин багато іменників не зможуть стати класами. Давайте проведемо відбір тільки тих іменників, які можуть претендувати на високе звання класів.

Ми виписали назви рядків різних таблиць: рядок мешканців, рядок орендної плати, рядок витрат. Іноді з рядків можуть виходити чудові класи, якщо вони складені або містять складні дані. Але кожен рядок таблиці мешканців містить дані тільки про одного мешканця, кожен рядок у таблиці витрат – тільки про один

платіж. Класи мешканців і витрат вже існують, тому ми наважимося припустити, що нам не потрібні два класи з однаковими даними, тобто ми не розглядатимемо рядки мешканців і витрат як претенденти на класи. Рядок орендної плати, з іншого боку, містить дані про номер кімнати і масив з 12 платежів за оренду по місяцях. Вона відсутня в таблиці доти, доки не буде зроблено перший внесок у поточному році. Подальші платежі вносяться у вже наявні рядки. Це ситуація складніша, ніж з мешканцями і витратами, тому зробимо рядок орендної плати класом. Тоді клас орендної плати як такий не міститиме нічого, окрім суми платежу, тому цей іменник перетворювати на клас ми не станемо.

Програма може породжувати значення в річному звіті з таблиці орендної плати і таблиці витрат, тому, напевно, не варто суму орендних платежів, а також загальні витрати за категоріями і баланс робити окремими класами. Вони є просто результатами обчислень.

Отже, складемо перелік класів, які було тільки що уточнено:

1. Вікно інтерфейсу користувача.
2. Мешканець.
3. Вікно введення мешканців.
4. Перелік мешканців.
5. Вікно введення орендної плати.
6. Таблиця доходів від оренди.
7. Рядок таблиці доходів від оренди.
8. Витрата.
9. Вікно введення витрат.
10. Таблиця витрат.
11. Річний звіт.

Визначення атрибутів. Багато іменників, яким відмовлено в реєстрації як кандидатами у класи, будуть потенційними кандидатами в атрибути (компонентні дані) класів. Наприклад, у класу "Мешканці" будуть такі атрибути: ім'я мешканця, номер кімнати. У класу "Витрати": отримувач, місяць, день, сума, бюджетна категорія. Більшість атрибутів можуть бути визначені так само.

Перехід від дієслів до повідомлень. Тепер подивимося, що нам дають варіанти використання для з'ясування того, якими повідомленнями обмінюватимуться класи. Оскільки повідомлення – це, по суті справи, є викликом методу в об'єкті, то визначення повідомлень зводиться до визначення методів класу, що приймає те або інше повідомлення. Як і у випадку з іменниками, далеко не кожне дієслово стає кандидатом у повідомлення. Деякі з них, замість прийняття даних від користувачів, зв'язуються з такими операціями, як виведення інформації на екран, або з якими-небудь ще діями.

Як приклад розглянемо опис варіанта використання функції "Вивести перелік мешканців". Курсивом виділено дієслова:

Програма виводить *на екран перелік мешканців, кожен рядок переліку* складається *з двох полів: номер кімнати й ім'я мешканця.*

Під словом "програма" ми насправді маємо на увазі вікно інтерфейсу користувача, отже, слово "виводить" означає, що об'єкт "Вікно інтерфейсу користувача" посилає повідомлення об'єкту "Перелік мешканців" (тобто викликає його метод). У повідомленні має міститься вказівка вивести самого себе на екран. Нескладно здогадатися, що метод може називатися, наприклад, Display().

Дієслово "складається" не належить ні до якого повідомлення. Він просто приблизно визначає склад рядка об'єкта "Перелік мешканців".

Розглянемо складніший приклад: варіант використання "Додати нового мешканця":

У вікні має відобразитися *повідомлення, у якому програма* просить *користувача* ввести *ім'я мешканця і номер кімнати. Ця інформація повинна* заноситися *в таблицю. Перелік автоматично* сортується *за номерами кімнат.*

Дієслово "складається" в цьому випадку означатиме таке. Екран інтерфейсу користувача повинен послати повідомлення класу "Екран введення мешканців", наказуючи йому вивести себе і отримати дані від користувача. Це повідомлення може бути викликом методу класу з іменем, наприклад getTenant().

Дієслова "просить" і "ввести" стосуються взаємодії класу "Екран введення мешканців" з користувачем. Вони не стають повідомленнями в об'єктному сенсі. Справді getTenant() виводить запрошення і записує відповіді користувача (ім'я мешканця і номер кімнати).

Дієслово "заноситися" означає, що об'єкт класу "Екран введення мешканців" посилає повідомлення об'єктові класу "Перелік мешканців". Можливо, як аргумент використовується новий об'єкт класу "Мешканці". Об'єкт "Перелік мешканців" може потім вставити цей новий об'єкт у свій перелік. Ця функція може мати ім'я типу insertTenant().

Дієслово "сортується" – це не повідомлення і взагалі не вид взаємодії об'єктів. Це просто опис переліку мешканців.

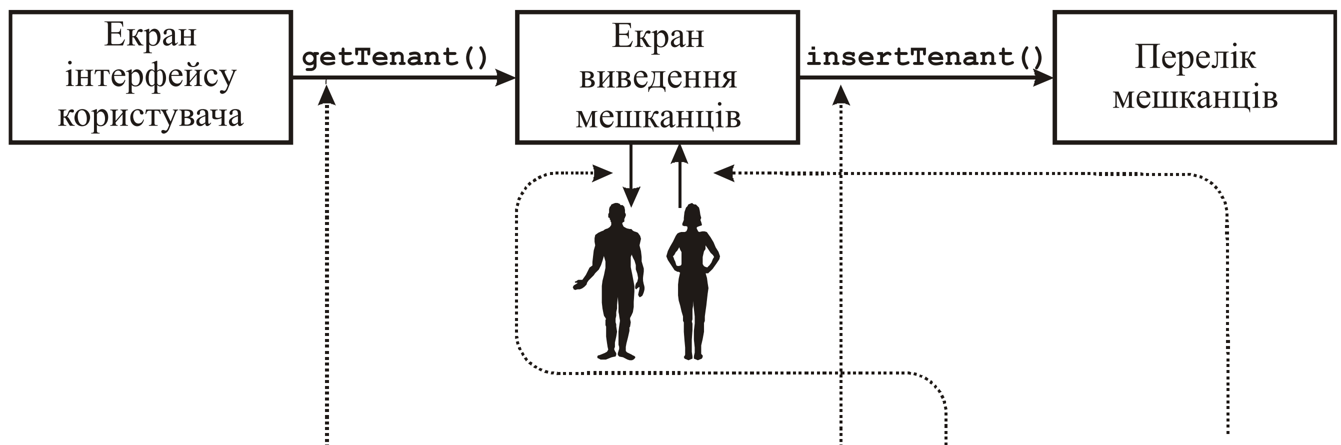

Рис. 14.6. Дієслова варіанта використання "Додати нового мешканця"

На рис. 14.6 показано варіант використання "Додати нового мешканця" і його зв'язку з описаними вище повідомленнями.

Коли ми почнемо писати код програми, то виявимо, що деякі дії залишилися поза нашим розглядом у цьому варіанті використання, але потрібні програмі. Наприклад, ніде не йдеться про створення об'єкта "Мешканець". Проте, напевно, зрозуміло, що "Перелік мешканців" містить об'єкти типу "Мешканець", а останні повинні бути створені до їх внесення в перелік. Отже, системний інженер вирішує, що метод getTenant() класу "Екран введення мешканців" – це відповідне місце для створення об'єкта "Мешканець", що вставляється в перелік мешканців.

Усі інші варіанти використання повинні бути проаналізовані аналогічним чином, щоб можна було створити основу для зв'язування класів. Зверніть увагу, що ми все ще використовуємо імена класів, що збігаються із словами або словосполученнями варіантів використання. Коли ми почнемо писати код програми, нам, зазвичай, доведеться перейменувати їх у що-небудь прийнятніше (імена повинні складатися з одного слова).

Побудова діаграм класів. Знаючи, які класи буде включено в розроблювану програму і як вони пов'язані між собою, ми зможемо побудувати діаграми класів. У попередніх розділах ми вже бачили приклади таких діаграм. На рис. 14.7 показано діаграму класів програми landlord.

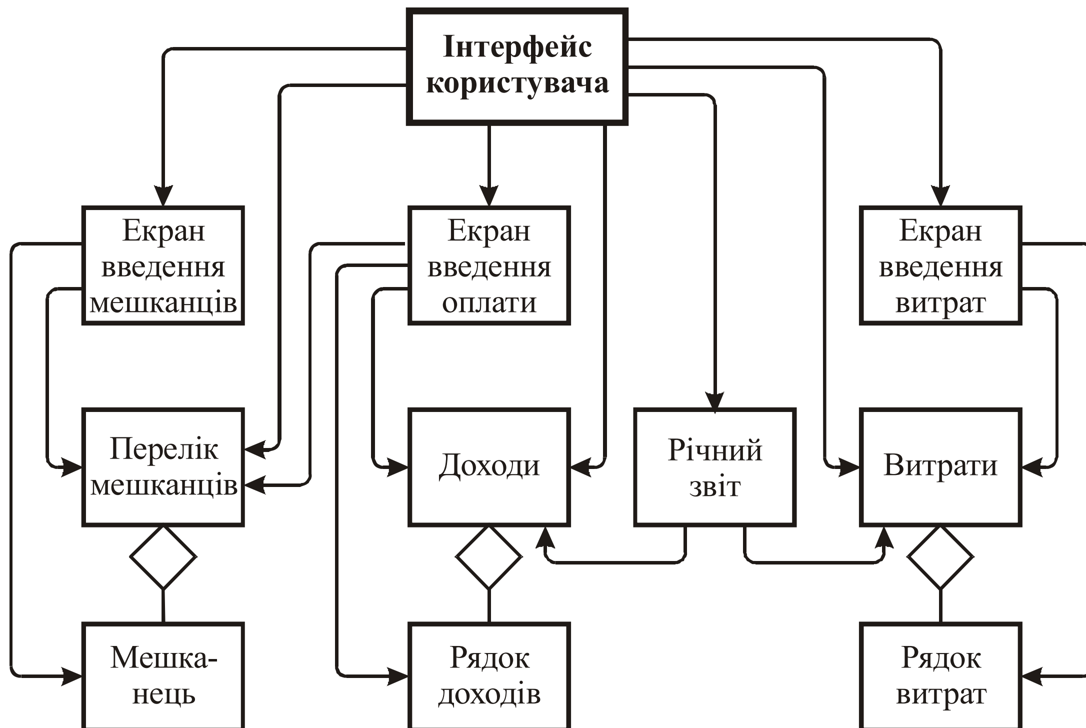

Рис. 14.7. Діаграма класів для програми landlord

Побудова діаграм послідовностей. Перш ніж почати писати код програми, було б логічно з'ясувати детальніше, як здійснюється кожен крок кожного варіанта використання. Для цього можна побудувати діаграми послідовностей UML. Це один з двох типів діаграм взаємодії UML (другий тип – сумісна діаграма). І на тій, і на іншій відображається, як події розгортаються в часі. Просто діаграма послідовностей наочніше зображає процес перебігу часу. На ній вертикальна вісь – це час. Вона "починається" вгорі і відбувається зверху вниз за діаграмами. Вгорі знаходяться імена об'єктів, які братимуть участь в цьому варіанті використання. Дія зазвичай починається з того, що об'єкт, розташований зліва, посилає повідомлення об'єкту, розташованому справа. Зазвичай чим правіше розташований об'єкт, тим нижче його значущість для програми або більше його залежність від інших.

Зверніть увагу на те, що на діаграмі показано не класи, а об'єкти. Кажучи про послідовності повідомлень, необхідно згадати, що повідомлення пересилаються саме між об'єктами, а не між класами. На діаграмах UML назви об'єктів відрізняються від назв класів наявністю підкреслення.

Лінією життя називається пунктирна лінія, що йде вниз від кожного об'єкта. Вона показує, коли об'єкт починає і закінчує своє існування. У тому місці, де об'єкт видаляється з програми, лінія життя закінчується.

Діаграма послідовностей для варіанта використання "Почати програму". Зверніть увагу на деякі з діаграм послідовностей нашої програми. Почнемо з найпростішої з них. На рис. 14.8 показано діаграму для варіанта використання "Почати програму".

Під час запуску програми визначається userInterface – клас підтримки екрану інтерфейсу користувача, який ми так довго обговорювали, кажучи про варіанти використання. Припустимо, програма створює єдиний об'єкт класу під назвою theUserInterface. Саме цей об'єкт породжує всі варіанти використання. Він з'являється зліва на діаграмі послідовностей. Як бачите, ми на цьому етапі вже перейшли до нормальних імен класів, прийнятих у процесі написання коду програми.

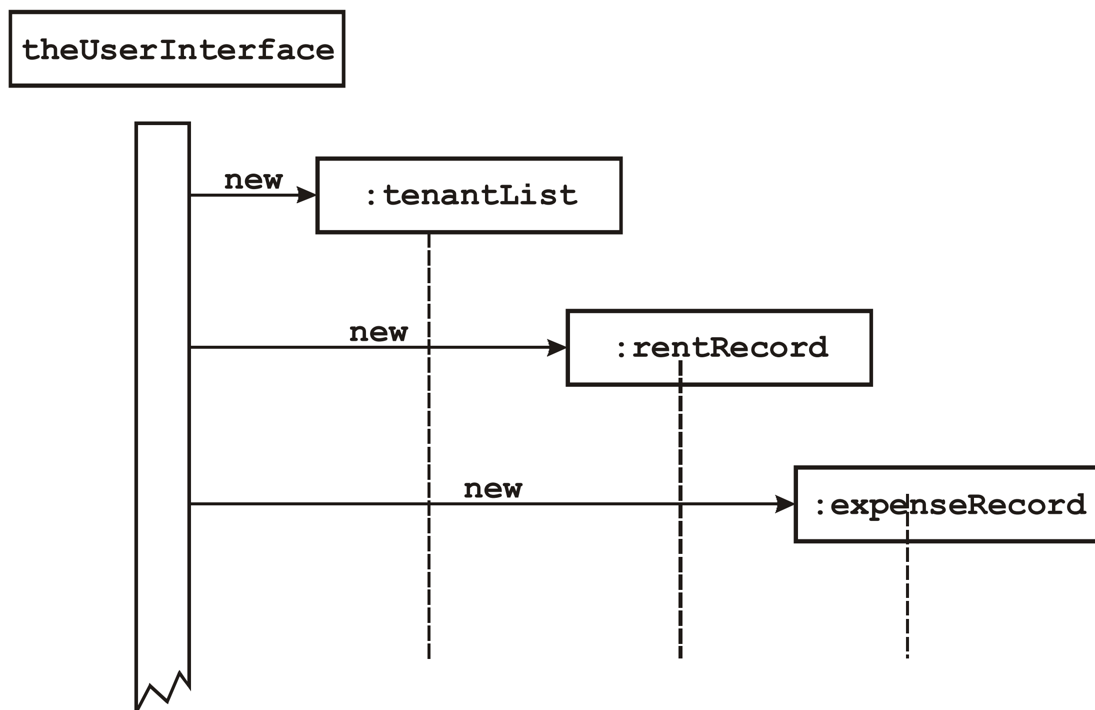

Рис. 14.8. Діаграма послідовностей для варіанта використання "Почати програму"

Коли вступає в роботу об'єкт theUserInterface, перше, що він має виконати, то це створити три основні структури даних у програмі. Це об'єкти класів tenantList, rentRecord і expenceRecord. Виходить, що вони народжуються безіменними, оскільки для їх створення використовується **new**. Імена мають тільки покажчики на них. Як же нам їх назвати? На щастя, як ми переконалися на прикладі об'єктних діаграм, UML надає декілька способів іменування об'єктів. Якщо справжнє ім'я невідоме, то можна використовувати замість нього двокрапку з іменем класу (:tenantList). На діаграмі підкреслення імені і двокрапка перед ним нагадує про те, що ми маємо справу з об'єктом, а не з класом.

Вертикальна позиція прямокутника з іменем об'єкта указує на той момент часу, коли об'єкт був розроблений (перший створює об'єкт класу tenantList). Всі об'єкти, які Ви бачите на діаграмі, продовжують існувати весь час, доки програма знаходиться на виконанні. Шкала часу, строго кажучи, рисується не в масштабі, вона призначена тільки для того, щоби показати взаємозв'язки між різними подіями.

Горизонтальні лінії є повідомленнями (тобто викликами методів). Суцільна стрілка свідчить про нормальний синхронний виклик функції, відкрита – про асинхронну подію.

Прямокутник, розташований під theUserInterface, називається *блоком активності* (або *центром керування*). Він показує, що розташований над ним об'єкт є активним. У звичайній процедурній програмі, такій, як наша (landlord), "активний" означає, що метод цього об'єкта або здійснюється сам, або викликав на виконання іншу функцію, яка ще не завершила свою роботу. Три інші об'єкти на цій діаграмі не є активними, тому що theUserInterface ще не послав їм активізаційних повідомлень.

Діаграма послідовностей для варіанта використання "Виведення переліку мешканців". Ще один приклад діаграми послідовностей представлено на рис. 14.9. На рисунку зображено роботу варіанта використання "Виведення переліку мешканців". Повернення значень функціями показано переривистими лініями. Зверніть увагу: об'єкти активні тільки тоді, коли викликано будь-який їх метод. Зверху над лінією повідомлення може бути вказано ім'я викликуваному методу.

Що ми бачимо? Об'єкт theUserInterface дає завдання об'єктові tenantList вивести себе на екран (викликом його методу Display()), а той, своєю чергою, виводить всі об'єкти класу tenant. Зірочка означає, що повідомлення посилатиметься циклічно, а фраза в квадратних дужках **[**для усіх об'єктів tenant**]** повідомляє умову повторення[1] об'єкта.

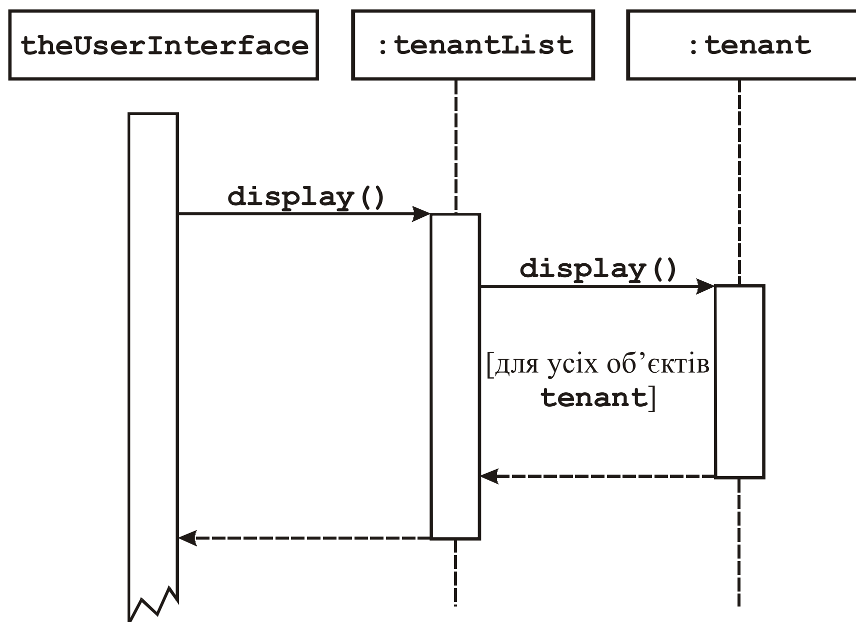

Рис. 14.9. Діаграма послідовностей для варіанта використання "Виведення переліку мешканців"

Діаграма послідовностей для варіанта використання "Додати нового мешканця". Нарешті, останнім прикладом діаграми послідовностей, який ми тут наводимо, є діаграма для варіанта використання "Додати нового мешканця". Її показано на рис. 14.10. Сюди ми включили самого домовласника у вигляді об'єкта, який визначає різні дії. У нього є свій власний блок активності. За допомогою

[1] У програмі, насправді, ми використовуватимемо **cout** замість функції Display().

цього об'єкта можна повністю змоделювати процес взаємодії користувача з програмою.

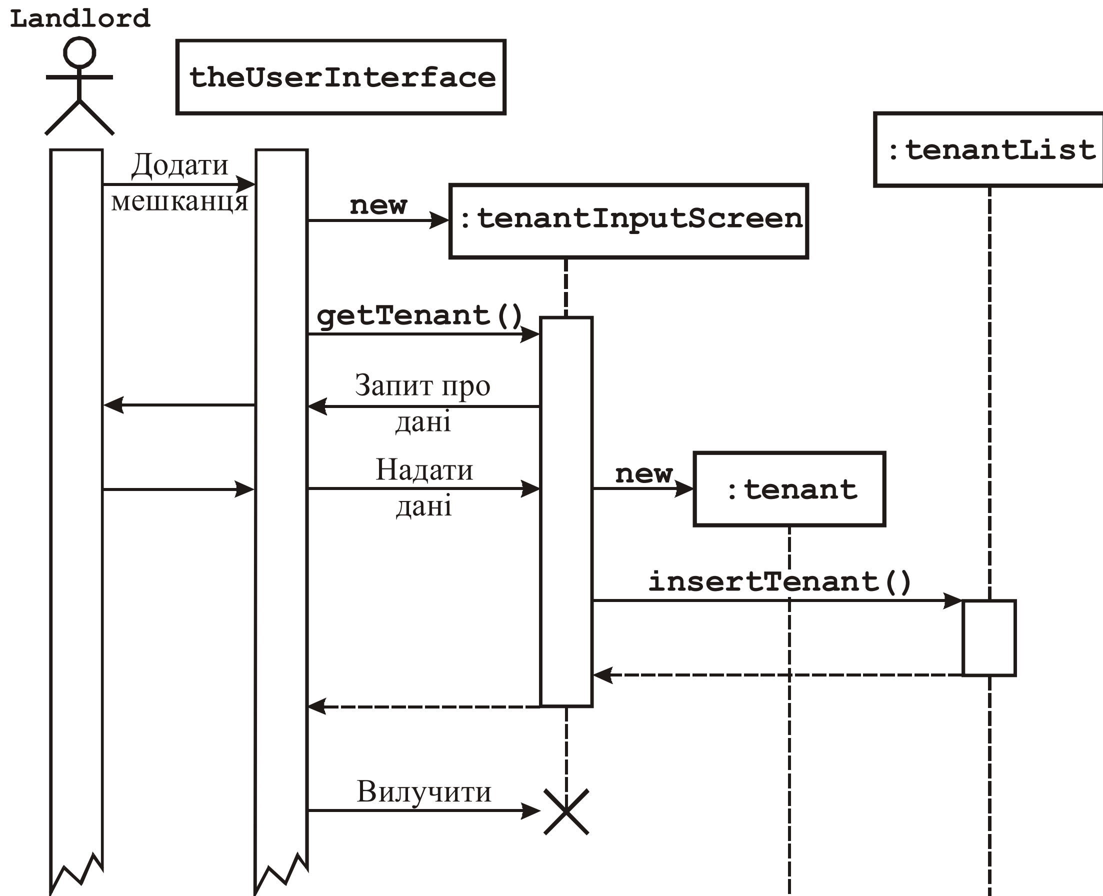

Рис. 14.10. Діаграма послідовностей для варіанта використання "Додати нового мешканця"

Користувач повідомляє програму про те, що він бажає додати нового мешканця. Об'єкт theUserInterface створює новий об'єкт класу tenantInputScreen. У цьому об'єкті є методи, що дають змогу отримати від користувача дані про мешканця, створити новий об'єкт типу tenant і викликати метод об'єкта класу tenantList для додавання в перелік знов створеного мешканця. Коли все це виконано, то об'єкт theUserInterface віддаляється. Велика буква "X" в кінці лінії життя tenantInputScreen свідчить про те, що об'єкт видалено.

Діаграми послідовностей, приклади яких ми тут навели, мають справу тільки з головними сценаріями кожного варіанта використання. Існують, насправді ж, способи показати на діаграмах і декілька сценаріїв, але можна і для кожного сценарію створити свою діаграму.

Обмеження на кількість сторінок у навчальному посібнику не дає змоги нам навести всі приклади діаграм послідовностей, але, здається, і так вже показано достатньо для їх загального розуміння.

14.6. Кроки написання коду програми

Маючи діаграми варіантів використання, детальні їх описи, діаграми класів і послідовностей, а також попередні плани щодо створення коду програми, можна запустити компілятор і почати писати сам код програми. Це друга частина етапу її

побудови, яка складається з декількох кроків: написання заголовного файлу; створення початкових *.cpp файлів; введення вимушеного спрощення коду програми; передбачити взаємодію користувача з програмою.

Варіанти використання, визначені на етапі удосконалення, стають ітераціями на новому етапі (див. рис. 14.2). У великому проекті кожна ітерація може проводитися різними групами програмістів. Всі ітерації повинні проектуватися окремо, а їх результати – надаватися замовнику для внесення доповнень і виправлень. У нашій невеликій програмі, втім, ці ускладнення зайві.

Написання заголовного файлу. Врешті-решт ми дійшли до написання таких звичних файлів з кодами програми. Найкраще починати із заголовного (*.h) файлу, у якому необхідно визначити тільки інтерфейсні частини класів, але не подробиці їх реалізації. Як зазначалося раніше, оголошення в заголовному файлі – це загальнодоступна частина класів. Тіла функцій, які є розташовані в *.cpp-файлах, називаються реалізацією і користувачам недоступні.

Написання заголовного файлу – це проміжний крок між етапами плануванням і звичайним кодуванням методів. Розглянемо вміст заголовного файлу landlord.h.

Код програми 14.1. Заголовний файл до програми landlord

```
// Заголовний файл landlord.cpp – містить оголошення класів і т.ін.
#pragma warning(disable:4786) // Для множин (тільки компілятори Microsoft)
#include <iostream>            // Для потокового введення-виведення
#include <vector>              // Для роботи з контейнерним класом "Вектор"
#include <set>
#include <string>              // Для роботи з рядковими типами
#include <algorithm>           // Для sort()
#include <numeric>             // Для accumulate()
using namespace std;           // Використання стандартного простору імен

/// Глобальні методи ///
void getaLine(string& inStr);  // Отримання рядка тексту
char getaChar();               // Отримання символу

class tenant {          // Мешканці
private:
	string name;        // Ім'я мешканця
	int aptNumber;      // Номер кімнати мешканця
			// Тут може бути будь-яка інша інформація про мешканця,
			// наприклад, номер телефону і т.ін.
public:
	tenant(string n, int aNo);
	~tenant();
	int getAptNumber(); // Потрібно для використання в множині
	friend bool operator < (const tenant&, const tenant&);
	friend bool operator == (const tenant&, const tenant&);
	// Для операцій введення/виведення
	friend ostream& operator << (ostream&, const tenant&);
}; // <------> Кінець оголошення класу tenant
// Функціональний об'єкт для порівняння імен мешканців
class compareTenants {
```

```
public:
        bool operator () (tenant*, tenant*) const;
};

// >------> Клас tenantList
class tenantList {
private:
        // Встановити покажчики на мешканців
        set<tenant*, compareTenants> setPtrsTens;
        set<tenant*, compareTenants>::iterator iter;
public:
        ~tenantList();      // Оголошення деструктора (Видалення мешканців)
        void insertTenant(tenant*);        // Внесення мешканця в перелік
        int getAptNo(string);              // Повертає номер кімнати
        void Display();                    // Виведення переліку мешканців
};
// <------> Кінець оголошення класу tenantList

class tenantInputScreen {
private:
        tenantList* ptrTenantList;
        string tName;
        int aptNo;
public:
        tenantInputScreen(tenantList* ptrTL): ptrTenantList(ptrTL)
        { /* тут порожньо */ }
        void getTenant();
}; // <------> Кінець класу tenantInputScreen

// Один рядок таблиці прибутку: адреса і 12 місячних плат
class rentRow {
private:
        int aptNo;
        float rent[12];
public:
        rentRow(int);       // Оголошення конструктора з одним параметром
        void setRent(int, float);  // Запис плати за місяць
        float getSumOfRow();       // Сума платежів з одного рядка
                                   // Потрібно для збереження в множині
        friend bool operator < (const rentRow&, const rentRow&);
        friend bool operator == (const rentRow&, const rentRow&);
        friend ostream& operator << (ostream&, const rentRow&); // Для виведення
}; // <------> Кінець класу rentRow

// Функціональний об'єкт порівняння об'єктів rentRows
class compareRows {
public:
        bool operator () (rentRow*, rentRow*) const;
};
```

```
class rentRecord {
private:
	// Множину покажчиків на об'єкти rentRow (по одному на мешканця)
	set<rentRow*, compareRows> setPtrsRR;
	set<rentRow*, compareRows>::iterator iter;
public:
	~rentRecord();
	void insertRent(int, int, float);
	void Display();
	float getSumOfRents(); // Сума усіх платежів
}; // <------> Кінець класу rentRecord

class rentInputScreen {
private:
	tenantList* ptrTenantList;
	rentRecord* ptrRentRecord;
	string renterName;
	float rentPaid;
	int month;
	int aptNo;
public:
	rentInputScreen(tenantList* ptrTL rentRecord* ptrRR):
	ptrTenantList(ptrTL), ptrRentRecord(ptrRR)
	{ /*тут пусто*/ }
	void getRent(); // Орендна плата одного мешканця за один місяць
}; // <------> Кінець класу rentInputScreen

class expense {
public:
	int month, day;
	string category, payee;
	float amount;
	expense() { }
	expense(int m, int d, string c, string p, float a):
	month(m), day(d), category(c), payee(p), amount(a)
	{ /* тут порожньо! */ }

	// Потрібно для зберігання в множині
	friend bool operator < (const expense&, const expense&);
	friend bool operator == (const expense&, const expense&);
	// Потрібно для виведення
	friend ostream& operator << (ostream&, const expense&);
}; // <------> Кінець класу expense
// Функціональний об'єкт порівняння витрат
class compareDates {
public:
	bool operator () (expense*, expense*) const;
};
```

```
// Функціональний об'єкт порівняння витрат
class compareCategories {
public:
	bool operator () (expense*, expense*) const;
};

class expenseRecord {
private:
	// Вектор покажчиків на витрати
	vector<expense*> vectPtrsExpenses;
	vector<expense*>::iterator iter;
public:
	~expenseRecord();
	void insertExp(expense*);
	void Display();
	float displaySummary(); // Потрібно для річного звіту
}; // <------> Кінець класу expenseRecord

class expenseInputScreen {
private:
	expenseRecord* ptrExpenseRecord;
public:
	expenseInputScreen(expenseRecord*);
	void getExpense();
}; // <------> Кінець класу expenseInputScreen

class annualReport {
private:
	rentRecord* ptrRR;
	expenseRecord* ptrER;
	float expenses, rents;
public:
	annualReport(rentRecord*, expenseRecord*);
	void Display();
}; // <------> Кінець класу annualReport

class userInterface {
private:
	tenantList* ptrTenantList;
	tenantInputScreen* ptrTenantInputScreen;
	rentRecord* ptrRentRecord;
	rentInputScreen* ptrRentInputScreen;
	expenseRecord* ptrExpenseRecord;
	expenseInputScreen* ptrExpenseInputScreen;
	annualReport* ptrAnnualReport;
	char ch;
public:
	userInterface();
	~userInterface();
```

```
        void interact();
}; // <------> Кінець класу userInterfac

// <------> Кінець файлу landlord.h
```

Оголошення класів – це просто. Більшість оголошень зростають безпосередньо з класів, створених за допомогою узятих з описів варіантів використання іменників, і відображаються на діаграмі класів. Тільки імена з багатослівних потрібно зробити однослівними. Наприклад, ім'я "Перелік мешканців" (Tenant List) перетворюється в TenantList.

У заголовному файлі було додано ще декілька допоміжних класів. Згодом виявиться, що ми зберігаємо покажчики на об'єкти в різних типах контейнерів STL. Це означає, що ми повинні порівнювати об'єкти цих контейнерів так, як це описано в розд. 12 "Введення в стандартну бібліотеку шаблонів". Об'єктами порівняння насправді є класи compareTenants, compareRows, compareDates і compareCategories.

Описи атрибутів. Як уже було зазначено вище, багато атрибутів (методи) для кожного з класів є похідними з тих іменників, які самі не стали класами. Наприклад, name і aptNumber стали атрибутами класу tenant.

Інші атрибути можуть бути виведені з асоціацій в діаграмі класів. Асоціації можуть визначати ті атрибути, які є покажчиками або посиланнями на інші класи. Це пояснюється неможливістю асоціювати щось з чимось, що знаходиться невідомо де. Таким чином, у класі rentInputScreen з'являються такі атрибути, як ptrTenantList і ptrRentRecord.

Складені значення (агрегати). Агрегатні зв'язки показані в трьох місцях на діаграмі класів. Зазвичай агрегати виявляють ті контейнери, які є атрибутами агрегатного класу (тобто "цілого" класу, що містить "частини").

Ні за описами варіантів використання, ні за діаграмами класів неможливо вгадати, якого роду контейнери повинні використовуватися для цих агрегатів. Вам як програмістам доведеться самим щоразу вибирати відповідний контейнер для кожного складеного значення – будь-то простий масив, контейнер STL або що-небудь ще. У програмі landlord ми зробили такий вибір:

- клас tenantList містить STL-множину покажчиків на об'єкти класу tenant;
- клас rentRecord містить множину покажчиків на об'єкти класу rentRow;
- клас expenseRecord містить вектор покажчиків на об'єкти класу expense.

Для tenantList і rentRecord ми вибрали множини, оскільки основним параметром є швидкий доступ до даних. Для expenseRecord вибрано вектор, тому що нам важливо здійснювати швидке сортування і за датою, і за категоріями, а вектори дають змогу сортувати дані найефективніше.

У всіх агрегатах ми вважали за краще зберігати покажчики замість самих об'єктів, щоб уникнути зайвого копіювання даних у пам'яті. Зберігання самих об'єктів необхідно застосовувати в тих випадках, коли об'єктів мало і вони невеликі. Зазвичай, великої різниці в швидкості на прикладі якихось 12 екземплярів об'єкта ми не побачимо, але у принципі про ефективність методу зберігання даних необхідно замислюватися завжди.

Створення початкових *.cpp файлів. У початкових файлах містяться тіла методів, які були оголошені в заголовному файлі. Написання коду програми цих

методів має починатися тільки на цьому етапі розроблення ПЗ і ні кроком раніше, тому що тільки зараз ми знаємо ім'я кожної функції, її призначення і навіть, можливо, можемо передбачити аргументи, що передаються їй.

Нарешті ми, так би мовити, відокремили зерна від полови: **main**() зберігаємо в одному коротенькому файлі lordApp.cpp, а визначення функцій, оголошених у заголовному файлі, – в іншому. У секції **main**() створюється об'єкт userInterface і викликається метод interact(). Наведемо файл, у якому зберігається **main**().

Код програми 14.2. Демонстрація програми lordApp.cpp

```
// Файл, що постачається клієнту.
#include "landlord.h"

int main()
{
	userInterface theUserInterface;
	theUserInterface.interact();
	getch(); return 0;
}
// <------> Кінець файлу lordApp.cpp
```

На завершення розглянемо файл landlord.cpp, у якому містяться всі визначення методів.

Код програми 14.3. Демонстрація програми landlord.cpp

```
void getaLine(string& inStr) // Отримання рядка тексту
{
	char temp[21];
	cin.get(temp, 20, '\n');
	cin.ignore(20, '\n');
	inStr = temp;
} // <------> Кінець getaLine()

char getaChar() // Отримання символу
{
	char ch = cin.get();
	cin.ignore(80, '\n');
	return ch;
} // <------> Кінець getaChar()

// >------> Методи класу tenant

tenant::tenant(string n, int aNo) : name(n), aptNumber(aNo)
	{ /* тут порожньо */ }
//---------------------------------------------------------
tenant::~tenant()
	{ /* тут також порожньо */ }
//---------------------------------------------------------

int tenant::getAptNumber()
	{ return aptNumber; }
//---------------------------------------------------------
```

```
bool operator < (const tenant& t1, const tenant& t2)
        { return t1.name < t2.name; }
//--------------------------------------------------------

bool operator == (const tenant& t1, const tenant& t2)
        { return t1.name == t2.name; }
//--------------------------------------------------------

ostream& operator << (ostream& s, const tenant& t)
        { s << t.aptNumber << '\t' << t.name << endl; return s; }
//--------------------------------------------------------

// >------> Методи класу tenantInputScreen

void tenantInputScreen::getTenant()      // Отримання даних про мешканцях
{
        cout << "Введіть ім'я мешканця (Тарас Редько): ";
        getaLine(tName);
        cout << "Введіть номер кімнати (101): ";
        cin >> aptNo;
        cin.ignore(80, '\n');                    // Створити мешканця
        tenant* ptrTenant = new tenant(tName, aptNo);
        ptrTenantList->insertTenant(ptrTenant); // Занести перелік мешканців
}

bool compareTenants::operator () (tenant* ptrT1, tenant* ptrT2) const
        { return *ptrT1 < *ptrT2; }
//--------------------------------------------------------

// >------> Методи класу tenantList

tenantList::~tenantList()               // Оголошення деструктора
{
        while( !setPtrsTens.empty() ) { // Видалення усіх мешканців
                                        // Видалення покажчиків з множини
                iter = setPtrsTens.begin();
                delete *iter;
                setPtrsTens.erase(iter);
        }
} // <------> Кінець ~tenantList()

void tenantList::insertTenant(tenant* ptrT)
{
        setPtrsTens.insert(ptrT);           // Вставка
}
//--------------------------------------------------------

int tenantList::getAptNo(string tName) // Ім'я присутнє у переліку?
{
        int aptNo;
```

```
        tenant dummy(tName, 0);

        iter = setPtrsTens.begin();
        while( iter != setPtrsTens.end() ) {
                aptNo = (*iter)->getAptNumber();   // Пошук мешканця
                if(dummy == **iter++)                           // В переліку?
                return aptNo;                                   // Так
        }
        return -1;                                              // Немає
} // <------> Кінець getAptNo()

void tenantList::Display() // Виведення переліку мешканців
{
        cout << endl << "Apt#\tIм'я мешканця\n-------------------" << endl;
        if( setPtrsTens.empty() )
                cout << "***Нема мешканців***" << endl;
        else {
                iter = setPtrsTens.begin();
                while( iter != setPtrsTens.end() ) cout << **iter++;
        }
} // <------> Кінець Display()

// >------> Методи класу rentRow

rentRow::rentRow(int an) : aptNo(an) // Однопараметризований конструктор
        { fill( &rent[0] &rent[12], 0); }
//--------------------------------------------------------

void rentRow::setRent(int m, float am)
        { rent[m]= am; }
//--------------------------------------------------------

float rentRow::getSumOfRow() // Сума орендних платежів у рядку
        { return accumulate( &rent[0] &rent[12], 0); }
//--------------------------------------------------------

bool operator < (const rentRow& t1, const rentRow& t2)
        { return t1.aptNo < t2.aptNo; }
//--------------------------------------------------------

bool operator == (const rentRow& t1, const rentRow& t2)
        { return t1.aptNo == t2.aptNo; }
//--------------------------------------------------------
ostream& operator << (ostream& s, const rentRow& an)
{
        s << an.aptNo << '\t';          // Вивести номер кімнати
        for(int j=0; j<12; j++) {       // Вивести 12 орендних платежів
                if(an.rent[j] == 0) s << " 0 ";
                else s << an.rent[j] << " ";
        }
```

```
        s << endl;
        return s;
} // <------> Кінець operator <<

bool compareRows::operator () (rentRow* ptrR1, rentRow* ptrR2) const
        { return *ptrR1 < *ptrR2; }

// >------> Методи класу rentRecord

rentRecord::~rentRecord()          // Оголошення деструктора
{
        while( !setPtrsRR.empty() ) {     // Видалити рядки з платежами
                                          // Видалити покажчики з множини
                iter = setPtrsRR.begin();
                delete *iter;
                setPtrsRR.erase(iter);
        }
} // <------> Кінець ~rentRecord()

void rentRecord::insertRent(int aptNo, int month, float amount)
{
        rentRow searchRow(aptNo);         // Тимчасовий рядок з тим самим aptNo
        iter = setPtrsRR.begin(); // Пошук setPtrsRR
        while( iter != setPtrsRR.end() ) {
                if(searchRow==**iter) {   // rentRow знайдений?
                        // так, заносимо
                        (*iter)->setRent(month, amount); // Рядок у перелік
                        return;
                }
                else iter++;
        }                                               // Не знайдений
        rentRow* ptrRow = new rentRow(aptNo);           // Новий рядок
        ptrRow->setRent(month, amount);                 // Занести в неї платіж
        setPtrsRR.insert(ptrRow);                       // Занести рядок вектор
} // <------> Кінець insertRent()

void rentRecord::Display()
{
        cout << endl << "AptNo\tСіч Лют Бер Квіт Трав Черв "
                << "Лип Серп Вер Жовт Лист Груд\n"
                << "---------------------------------"
                << "---------------------------------" << endl;
        if( setPtrsRR.empty() )
                cout << "***Нема платежів!***" << endl;
        else {
                iter = setPtrsRR.begin();
                while( iter != setPtrsRR.end() )
                cout << **iter++;
        }
} // <------> Кінець Display()
```

```
float rentRecord::getSumOfRents()        // Сума усіх платежів
{
        float sumRents = 0.0;

        iter = setPtrsRR.begin();
        while( iter != setPtrsRR.end() ) {
                sumRents += (*iter)->getSumOfRow();
                iter++;
        }
        return sumRents;
} // <------> Кінець getSumOfRents()

// >------> Методи класу rentInputScreen

void rentInputScreen::getRent()
{
        cout << "Введіть ім'я мешканця: ";
        getaLine(renterName);
        aptNo = ptrTenantList->getAptNo(renterName);
        if(aptNo > 0) {      // Якщо ім'я знайдене
                                    // Отримати суму платежу
                cout << "Введіть суму платежу (345.67): ";
                cin >> rentPaid;
                cin.ignore(80, '\n');
                cout << "Введіть номер місяця оплати (1-12): ";
                cin >> month;
                cin.ignore(80, '\n');
                month--; // (внутрішня нумерація 0-11)
                ptrRentRecord->insertRent(aptNo, month, rentPaid);
        }
        else                        // Повернення
                cout << "Такого мешканця немає" << endl;
} // <------> Кінець getRent()

// Методи класу expense

bool operator < (const expense& e1, const expense& e2)
{                                              // Порівнює дати
        if(e1.month == e2.month)               // Якщо той же місяць
                return e1.day < e2.day;        // Порівняти дні
        else                                          // Інакше
                return e1.month < e2.month;    // Порівняти місяці
}
//--------------------------------------------------------
bool operator == (const expense& e1, const expense& e2)
        { return e1.month == e2.month && e1.day == e2.day; }
//--------------------------------------------------------

ostream& operator << (ostream& s, const expense& exp)
{
        s << exp.month << '/' << exp.day << '\t' << exp.payee << '\t' ;
```

```
        s << exp.amount << '\t' << exp.category << endl;
        return s;
}
//------------------------------------------------------------

bool compareDates::operator () (expense* ptrE1, expense* ptrE2) const
        { return *ptrE1 < *ptrE2; }
//------------------------------------------------------------

bool compareCategories::operator ()(expense* ptrE1, expense* ptrE2) const
        { return ptrE1->category < ptrE2->category; }
//------------------------------------------------------------

// >------> Методи класу expenseRecord

expenseRecord::~expenseRecord()                 // Оголошення деструктора
{
        while( !vectPtrsExpenses.empty() ) { // Видалити об'єкти expense
                                                    // Видалити покажчики на вектор
                iter = vectPtrsExpenses.begin();
                delete *iter;
                vectPtrsExpenses.erase(iter);
        }
} // <------> Кінець ~expenseRecord()

void expenseRecord::insertExp(expense* ptrExp)
        { vectPtrsExpenses.push_back(ptrExp); }
//------------------------------------------------------------

void expenseRecord::Display()
{
        cout << endl << "Дата\tОтримувач\t\tСума\tКатегорія\n"
                << "----------------------------------------" << endl;
        if( vectPtrsExpenses.size() == 0 )
                cout << "*** Витрат немає ***" << endl;
        else {
                sort( vectPtrsExpenses.begin(), // Сортування за датою
                vectPtrsExpenses.end(), compareDates() );
                iter = vectPtrsExpenses.begin();
                while( iter != vectPtrsExpenses.end() ) cout << **iter++;
        }
} // <------> Кінець Display()

float expenseRecord::displaySummary() // Використовується під час складання річного звіту
{
        float totalExpenses = 0; // Сума, всі категорії
        if( vectPtrsExpenses.size() == 0 ) {
                cout << "\tВсі категорії\t0" << endl;
                return;
        }
```

```
        // Сортувати за категоріями
        sort( vectPtrsExpenses.begin(),
        vectPtrsExpenses.end(), compareCategories() );

        // По кожній категорії сума записів
        iter = vectPtrsExpenses.begin();
        string tempCat = (*iter)->category;
        float sumCat = 0.0;
        while( iter != vectPtrsExpenses.end() ) {
                if(tempCat == (*iter)->category)
                        sumCat += (*iter)->amount;    // Та ж категорія
                else {                                                    // Інша
                        cout << '\t' << tempCat << '\t' << sumCat << endl;
                        totalExpenses += sumCat;     // Додати попередню категорію
                        tempCat = (*iter)->category;
                        sumCat = (*iter)->amount;     // Додати остаточне значення суми
                }
                iter++;
        } // <------> Кінець while
        totalExpenses += sumCat; // Додати суму кінцевої категорії
        cout << '\t' << tempCat << '\t' << sumCat << endl;
        return totalExpenses;
} // <------> Кінець displaySummary()

// >------> Методи класу expenseInputScreen

expenseInputScreen::expenseInputScreen(expenseRecord* per) :
        ptrExpenseRecord(per)
        { /*пусто*/ }
//---------------------------------------------------------

void expenseInputScreen::getExpense()
{
        int month, day;
        string category, payee;
        float amount;

        cout << "Введіть місяць (1-12): "; cin >> month; cin.ignore(80, '\n');

        cout << "Введіть день (1-31): "; cin >> day; cin.ignore(80, '\n');

        cout << "Введіть категорію витрат (Ремонт, Податки): ";
        getaLine(category);
        cout << "Введіть отримувача " << "(ЛьвівЕлектроЗбут): ";
        getaLine(payee);

        cout << "Введіть суму (39.95): "; cin >> amount; cin.ignore(80, '\n');

        expense* ptrExpense = new
        expense(month, day, category, payee, amount);
```

```
        ptrExpenseRecord->insertExp(ptrExpense);
} // <------> Кінець getExpense()

// >------> Методи класу annualReport

annualReport::annualReport(rentRecord* pRR,
        expenseRecord* pER) : ptrRR(pRR), ptrER(pER)
        { /* порожньо */ }
//--------------------------------------------------------

void annualReport::Display()
{
        cout << "Річний звіт\n--------------" << endl;
        cout << "Доходи" << endl;
        cout << "\tОрендна плата\t\t";
        rents = ptrRR->getSumOfRents();
        cout << rents << endl;

        cout << "Витрати" << endl;
        expenses = ptrER->displaySummary();
        cout << endl << "Баланс\t\t\t" << rents - expenses << endl;
} // <------> Кінець Display()

// >------> Методи класу userInterface

userInterface::userInterface()
{           // Це життєво важливо для програми
        ptrTenantList = new tenantList;
        ptrRentRecord = new rentRecord;
        ptrExpenseRecord = new expenseRecord;
} // <------> Кінець userInterface()

userInterface::~userInterface() // Оголошення деструктора
{
        delete ptrTenantList;
        delete ptrRentRecord;
        delete ptrExpenseRecord;
} // <------> Кінець ~userInterface()

void userInterface::interact()
{
        while(true) {
        cout << "Для введення даних натисніть 'i'\n"
                << " 'd' для виведення звіту\n"
                << " 'q' для виходу: ";
        ch = getaChar();
        if(ch=='i') { // Введення даних
                cout << " 't' для додавання мешканця\n"
                        << " 'r' для запису орендної плати\n"
                        << " 'e' для запису витрат: ";
```

```
            ch = getaChar();
            switch(ch) {
            // Екрани введення існують тільки під час їх використання
                case 't': ptrTenantInputScreen =
                        new tenantInputScreen(ptrTenantList);
                        ptrTenantInputScreen->getTenant();
                        delete ptrTenantInputScreen;
                        break;
                case 'r': ptrRentInputScreen =
                        new rentInputScreen(ptrTenantList, ptrRentRecord);
                        ptrRentInputScreen->getRent();
                        delete ptrRentInputScreen;
                        break;
                case 'e': ptrExpenseInputScreen =
                        new expenseInputScreen(ptrExpenseRecord);
                        ptrExpenseInputScreen->getExpense();
                        delete ptrExpenseInputScreen;
                        break;
                default: cout << "Невідома функція" << endl;
                        break;
            } // <------> Кінець секції switch
        } // <------> Кінець умови if
        else if(ch=='d') {    // Виведення даних
            cout << " 't' для виведення мешканців\n"
                << " 'r' для виведення орендної плати\n"
                << " 'e' для виведення витрат\n"
                << " 'a' для виведення річного звіту: ";
            ch = getaChar();
            switch(ch) {
                case 't': ptrTenantList->Display();
                        break;
                case 'r': ptrRentRecord->Display();
                        break;
                case 'e': ptrExpenseRecord->Display();
                        break;
                case 'a':
                        ptrAnnualReport = new annualReport(ptrRentRecord,
                        ptrExpenseRecord);
                        ptrAnnualReport->Display();
                        delete ptrAnnualReport;
                        break;
                default: cout << "Невідома функція виведення" << endl;
                        break;
            } // <------> Кінець switch
        } // <------> Кінець elseif
        else if(ch=='q')
            return; // Вихід
        else
            cout << "Невідома функція" << endl;
                << "Натискайте тільки 'i', 'd' або 'q'" << endl;
        } // <------> Кінець while
    } // <------> Кінець interact()

    // <------> Кінець файлу landlord.cpp
```

Вимушені спрощення коду програми. Хоча код програми вийшов достатньо довгим, та все ж таки він ще містить велику кількість допущень і спрощень. Текстовий інтерфейс користувача замість меню, віконець і сучасної графічної оболонки; майже повна відсутність перевірок на помилки введення даних; підтримка даних тільки за один рік тощо.

Взаємодія користувача з програмою. Отож, ми, наче у казці, пройшли крізь "вогонь і воду" – спланували та написали програму. Тепер цікаво було б подолати й "мідні труби" – подивитися на неї "в дії". Ось підходить до комп'ютера Степан Полатайко і натискає "i", а потім "t" для введення нового мешканця. Після відповідних запитів програми (у дужках у кінці запиту зазвичай пишуть формат даних) він вводить інформацію про мешканця.

```
Натисніть 'i' для введення даних.
'd' для виведення звіту
'q' для виходу: i
Натисніть 't' для додавання мешканця
'r' для запису орендної плати
'e' для запису витрат: t
Введіть ім'я мешканця (Тарас Редько): Анатолій Бондарчук
Введіть номер кімнати: 101
```

Після введення усіх мешканців домовласник зажадав проглянути їх повний перелік (скорочено обмежимося п'ятьма мешканцями з дванадцяти):

```
Натисніть 'i' для введення даних
'd' для виведення звіту
'q' для виходу: d
Натисніть 't' для виведення мешканців
'r' для виведення орендної плати
'e' для виведення витрат
'a' для виведення річного звіту: t

Apt #   Ім'я мешканця
----------------
101     Анатолій Бондарчук
102     Петро Ничипорук
103     Тарас Редько
104     Валентина Петрова
201     Ігор Склярук
```

Для фіксації внесеної орендної плати домовласник Степан Полатайко, кум знаменитого авторитета Валерія Івановича Міщука з вулиці Левандівка Залізничного району м. Львова, натискає спочатку "i", потім "r" (далі ми не повторюватимемо пункти меню у прикладах роботи коду програми). Подальша взаємодія з програмою відбувається так:

```
Введіть ім'я мешканця: Петро Ничипорук
Введіть внесену суму (345.67): 595
Введіть місяць оплати (1-12): 5
```

Петро Ничипорук послав домовласнику чек оплати за травень у розмірі 595 грн. Зрозуміло, ім'я мешканця повинно бути надруковано так само, як воно з'являлося в переліку мешканців. Окрім цього, дещо розумніша програма, можливо, дала б гнучкіше вирішення цього питання.

Щоб побачити всю таблицю доходів від оренди приміщень, потрібно натиснути 'd', а потім "r". Ось який стан таблиці після внесення травневої орендної плати:

```
Apt No Січ Лют Бер Квіт Трав Черв Лип Серп Вер Жоат Лист Груд
---------------------------------------------------
101    695 695 695 695 695 0 0 0 0   0 0 0
102    595 595 595 595 595 0 0 0 0   0 0 0
103    810 810 825 825 825 0 0 0 0   0 0 0
104    645 645 645 645 645 0 0 0 0   0 0 0
201    720 720 720 720 720 0 0 0 0 0 0 0
```

Зверніть увагу, оплату для ТарасаРедька з березня було збільшено.

Щоби ввести значення витрат, потрібно натиснути "i" та 'e'. Наприклад:

```
Введіть місяць: 1
Введіть день: 15
Введіть категорію витрат (Ремонт, Податки): Комунальні послуги
Введіть отримувача (ЛьвівЕлектроЗбут): ЛЕЗ
Введіть суму платежу: 427.23
```

Для виведення на екран таблиці витрат необхідно натиснути 'd' і "e". Нижче показано початок такої таблиці:

```
Дата Отримувач Сума Категорія............ 714
-------------
1/3 УльтраМегаБанк 5187.30 Сплата за заставу
1/8 ПростоВодоканал 963.0 Комунальні послуги
1/9 Суперстрах 4840.00 Страхування
1/15 ПЭС 727.23 Комунальні послуги
1/22 Непотріб дядька Сема 54.81 Постачання
1/25 Підвал Майстра 150.00 Ремонт
2/3 УльтраМегаБанк 5187.30 Сплата за заставу
```

Нарешті, для виведення річного звіту користувач повинен натиснути 'd' і "e". Подивимося на звіт за перші п'ять місяців року:

```
Річний звіт
-------------
Доходи
Орендна плата  42610.12
Витрати
Сплата за заставу      25936.57
Комунальні послуги     7636.15
Реклама         95.10
Ремонт          1554.90
Постачання      887.22
Страхування     4840.00
Баланс          1660.18
```

Категорії витрат сортуються в алфавітному порядку. У реальній ситуації може бути достатньо багато бюджетних категорій, одні з яких містять податки, інші – витрати на поїздки, впорядкування ландшафтного дизайну дворової території, прибирання приміщень і т.д.

Труднощі написання коду програми. Процес розроблення реальних проектів може відбуватися зовсім не так гладко, як це було продемонстровано у нашому прикладі. Може знадобитися не одна, а декілька ітерацій кожного з розглянутих вище етапів. Програмісти можуть по-різному уявляти собі потреби користува-

чів, що вимагатиме повернення з середини етапу побудови на етап удосконалення. Користувачі також можуть зненацька змінити свої вимоги, не дуже замислюючись, які незручності вони тим самим створюють для розробників.

* * *

Для простих програм під час їх розроблення може бути достатньо методу проб і помилок. Але під час розроблення крупних проектів потрібен більш організований підхід. У цьому розділі ми розглянули один з можливих методів розроблення ПЗ під загальною назвою уніфікований процес. Він складається з таких етапів: початок, удосконалення, побудова і впровадження. Етап удосконалення відповідає програмному аналізу, а побудова – плануванню структури програми і написанню її коду.

В уніфікованому процесі використовують прецедентний підхід до розроблення сучасного ПЗ. Ретельно вивчаються потенційні користувачі та аналізуються їх вимоги. Діаграма варіантів використання UML демонструє діючі суб'єкти і виконувані операції, що ініціюються ними (варіанти використання). Будь-який іменник з описів варіантів використання може в майбутньому стати іменем класу або атрибуту. Дієслова перетворюються на методи.

Доповненням до діаграм варіантів використання є ще багато інших UML-діаграм, що допомагають більш повною мірою досягти взаєморозуміння між користувачами і розробниками. Відносини між класами показують на діаграмах класів, керівні потоки – на діаграмах дій, а діаграми послідовностей відображають взаємозв'язки між об'єктами під час виконання варіантів використання.

Додаток А. ОСОБЛИВОСТІ РОЗРОБЛЕННЯ КОНСОЛЬНИХ ПРОГРАМ У СЕРЕДОВИЩІ BORLAND C++ BUILDER

У цьому додатку розглянуто питання, які стосуються використання інтегрованого середовища Borland C++ Builder для створення консольних кодів програм, що повністю відповідають прикладам, наведеним у цьому посібнику.

C++ Builder – це найкраще інтегроване середовище розробника, яке випускає фірма Borland. Воно повністю узгоджується із стандартом мови програмування C++, не виникає жодних проблем з сумісністю середовища і мови. Існує студентська версія системи, вартість якої становить приблизно $100. Також є і безкоштовна її версія, яку можна скачати з сайту фірми Borland. Якщо у Вас виникне бажання нею скористатися, то коди програм доведеться писати в текстовому редакторі Notepad або будь-якому, подібному до нього. У цьому додатку розглядатимемо інтегроване середовище C++ Builder версії 6.0.

Будемо вважати, що систему встановлено на Вашому комп'ютері, і Ви умієте її запускати у середовищі Windows. Вам знадобиться, щоб на екрані відображалися розширення файлів (наприклад, *.cpp), потрібних для роботи з середовищем MVC++. Переконайтеся, що в настройках **Провідника** вимкнено функцію, яку приховує розширення зареєстрованих типів файлів.

У цьому ж додатку ми також детально розглянемо питання використання середовища C++ Builder для редагування, компілювання, зв'язування і виконання консольних програм.

А.1. Основні правила роботи у середовищі C++ Builder

Консольні програми з цього навчального посібника вимагатимуть мінімальних змін для запуску з середовища C++ Builder. Дамо короткі рекомендації щодо основних правил роботи при створенні консольних програм у інтегральному середовищі C++ Builder.

Підготовка панелей інструментів. Початкове завантаження інтегрованого середовища C++ Builder призводить до того, що на екрані з'являються безліч об'єктів, які Вам, у принципі, не знадобляться при створенні консольних програм. Справа Ви побачите вікно, яке називається **Form1**. Закрийте його (клікнувши на хрестику в правому верхньому кутку). Також Вам не знадобиться і **Object Inspector** (**Інспектор об'єктів**), закрийте і його. Вам доведеться позбавлятися від цих віконець під час кожного запуску C++ Builder.

Коли Ви закриєте вікно **Form1**, то виявите під ним ще одне вікно з деяким початковим кодом, написаним мовою C++. Це вікно називається *редактором коду програми.* Саме тут Ви проглядатимете початкові файли і писатимете свої програми. Проте під час запуску коду програми Ви бачите файл **Unit1**, який Вас абсолют-

но не цікавить. Закрийте вікно і натисніть **No**, якщо система запитає, чи не хочете Ви зберегти зміни.

Під час початкового запуску середовища C++ Builder на екрані з'являється безліч панелей інструментів, значна частина з яких Вам також не буде потрібною. Вам, мабуть, знадобляться панелі **Стандартна** (**Standard**) і **Debug** (**Відлагодження**). Можете прибрати всю решту панелей, вибравши з меню **Вигляд** (**View**) пункт **Toolbars** (**Панелі інструментів**) і прибравши галочки перед їх іменами.

Утримання на екрані вікна з результатами роботи коду програми. Відкомпілювати і запустити у вікні **Сесії MS-DOS** усі консольні програми можна, взагалі не вносячи ніяких змін. Проте, якщо Ви хочете запускати їх з середовища C++ Builder за допомогою команди **Run** (**Пуск**) з однойменного меню, потрібно буде вставити в кінець коду програми рядки, які утримуватимуть на екрані вікно з результатами роботи коду програми достатньо довго. Це можна зробити за два кроки:

- вставте функцію **getch()**; перед останнім **return** в секції **main**(). За допомогою цієї функції Ви зможете побачити результати роботи коду програми на екрані;
- вставте вираз **#include <conio.h>** у початок **main**(). Це необхідно для функції **getch()**.

Якщо у програмі використовується консольна графіка, то потрібно ще інші додаткові зміни, про які буде сказано в цьому додатку дещо пізніше.

Створення нового консольного проекту. Середовище C++ Builder, як і інші сучасні компілятори, мислить в термінах *проектів* при створенні консольних програм. Консольний проект складається з одного або декількох початкових файлів, а також з декількох додаткових файлів, таких як файли ресурсів і визначень. Втім, вони нам спочатку не знадобляться. Результатом створення консольної програми є зазвичай один виконуваний *.exe-файл.

Щоб почати створювати проект, потрібно вибрати пункт **New...** (**Новий...**) з меню **File** (**Файл**). Ви побачите діалогове вікно, що називається **New Items** (**Нові елементи**). Виберіть закладку **New** (**Нові**), якщо це необхідно. Потім клікніть двічі на значку **Console Wizard** (**Майстер консольних додатків**). Переконайтеся в тому, що в діалоговому вікні у пункті меню **Source Type** (**Тип коду програми**) вибрано C++ і що встановлений опція **Console Application** (**Консольний додаток**). Зніміть опції **Use VCL**, **Multi Threaded** і **Specify Project Source**. Натисніть **<OK>**. Відповідайте негативно, якщо система запитає, чи зберегти зміни в **Project1**. У новому вікні редактора коду програми Ви побачите такий початковий текст:

```
//---------------------------------------------------------------
#pragma hdrstop

//---------------------------------------------------------------

#pragma argsused
int main(int argc,char **argv [])
{
	getch(); return 0;
}
//---------------------------------------------------------------
```

Це як би скелет майбутньої консольної програми. Деякі з цих рядків Вам будуть не потрібні, а деякі доведеться додати. Ми зараз виконаємо необхідні зміни і додамо вираз, що виводить який-небудь текст, щоб переконатися в тому, що програма працює. Результат:

```
//test1.cpp
#include <vcl>
#include <iostream>            // Для потокового введення-виведення
#include <conio>               // Для консольного режиму роботи
using namespace std;           // Використання стандартного простору імен
//#pragma hdrstop //не знадобиться
//#pragma argsused // не знадобиться

int main() // Аргументи не потрібні
{
	cout << "Щасливі ті, хто вперше компілює програму.";

	getch(); return 0;
}
```

Дві директиви **#**pragma виявилися не потрібні, також не потрібні виявилися аргументи **main**().

Якщо Ви у такому вигляді запустите програму, то переконаєтеся в тому, що вікно з результатами дуже швидко зникають з екрану. Це виправляється вставкою функції **getch**(); у кінець програми, безпосередньо перед останнім **return**. У результаті такої зміни програма чекатиме натиснення якої-небудь клавіші і тільки після цього прибере вікно виведення результатів з екрану. Функції **getch**() потрібен заголовний файл **conio.h**, який ми включаємо за допомогою відповідного виразу на початку програми.

Якщо Ви створюєте свою програму, то Вам знадобиться цей скелет, його Ви і змінюватимете і доповнюватимете. Якщо у Вас вже є якийсь файл, то читайте підрозділ "Робота з наявними файлами" даного додатку.

Надання імені консольному проекту та збереження його. Перейменувати і зберегти потрібно як початковий файл, так і консольний проект, у якому він знаходиться. Компілятор автоматично іменує початковий файл **Unit1.cpp**. Для того, щоби його перейменувати і зберегти, виберіть пункт **Save As...** (**Зберегти як...**) з меню **File** (**Файл**). Знайдіть потрібний каталог для свого проекту, назвіть більш-менш осмислено свій файл (зберігши при цьому розширення *.cpp) і натисніть **Save** (**Зберегти**).

Інформація про консольний проект зберігається у файлі з розширенням *.bpr. Тому коли Ви зберігаєте проект, то Ви насправді запам'ятовуєте стан як файлу (або файлів) *.cpp, так і файлу *.bpr. При створенні нового проекту йому дається автоматично ім'я **Project1** (або **Project** з будь-якою іншою цифрою). Щоб зберегти проект і змінити його назву, виберіть пункт **Save Project As...** (**Зберегти проект як...**) з меню **File** (**Файл**). Перейдіть в каталог, у якому Ви хочете зберегти файл, наберіть його ім'я з розширенням *.bpr і натисніть **Save** (**Зберегти**).

Робота з наявними файлами консольних програм. Розглянемо перелік дій, які потрібно виконати при створенні консольного проекту при наявному початко-

вому файлі. Наприклад, якщо Ви скачаєте з сайту будь-якого видавництва приклади кодів програм, то якраз опинитеся в такій ситуації. Ми тут розглядатимемо тільки консольні програми з одним початковим файлом.

Допустимо, що файл називається myprog.cpp. Переконайтеся в тому, що він знаходиться в тому самому каталозі, у якому Ви створюватимете проект. Виберіть пункт **Open** (**Відкрити**) з меню **File** (**Файл**). Виберіть файл в діалоговому вікні і натисніть **Open** (**Відкрити**). Файл з'явиться у вікні редактора коду програми. Вискочить діалогове віконце. У ньому буде питання, чи хочете Ви створити проект, у якому даний файл компілювався б і запускався. Відповідайте ствердно. Проект, таким чином, створений.

Тепер потрібно його перейменувати і зберегти. Виберіть з меню **File** (**Файл**) пункт **Save Project As...** (**Зберегти проект як...**). Замініть ім'я project1.bpr на яке-небудь інше (зазвичай вибирають співпадаюче з іменем програми): myprog.bpr. Натисніть **Save** (**Зберегти**). От і все.

А.2. Компілювання, зв'язування та запуск консольних програм на виконання

Щоб створити виконуваний файл, виберіть пункт **Make** (**Зробити**) або **Build** (**Компонування**) з меню **Project** (**Проект**). З Вашого *.cpp буде створено об'єктний файл *.obj, а з нього, своєю чергою, файл *.exe. Наприклад, якщо Ви компілюєте myprog.cpp, то результатом буде створення файлу myprog.exe. Якщо будуть виявлені помилки компілятора або компонувальника, то вони будуть зразу ж відображені. Постарайтеся домогтися їх зникнення.

Запуск консольної програми в середовищі C++ Builder. Якщо Ви модифікували код програми, вставивши в неї вираз **getch**() (див. розд. А.1), то Ви можете відкомпілювати, зв'язати і запустити консольну програму прямо з середовища C++ Builder буквально однією командою **Run** (**Пуск**) з однойменного меню. Якщо ніяких помилок не виявлено, з'явиться вікно отриманих результатів роботи коду програми.

Запуск програми в MS DOS. Можна запустити свою програму і з-під MS DOS. Сесія MS DOS відкривається в Windows з меню **Start 〉 Programs** (**Пуск 〉 Програми**) за допомогою ярлика **MS-DOS Prompt** (Сесія MS-DOS). В результаті Ви побачите чорне віконце із запрошенням MS DOS. Переміщайтеся по каталогах, використовуючи команду **cd <ім'я каталога>**. Для того, щоби запустити будь-які програми, що були заздалегідь скомпільовані, зокрема ті, тексти яких є в навчальному посібнику, то переконайтеся, що Ви знаходитеся в тому ж каталозі, що і потрібний файл *.exe, і наберіть ім'я програми. Детальнішу інформацію можна отримати, скориставшись **Довідковою системою Windows**.

Заздалегідь скомпільовані заголовні файли. Процес компілювання можна істотно прискорити, якщо використовувати заголовні файли, що були заздалегідь скомпільовані. Для цього потрібно вибрати пункт **Options** (**Параметри**) з меню **Project** (**Проект**), перейти до закладки **Compiler** (**Компілятор**) і встановити оп-

цію **Use Precompiled Headers** (**Використовувати прекомпільовані заголовні файли**). У коротких програмах більше всього часу витрачається на компілювання службових заголовних файлів С++, таких, як **iostream**. Включення вказаного параметра дає змогу скомпілювати ці файли тільки один раз.

Закриття та відкриття консольних проектів. Коли Ви закінчуєте роботу над проектом, то його необхідно закрити, вибравши пункт **Close All** (**Закрити все**) з меню **File** (**Файл**). Щоб відкрити вже наявний проект, потрібно з того ж меню вибрати пункт **Open Project** (**Відкрити проект**), перейти у відповідну директорію і двічі клікнути на імені файлу з розширенням *.bpr.

А.3. Додавання заголовного файлу до консольного проекту

Серйозні консольні програми, написані мовою програмування С++, можуть працювати з одним або навіть декількома призначеними для користувача заголовними файлами (і це на додаток до безлічі бібліотечних файлів, таких, як **iostream** і **conio**). Продемонструємо, як додати їх до консольного проекту.

Створення нового заголовного файлу. Виберіть пункт **New...** (**Новий...**) з меню **File** (**Файл**), переконайтеся в тому, що Ви знаходитеся на закладці, в якій відображаються нові елементи, клікніть двічі на значку **Text** (**Текст**). З'явиться вікно редактора коду програми з файлом **file1.txt**. Наберіть текст і збережіть файл за допомогою пункту **Save as...** (**Зберегти як...**) з меню **File** (**Файл**). Придумайте ім'я для файлу, але не забудьте залишити розширення *.h. Цей файл потрібно зберегти в тій самій директорії, де знаходяться Ваші початкові (*.cpp) файли. Ім'я збереженого файлу з'явиться у вигляді однієї із закладок вікна редактора. Тепер для редагування різних файлів можна просто перемикатися між закладками.

Редагування заголовного файлу. Щоб відкрити вже наявний заголовний файл, виберіть **Open** (**Відкрити**) з меню **File** (**Файл**), вкажіть в полі **Files of Type** (**Типи файлів**) значення *.* (**Всі файли**). Тепер можна вибрати із переліку потрібний заголовний файл.

У процесі написання виразу **#include** для заголовного файлу в початковому не забудьте переконатися в тому, що ім'я файлу поміщене в лапки:

```
#include "myHeader.h"
```

Ці лапки вкажуть компіляторові, що він повинен шукати заголовний файл в тому ж каталозі, де знаходяться початкові файли.

Визначення місцезнаходження заголовного файлу. Якщо вже Ви додаєте файл *.h, то потрібно повідомити компілятор про його місцезнаходження. Якщо він знаходиться в директорії з іншими Вашими файлами, що стосуються даного проекту, то нічого робити не потрібно.

Інакше потрібно вказати його розташування. Наприклад, це необхідно робити в тому випадку, якщо Ви використовуєте консольну графіку в своїй програмі. Потрібно вказати розташування файлу borlacon.h, якщо Вам лінь скопіювати його в каталог з рештою файлів. Перейдіть до пункту **Options** (**Параметри**) меню **Project** (**Проект**) і спробуйте вибрати закладку **Directories/Conditionals** (**Катало-**

ги/Імплікації). У секції **Directories** (**Каталоги**) клікніть на кнопці з трьома крапками, розташованій праворуч від переліку шляхів **Include Path**, що включаються. З'явиться діалогове вікно **Directories** (**Каталоги**).

У самому низу цього вікна наберіть повний шлях до каталога, що містить заголовний файл. Клікніть на кнопці **Add** (**Додати**), щоб занести цей каталог в перелік шляхів, що включаються. Натисніть кнопки <OK>, щоб закрилися всі діалоги. Не намагайтеся додавати заголовні файли за допомогою функції **Add to Project** (**Додати в проект**) з меню **Project** (**Проект**).

А.4. Проекти з декількома початковими файлами

Реальні додатки і навіть деякі приклади з цього навчального посібника вимагають підключення декількох початкових файлів. У C++ Builder початкові файли називаються *компонентами* (units), ця назва характерна саме для даного середовища програмування. У більшості інших середовищ файли все-таки називаються файлами, а у мові Pascal – *модулями.*

Створення додаткових початкових файлів. Додаткові *.cpp-файли можна створювати так само, як заголовні файли. Для цього потрібно вибрати **File 〉 New** (**Файл 〉 Новий**) і двічі клікнути на значку **Text** (**Текст**) в діалоговому вікні **New** (**Нові**). Наберіть початковий код програми і збережіть файл. Під час його збереження не забудьте вказати правильний тип: це повинен бути **C++ Builder Unit** (компонент C++ Builder). Розширення *.cpp підставиться автоматично після імені файлу. Якщо чомусь цього не трапилося і Ви просто наберете ім'я файлу і навіть правильне розширення, то цей файл не розпізнається як **C++ Builder unit**.

Додавання наявних початкових файлів. У Вас уже може бути створений початковий файл, наприклад borlacon.cpp, необхідний для роботи з консольною графікою. Щоб додати початковий файл в проект, виберіть пункт **Add To Project** (**Додати в проект**) з меню **Project** (**Проект**), перейдіть в необхідну директорію і виберіть файл, який хочете додати. Потім натисніть кнопку **Open** (**Відкрити**). Тепер цей файл в C++ Builder вважатиметься частиною проекту.

Початкові файли вказані в закладках вікна редагування, так що можна швидко перемикатися з одного файлу на іншій. Деякі з них можна закривати, щоб вони не займали місце на екрані, якщо вони не потрібні.

Застосування менеджера проектів. Щоб подивитися, які файли є складовими частинами проекту, потрібно вибрати **Project Manager** (**Менеджер проектів**) з меню **View** (**Вигляд**). Ви побачите діаграму, на якій показані зв'язки між файлами, приблизно так само як і у **Провіднику Windows**. Натиснувши на плюс поряд із значком проекту, можна побачити всі файли, що входять в проект. Файл, який Ви тільки що додали в проект, повинен бути серед них.

Якщо натиснути правою кнопкою миші на **Project Manager** (**Менеджері проектів**), з'явиться контекстне меню, з якого можна вибрати наступні дії: **Open** (**Відкрити**), **Close** (**Закрити**), **Save as...** (**Зберегти як...**) і **Compile** (**Компілювати**). Це зручний спосіб роботи з окремими початковими файлами.

У багатофайлових програмах можна компілювати початкові файли незалежно один від одного. Для цього потрібно вибрати пункт **Compile Unit** (**Компілювати Компонент**) з меню **Project** (**Проект**). При цьому перекомпілюються тільки ті файли, які були змінені з часу попередньої збірки програми.

Програми з консольною графікою. Опишемо, як здійснювати компонування програм, в яких використовується консольна графіка. Отже, за порядком.

1. Створіть новий проект, як це було описано вище. Як ім'я проекту найкраще використовувати ім'я програми. Поміняйте тільки розширення на *.bpr.
2. Потім у початковому файлі коду програми поміняйте рядок **#include <msoftcon.h>** на **#include <borlacon.h>**.
3. Скопіюйте (саме скопіюйте, а не переносіть) файли borlacon.cpp і borlacon.h у файл проекту (чи ж повідомте компілятор, де шукати заголовний файл).
4. Додайте початковий файл borlacon.cpp у свій проект, дотримуючись настанов, описаних трохи вище. Додавання наявних початкових файлів.
5. Щоб затримати на екрані вікно отриманих результатів роботи коду програми, потрібно вставити рядок **getch**(); безпосередньо перед останнім оператором **return** в кінці секції **main**().
6. Для підтримки **getch()** потрібно залучити до програми ще один заголовний файл. Для цього напишіть на початку програми рядок **#include <conio.h>** // Для консольного режиму роботи.

Тепер можна компілювати, пов'язувати і запускати програми з використанням консольної графіки так само, як і будь-які інші коди програм.

А.5. Відлагодження коду програми

У розділі "Настанови керування ходом виконання С++-програми" ми використовували відлагоджувач в основному з метою повнішого ознайомлення з процесом роботи циклів. Під час використання середовища С++ Builder ті самі кроки можуть допомогти Вам знайти помилку у програмі і усунути її. Ми розглядатимемо тільки однофайлові консольні програми, але необхідно зазначити, що відлагодження багатофайлових проектів нічим принципово не відрізняється.

Спершу просто створіть і відкомпілюйте яку-небудь програму. Позбавтеся усіх помилок компілятора і компонувальника. Переконайтеся, що код програми виводиться у віконці редагування.

Покроковий перегляд роботи коду програми. Щоб запустити відлагоджувач, натисніть клавішу **F8**. Вся програма буде перекомпільована, а перший її рядок (зазвичай описувач **main**()) – підсвічується. Послідовні натиснення **F8** приведуть до просування по коду програми від одного оператора до іншого. Якщо Ви проганяєте цикл, то побачите, як стрілка доходить до останнього виразу, а потім переходить знов на початок циклу.

Перегляд значень змінних коду програми. Існує можливість переглядати значення змінних у процесі покрокового трасування коду програми. Виберіть пункт **Add watch** (**Додати змінну**) з меню **Run** (**Пуск**). З'явиться діалогове вікно **Watch Properties** (**Параметри перегляду**). Наберіть ім'я змінної в полі **Вираз**,

потім виберіть її тип і натисніть <OK>. З'явиться вікно **Watch List** (**Перелік спостереження**). Виконуючи вказані кроки кілька разів, можна додати в цей перелік будь-яку необхідну кількість змінних. Якщо Ви нормально розмістите на екрані вікно перегляду і вікно редагування, то зможете спостерігати одночасну зміну значень змінних залежно від виконуваної в даний момент команди. Якщо змінна поки що недоступна (наприклад, ще не визначена у програмі), то вікно перегляду видасть повідомлення про помилку замість значення поряд з іменем змінної.

У випадку з програмою cubelist механізм спостереження змінних не розпізнає змінну cube, оскільки вона визначається усередині циклу. Перепишіть програму так, щоб вона визначалася поза циклом, і відразу побачите результат: у вікні спостереження вона відображається так, як це необхідно.

Покрокове трасування функцій. Якщо у вашій програмі використовуються функції, то майте на увазі, що є чудова можливість здійснити покрокове трасування кожного виразу функції. Це робиться за допомогою клавіші **F7**. На відміну від режиму трасування по **F8**, який розглядає кожну зустрінуту функцію як один вираз, такий режим дає змогу "увійти" всередину функції та виявити можливі помилки в ній. Якщо Ви примусите по **F7** відбуватися бібліотечні функції, наприклад **cout <<**, то милуватиметеся початковим кодом бібліотеки. Це процес довгий, тому радимо користуватися цією можливістю, тільки якщо Вам дійсно важливо знати, що відбувається усередині рутинних функцій. Втім, можна в міру потреби чергувати натиснення **F7** і **F8**, залежно від необхідності виявлення помилок у внутрішніх функціях.

Точки зупинки роботи коду програми. Точки зупинки дають змогу тимчасово переривати роботу програми в довільних місцях. Коли їх потрібно використовувати? Ми вже показали, як можна примусити відлагоджувач проганяти програму тільки до курсора. Але бувають випадки, коли необхідно мати декілька таких точок. Наприклад, можна зупинити програму в кінці умови **if**, а також після відповідної **else**. Точки зупинки вирішують цю задачу, оскільки їх можна ставити в необмеженій кількості. У них є ще і розширені функції, які ми тут розглядати не будемо.

Розглянемо, як вставити точки зупинки в код програми. По-перше, погляньте на код програми у вікні редагування. Напроти кожного виконуваного оператора Ви побачите крапочку зліва. Просто клікніть мишею на тій крапочці, напроти якої Ви збираєтеся вставити точку зупинки. Крапочка заміниться на червоний кружечок, а рядок коду програми підсвічуватиметься. Тепер при будь-якому варіанті запуску програми в середовищі програма зупинятиметься саме тут. Можна на цьому тимчасовому зрізі перевірити значення змінних, провести покрокове трасування коду програми і т.д.

Щоб видалити точку зупинки, клікніть мишкою на червоному кружечку. Він зникне. Є ще багато інших функцій відлагоджувача. Тут ми показали тільки основні, щоб Вам було з чого почати роботу.

Додаток Б. ОСОБЛИВОСТІ РОЗРОБЛЕННЯ КОНСОЛЬНИХ ПРОГРАМ У СЕРЕДОВИЩІ MICROSOFT VISUAL C++

У цьому додатку розглядаються питання використання Microsoft Visual C++ (MVC++) для створення консольних програм, наприклад, таких, які наводилися як навчальні приклади в цьому навчальному посібнику. Розмова піде про середовище MVC++ версії 6.0.

Справжня версія середовища MVC++ є спадкоємцем стандартного середовища С++. Вона постачається в різних варіантах, включаючи недорогу студентську комплектацію. Будемо вважати, що система MVC++ встановлена на Вашому комп'ютері і Ви знаєте, як запустити її у Windows.

Вам знадобиться, щоб на екрані відображалися розширення файлів (наприклад *.cpp), потрібних для роботи з середовищем MVC++. Переконайтеся, що в настройках Провідника вимкнена функція, яка приховує розширення зареєстрованих типів файлів.

Б.1. Елементи вікна середовища MVC++

Після запуску системи MVC++ вікно середовища розділено на три частини. Зліва знаходиться панель **View** (**Вигляд**). У неї дві закладки: **ClassView** (**Класи**) і **FHeView** (**Файли**). Якщо Ви відкриєте який-небудь проект, то на цій панелі в закладці **ClassView** (**Класи**) відображатиметься ієрархія класів програми, а в закладці **FileView** (**Файли**) – перелік файлів, з яких складається проект. При натисненні на плюсик показується наступний рівень ієрархії. Для того, щоби відкрити якийнебудь файл, потрібно двічі клікнути на його імені.

Найбільше місця вікні середовища MVC++ зазвичай займає та його частина, в якій знаходиться текст відкритого файлу. Його можна використовувати в різних цілях, зокрема для відображення початкового коду програми або різної довідкової інформації. Внизу на екрані знаходиться віконце з безліччю закладок: **Build** (**Компонування**), **Debug** (**Відлагодження**) і т.д. Тут також будуть відображатимуться різні повідомлення компілятора про такі виконувані дії, як, наприклад, відлагодження програми.

Б.2. Робота з однофайловими консольними програмами

З допомогою середовищем MVC++ можна достатньо легко створювати однофайлові консольні програми і виконувати роботу з ними: правити, компонувати, виправляти помилки тощо.

Можливі два варіанти роботи з однофайловими консольними програмами: або файл вже існує і його потрібно правити, або файлу ще немає. У будь-якому випадку починати необхідно, тільки переконавшись в тому, що немає відкритих

проектів (що таке проект, ми незабаром про це взнаємо). Натисніть на меню **File** (**Файл**). Якщо команда закриття робочої області **Close Workspace** недоступна, значить, відкритих проектів немає і можна приступати до роботи над своїм текстом консольної програми. Якщо ж вона доступна, то натисніть на неї для закриття відкритого у даний момент проекту.

Компонування наявного файлу. Якщо початковий файл *.cpp вже існує, то виберіть пункт **Open** (**Відкрити**) з меню **File** (**Файл**). Майте на увазі, що це не те ж саме, що **Open Workspace** (**Відкрити робочу область**). У діалоговому вікні, що з'явилося на екрані, знайдіть потрібний файл, виберіть його і клікніть на кнопці **Open** (**Відкрити**). Файл з'явиться у вікні документа (якщо у програмі використовується консольна графіка, як у прикладі cidcstrc, зверніться до розділу "Програми створення консольного оформлення" даного додаток).

Для компілювання і компонування початкового файлу виберіть пункт **Build** (**Компонування**) з однойменного меню. У діалоговому вікні з'явиться запит на створення звичайної робочої області проекту. Відповідайте ствердно. Файл відкомпілюється і буде пов'язаний зі всіма необхідними бібліотечними файлами.

Для запуску програми виберіть **Execute** (**Запустити**) з меню **Build** (**Компонування**). Якщо все зроблено правильно, то з'явиться вікно з результатами роботи коду програми.

Після закінчення роботи коду програми у вікні з'явиться фраза Натисніть будь-яку клавішу для продовження. Її вставляє в кінець програми компілятор. Вікно отриманих результатів роботи має бути видно впродовж тривалого періоду, щоб Ви могли їх переглянути.

Якщо Ви закінчуєте роботу з програмою, то закрийте її робочу область за допомогою пункту **Close Workspace** (**Закрити робочу область**) з меню **File** (**Файл**). Відповідайте ствердно, коли в діалоговому вікні з'явиться запит на закриття усіх вікон з документами. Програму можна запустити і вручну, безпосередньо з MS DOS. Сесія MS DOS відкривається у Windows з меню **Start** 〉 **Programs** (**Пуск** 〉 **Програми**) за допомогою ярлика MS-DOS **Prompt** (Сесія MS-DOS). В результаті Ви побачите чорне віконце із запрошенням MS DOS. Переміщайтеся по каталогах, використовуючи команду **cd <ім'я каталога>**. Виконувані (*.exe) файли програм, створених компілятором середовища MVC++, розташовуються в підкаталозі DEBUG каталога, у якому зберігаються файли проектів. Для того, щоби запустити будь-які програми, відкомпільовані у середовищі MVC++, зокрема ті, тексти яких є в цьому навчальному посібнику, переконайтеся, що Ви знаходитеся в тому ж каталозі, що і файл *.exe, і наберіть ім'я програми. Детальнішу інформацію можна отримати, скориставшись **Довідковою системою Windows**.

Створення нового файлу. Щоб створити свій файл *.cpp, закрийте робочу область відкритого проекту, виберіть пункт **New** (**Новий**) з меню **File** (**Файл**). Клікніть на закладці **Files** (**Файли**). Виберіть потрібний тип файлу **C++ Source File** (**Початковий файл C++**), наберіть ім'я файлу, переконавшись перед тим у тому, що Ви знаходитеся в потрібному каталозі. Клікніть на кнопці <OK>. З'явиться чисте вікно документа. У ньому можна набирати код консольної програми. Не за-

будьте зберегти файл, вибравши пункт **Save As**... (**Зберегти як**...) з меню **File** (**Файл**). Потім відкомпілюйте свою програму так, як це вже було описано вище.

Виправлення помилок. Якщо у програмі є помилки, повідомлення про них з'являтимуться у вікні **Build** (**Компонування**) внизу екрану. Якщо двічі клікнути на рядку з якою-небудь помилкою, з'явиться стрілка у вікні документа, що вказує на той рядок у коді програми, де ця помилка відбулася. Окрім цього, якщо у вікні **Build** (**Компонування**) поставити курсор на рядок з кодом помилки (наприклад, С2143) і натиснути клавішу **F1**, то у вікні документа з'являться коментарі до тієї чи іншої помилки. Помилки потрібно усувати доти, доки не з'явиться повідомлення 0 error(s), 0 warning(s) (0 помилок, 0 попереджень). Для запуску програми виберіть пункт **Execute Build** (**Виконання компонування**) з меню **Build** (**Компонування**).

Однією з поширених помилок є відсутність в коді програми рядка **using namespace std;**
// Використання стандартного простору імен

Якщо забути включити його у програму, то компілятор повідомить, що він не знає, що таке **cout << endl** і т.ін.

Перед початком роботи над новою програмою не забувайте закривати відриті проекти. Це гарантує, що Ви дійсно почнете створювати новий проект. Щоб відкрити вже скомпоновану програму, виберіть **Open Workspace** (**Відкрити робочу область**) з меню **File** (**Файл**). У діалоговому вікні перейдіть в потрібну директорію і двічі клікніть на файлі з розширенням *.dsw.

Інформація про типи в процесі виконання (RTTI). Деякі програми, такі, як empl_io.cpp з розд. 9 "С++-система введення-виведення потокової інформації", використовують RTTI. Можна налаштувати середовищем MVC++ так, щоб ця функція працювала.

Виберіть пункт **Settings** (**Настройки**) з меню **Project** (**Проект**) і клікніть на закладці **C/C++**. Із переліку категорій виберіть **C++ Language** (**Мова С++**). Встановіть опція **Enable Run-Time Type Information** (**Включити RTTI**). Потім натисніть <OK>. Ця функція системи MVC++ дає змогу уникнути появи багатьох помилок компілятора і компонувальника, деякі з яких взагалі тільки вводять в оману.

Б.3. Робота з багатофайловими консольними програмами

Вище було продемонстровано, як можна швидко, але непрофесійно створювати консольні програми. Цей спосіб, можливо, і хороший для однофайлових програм. Але коли в проекті більше одного файлу, все відразу стає значно складнішим. Почнемо з термінології. З'ясуємо з поняттями *робоча область* і *проект.*

Поняття про проекти і робочі області. У середовищі MVC++ використовується концепція робочих областей, що на один рівень абстракції вище, ніж концепція проекту. Річ у тому, що в робочій області може міститися декілька проектів. Вона складається з каталогу і декількох конфігураційних файлів. Усередині неї кожен проект може мати власний каталог чи ж файли усіх проектів можуть зберігатися в одному каталозі робочої області.

Напевно, концептуально простіше припустити, що у кожного проекту є своя окрема робоча область. Саме це ми і припускатимемо під час подальшого з'ясування.

Проект відповідає додатку (програмі), яку Ви розробляєте. Він складається зі всіх файлів, потрібних програмі, а також з інформації про ці файли і їх компонування. Результатом всього проекту зазвичай є один виконуваний *.exe файл (можливі й інші результати, наприклад файли *.dll).

Робота над проектом. Припустимо, що файли, які Ви збираєтеся включити в свій проект, вже існують і знаходяться у відомому Вам каталозі. Виберіть пункт **New** (**Новий**) з меню **File** (**Файл**) і клікніть на закладку **Projects** (**Проекти**) в діалоговому вікні, що з'явилося. Виберіть із переліку **Win32 Console Application**. У полі **Location** (**Розміщення**) виберіть шлях до каталога, але саме ім'я каталога *не вказуйте.* Потім наберіть ім'я каталога, що містить файли, в полі **Project Name** (**Ім'я проекту**) (клікнувши на кнопці, що знаходиться праворуч від поля **Location** (**Розміщення**), Ви зможете вибрати потрібний каталог, переходячи по дереву, але переконайтеся в тому, що в цей шлях не включено саме ім'я каталога). Переконайтеся також в тому, що опція **Create New Workspace** (**Створити нову робочу область**) встановлений, і натисніть <OK>.

Наприклад, якщо файли знаходяться в каталозі **c:\Book\Chl3\Elev**, слідує в полі **Location** (**Розміщення**) вказати **c:\Book\Chl3**, а в полі **Project Name Field** (**Ім'я проекту**) –**Elev**. Ім'я проекту автоматично додається до шляху (якби в полі **Location** (**Розміщення**) був вказаний повний шлях, то підсумковий шлях до файлів був би **c:\Book\Chl3\Elev\Elev**, а це, зазвичай, зовсім не те, що нам потрібно). Після цього з'являється ще одне діалогове вікно. Необхідно переконатися в тому, що вибрана кнопка **An Empty Project** (**Порожній Проект**), і клікнути на **Finish** (**Закінчити**). У наступному вікні натисніть <OK>.

У цей час в указаному каталозі створюються різні файли, що стосуються проекту. Вони мають розширення *.dsp, *.dsw і т.д. Окрім цього, створюється порожня папка DEBUG, в якій зберігатиметься відкомпілюваний файл, готовий для подальшого виконання.

Додавання початкового файлу. А зараз до проекту потрібно причепити початкові файли, що містять файли *.cpp і *.h, які повинні бути доступні для перегляду в закладці **Файл**. Виберіть пункт **Add to Project** (**Додати до проекту**) з меню **Project** (**Проект**), клікніть на **Files** (**Файли**), виберіть потрібні файли і натисніть <OK>. Тепер файли включені в проект, і їх можна проглядати в закладці **FileView** (**Файли**) вікна **View** (**Вигляд**). Клікнувши на закладці **QassView** (**Класи**) в тому ж вікні, можна проглянути структуру класів, функцій і атрибутів.

Щоб відкрити файл проекту для перегляду і редагування, потрібно двічі клікнути на його значку в закладці **FileView** (**Файли**). Другим способом відкриття файлу є його вибір за допомогою команди **Open** (**Відкрити**) меню **File** (**Файл**).

Визначення місцезнаходження заголовних файлів. У Вашому проекті можуть використовуватися заголовні файли (зазвичай з розширенням *.h), наприклад, такі, як msoftcon.h у програмах, що використовують консольну графіку. Їх не потрібно включати в проект, якщо, зазвичай, Ви не хочете проглядати їх код прог-

рами, але компіляторові потрібно знати, де вони знаходяться. Якщо вони знаходяться в тому ж каталозі, у якому зберігаються початкові файли, то питань не виникає. Інакше потрібно повідомити компілятор про їх місцезнаходження.

Виберіть пункт **Options** (**Параметри**) з меню **Tools** (**Інструменти**). Клікніть на закладці **Directories** (**Каталоги**). Для того, щоби побачити перелік каталогів з файлами компілятора, що включаються, потрібно спочатку вибрати **Include Files** (**Файли**, **що включаються**) з пропонованого переліку під назвою **Show Directories For** (**Показати директорії**). Двічі клікніть на полі з крапочками в останньому рядку переліку. Потім перейдіть в директорію, в якій зберігається потрібний заголовний файл. Крапочки в полі будуть замінені на новий шлях до файла. Натисніть <ОК>. Ще можна набрати повний шлях до цієї директорії в полі **Location** (**Розміщення**).

Збереження, закриття і відкриття проектів. Щоб зберегти проект, виберіть пункт **Save Workspace** (**Зберегти робочу область**). Щоб закрити пункт **Close Workspace** (**Закрити робочу область**) (відповідайте ствердно на запит закриття усіх вікон документів). Для відкриття наявного проекту виберіть пункт **Open Workspace** (**Відкрити робочу область**) з меню **File** (**Файл**). Перейдіть в необхідний каталог і відкрийте потрібний файл з розширенням *.dsw. Клікніть на кнопці **Open** (**Відкрити**).

Компілювання та компонування. Як і у випадку роботи з однофайловими програмами, простим способом відкомпілювати, скомпонувати і запустити багатофайлову програму є вибір пункту **Execute** (**Виконати**) з меню **Build** (**Компонування**). Якщо запускати програму прямо з системи не потрібно, можна вибрати пункт **Build** (**Компонування**) з однойменного меню.

Б.4. Програми з консольною графікою

Програми, що використовують функції консольної графіки (описувані в дод. Д), вимагають декількох додаткових дій порівняно з іншими програмами. Вам знадобляться файли msoftcon.h і msoftcon.cpp. Вони були створені спеціально для цього навчального посібника, тому їх можна знайти тільки на сайті, адреса якого вказана у введенні.

1. Відкрийте початковий файл програми, як це було описано вище. У цьому файлі повинен бути рядок **#indude "msoftcon.h"**.
2. Виберіть пункт **Build** (**Компонування**) з однойменного меню. Відповідайте ствердно на питання про створення звичайної робочої області проекту. Буде створений проект, але компілятор видасть повідомлення про те, що не знайдений файл msoftcon.h. У цьому файлі містяться оголошення функцій консольної графіки.
3. Найпростіше скопіювати цей файл в каталог з початковими файлами. Правильніше було б повідомити компілятор про те, де він може знайти цей файл. Слідуйте настановам з розділу "Визначення місцезнаходження заголовних файлів" даного додатку.

4. Тепер знову спробуйте скомпонувати свою програму. Цього разу компілятор знайде заголовний файл, але буде довго призупиняти свою роботу від кожної знайденої в ньому "помилки", оскільки компонувальник не знає, де шукати визначення графічних функцій. А вони знаходяться у файлі msoftcon.cpp. Підключіть його до програми, як це було описано в розділі "Додавання початкових файлів" даного додаток.

Тепер програма повинна відкомпілюватися та скомпонуватися нормально. Виберіть **Execute** (**Виконати**) з меню **Build** (**Компонування**) для того, щоби подивитися, як вона працює.

Б.5. Відлагодження програм

У розділі "Настанови керування ходом виконання С++-програми" ми пропонували використовувати відлагоджувач для того, щоби легше було розібратися, що таке цикл. Тепер перед Вами конкретні рекомендації з використання вбудованого відлагоджувача Microsoft Visual C++.

Ті ж самі дії допоможуть Вам зрозуміти, де саме трапляються помилки, і як їх усувати. У цьому додатку ми розглядатимемо тільки однофайлові консольні програми, але той же підхід з невеликими варіаціями застосовується і для багатофайлових програм.

Почніть із звичайного компонування своєї програми. Спробуйте виправити всі помилки компілятора і компонувальника. Переконайтеся в тому, що код програми доступний у вікні редагування.

Покрокове трасування. Щоб запустити відлагоджувач, просто натисніть **F10**. Ви побачите жовту стрілку поряд з вікном документа, що вказує на рядок з відкритою фігурною дужкою в main(). Якщо Ви хочете починати трасування не з початку, то встановіть курсор на потрібний рядок. Потім з меню **Debug** (**Відлагодження**), яке в цьому режимі замінює меню **Build** (**Компонування**), виберіть **Start Debug** (**Почати відлагодження**), а після цього **Run to Cursor** (**Виконати до курсора**). Жовта стрілка тепер з'явиться слідом за вибраним виразом.

Натисніть клавішу **F10**. Це приведе до того, що відлагоджувач перейде на наступну команду, яку можна виконувати. Відповідно, туди ж зрушиться стрілка. Кожне натиснення **F10** означає крок трасування, тобто перехід до наступного виразу. Якщо Ви проганяєте цикл, то побачите, як стрілка доходить до останнього виразу, а потім перескакує знову на початок циклу.

Перегляд значень змінних. Існує можливість перегляду значень змінних у процесі покрокового трасування. Клікніть на закладці **Locals** (**Локальні**) в лівому нижньому кутку екрану, щоб побачити локальні змінні. Закладка **Auto** (**Авто**) покаже вибірку змінних, зроблену компілятором.

Якщо потрібно створити власний набір змінних, що проглядаються, внесіть їх у вікно перегляду, розташоване в правому нижньому кутку екрану. Щоб зробити це, клікніть правою кнопкою миші на імені змінної в початковому коді програми. Із спливаючого меню, що з'явилося, виберіть пункт **QuickView** (**Швидкий перегляд**). У діалоговому вікні клікніть на **Add watch** (**Додати**) для підтвердження

свого наміру. Ім'я змінної та її поточне значення з'явиться у вікні перегляду. Якщо змінна поки що недоступна (наприклад, ще не визначена у програмі), вікно перегляду видасть повідомлення про помилку замість значення поряд з іменем змінної.

Покрокове трасування функцій. Якщо у програмі використовуються функції, є чудова можливість здійснити покрокове трасування кожного виразу функції. Це робиться за допомогою клавіші **F11**. На відміну від режиму трасування по клавіші **F10**, який розглядає кожну зустрінуту функцію як один вираз, такий режим дає змогу "увійти" всередину функції та виявити можливі помилки в ній. Якщо Ви примусите по клавіші **F11** відбуватися бібліотечні функції, наприклад **cout <<**, то зможете милуватиметеся початковим кодом бібліотеки. Це процес довгий, тому радимо користуватися цією можливістю, тільки якщо Вам дійсно важливо знати, що відбувається усередині рутинних функцій. Втім, можна в міру потреби чергувати натиснення клавіш **F10** і **F11**, залежно від необхідності виявлення помилок у внутрішніх функціях.

Використання точок зупинки роботи коду програми. Точки зупинки, як випливає з їх назви, дають змогу тимчасово переривати роботу програми у вказаних Вами місцях. Коли їх потрібно використовувати? Ми вже показали, як можна примусити відлагоджувач проганяти програму тільки до курсора. Але бувають випадки, коли необхідно мати декілька таких точок. Наприклад, можна зупинити програму в кінці умови **if**, а також після відповідної **else**. Точки зупинки вирішують це завдання, оскільки їх можна ставити в необмеженій кількості. У них є ще і розширені функції, які ми тут розглядати не будемо.

Розглянемо, як вставити точки зупинки в код програми. По-перше, встановіть курсор на той рядок, у якому програма при трасуванні повинна зупинитися. Натисніть праву кнопку миші і виберіть з меню **Insert/Delete Breakpoint** (**Вставити/Видалити точку зупинки**). Напроти цього рядка коду програми з'явиться червоний кружок. Тепер, навіть якщо Ви просто запустите програму на виконання (вибравши, наприклад, **Debug/Go** (**Відлагодження/Запуск**)), програма зупиниться на цьому рядку. Можна на цьому часовому зрізі перевірити значення змінних, провести покрокове трасування коду програми і т.д. Загалом, можна зробити все, що Вам заманеться, наприклад, взагалі вийти з програми.

Щоб видалити точку зупинки, натисніть праву кнопку миші і виберіть пункт **Видалити з меню**, що з'явилося. Незважаючи на те, що є ще багато інших функцій відлагоджувача, проте ми тут показали тільки деякі основні з них для того, щоби Вам було з чого почати роботу.

Додаток Д. .NET-РОЗШИРЕНИЯ ДЛЯ С++

Розроблена компанією Microsoft інтегрована оболонка .NET Framework визначає середовище, яке призначене для підтримки розробки і виконання сильно розподілених застосувань, заснованих на використанні компонентних об'єктів. Вона дає змогу "мирно співіснувати" різним мовам програмування і забезпечує безпеку, переносність програм і загальну модель програмування для платформи Windows. Незважаючи на відносну новизну оболонки .NET Framework, ймовірно, в найближчому майбутньому у цьому середовищі працюватимуть багато С++-програмісти.

Інтегрована оболонка .NET Framework надає кероване середовище, яке стежить за виконанням програми. Програма, призначена для приміщення в оболонку .NET Framework, не компілюється з метою отримання об'єктного коду програми. Натомість вона перекладається проміжною мовою Microsoft Intermediate Language (MSIL)[1], а потім виконується під управлянням універсального засобу CLR (Common Language Runtime)[2]. Кероване виконання це механізм, який підтримує ключові переваги, пропоновані оболонкою .NET Framework.

Щоб скористатися перевагами керованого виконання, необхідно застосовувати для С++-програм спеціальний набір нестандартних ключових слів і директив препроцесора, які були визначені розробниками компанії Microsoft. Важливо розуміти, що цей додатковий набір не включений у стандарт С++ (ANSI/ISO Standard C++). Тому програмний код, у якому використовуються ці ключові слова, не можна переносити в інші середовища виконання.

Опис оболонки .NET Framework і методів С++-програмування, необхідних для її використання, виходить за рамки цього навчального посібника. Проте тут наведено короткий огляд .NET-розширення мови програмування С++ заради тих програмістів, які працюють в .NET-середовищу.

---

[1] Для полегшення перекладу програм, написаних різними мовами у середовище .NET, у Microsoft розроблена проміжна мова – Microsoft Intermediate Language (MSIL). Аби відкомпілювати додаток .NET, компілятори беруть початковий код і створюють з нього MSIL-код. MSIL – це повноцінна мова, придатна для написання будь-яких додатків. Проте, як у випадку з мовою Асемблер, користувачу навряд чи доведеться цим займатися, окрім деяких особливих обставин. Кожна група розробників компілятора вирішує, якою мірою він підтримуватиме MSIL. Але якщо творці компіляторів захочуть, аби їх мова повноцінно взаємодіяла з іншими мовами, їм доведеться обмежити себе рамками, які визначаються специфікаціями CLS.

[2] Common Language Runtime (CLR) – "загальне середовище виконання мов" – це компонент пакету Microsoft .NET Framework, віртуальна машина, на якій виконуються всі мови платформи .NET Framework.

CLR транслюється початковий код в байт-код мовою IL, реалізація компіляції якого компанією Microsoft називається MSIL, а також надає MSIL-програмам (а отже, і програмам, написаних мовами високого рівня, що підтримують .NET Framework) доступ до бібліотеки класів .NET Framework, або так званою .NET FCL (англ. Framework Class Library).

Середовище CLR є реалізацією специфікації CLI (англ. Common Language Infrastructure), специфікації загальномовної інфраструктури, компанією Microsoft.

Віртуальна машина CLR дає змогу програмістам забути про багато деталей про конкретний процесор, на якому виконуватиметься програма. CLR також забезпечує такі важливі служби як: управління пам'яттю; управління потоками; оброблення винятків; збирання сміття; безпека виконання.

Д.1. Ключові слова .NET-середовища

Для підтримки .NET-середовища керованого виконання C++-програм Microsoft вводить в мову C++ такі ключові слова:

__abstract	__box	**__delegate**
__event	**__finally**	**__gc**
__identifier	**__interface**	**__nogc**
__pin	**__property**	**__sealed**
__try_cast	**__typeof**	**__value**

Короткий опис кожного з цих ключових слів наведено нижче.

Ключове слово __abstract використовується у поєднанні із словом **__gc** під час визначення абстрактного керованого класу. Об'єкт **__abstract**-класса створити не можна. Для класу, визначеного з використанням ключового слова **__abstract**, необов'язково внесення в нього чисте віртуальній функції.

Ключове слово __box укладає у спеціальну оболонку значення усередині об'єкта. Таке "упакування" дає змогу використовувати тип цього значення у коді, який вимагає, щоб даний об'єкт був виведений з класу System::Object, базового класу для всіх .NET-об'єктів.

Ключове слово __delegate визначає об'єкт-делегат, який інкапсулює покажчик на функцію усередині керованого класу (тобто класу, модифікованого ключовим словом**__gc**).

Ключове слово __event визначає функцію, яка представляє деяку подію. Для такої функції задається тільки прототип.

Ключове слово __finally це доповнення до стандартного C++-механізму оброблення виняткових ситуацій. Воно використовують для визначення блоку коду програми, який повинен виконуватися після виходу з блоків **try/catch**. До того ж, не має значення, які умови приводять до завершення **try/caten**-блока. Блок**__finally** повинен бути виконаний у будь-якому випадку.

Ключове слово __gc визначає керований клас. Позначення "**gc**" є скорочення від словосполучення "**g**arbage **c**ollection" (тобто "збирання сміття") і означає, що об'єкти цього класу автоматично піддаються процесу утилізації пам'яті, що звільняється під час роботи коду програми, коли вони більше не потрібні. В об'єкті відпадає необхідність у разі, коли на нього не існує жодного посилання. Об'єкти **__gc**-класу повинні створюватися за допомогою оператора **new**. Масиви, покажчики і інтерфейси також можна визначати з використанням ключового слова**__gc**.

Ключове слово __identifier дає змогу будь-якому іншому ключовому слову мови програмування C++ використовуватися як ідентифікатор. Ця можливість не призначена для широкого застосування і введена для вирішення спеціальних завдань.

Ключове слово __interface визначає клас, який повинен діяти як інтерфейс. У будь-якому інтерфейсі жодна з функцій не повинна містити тіло, тоб-

то всі функції інтерфейсу є опосередковано заданими суто віртуальними функціями. Таким чином, інтерфейс є абстрактний клас, у якому не реалізована жодна з його функцій.

Ключове слово __nogc визначає некерований клас. Оскільки такий (некерований) тип класу створюється за замовчуванням, **Ключове слово __nogc** використовується рідко.

Ключове слово __pin використовують для визначення покажчика, який фіксує місцезнаходження у пам'яті об'єкта, на який він вказує. Таким чином, "закріплений" об'єкт не переміщатиметься у пам'яті у процесі збірки сміття. Як наслідок, складальник сміття не у змозі зробити недійсним покажчик, модифікований за допомогою ключового слова **__pin**.

Ключове слово __property визначає властивість, що є функцією-членом, яка дає змогу встановити або набути значення певної змінної (члена даних класу). Властивості надають зручний засіб керування доступом до закритих (**private**) або захищених (**protected**) даних.

Ключове слово __sealed оберігає клас, який модифікується ним, від успадкування іншими класами. Це слово можна також використовувати для інформування про те, що віртуальна функція не може бути перевизначена.

Ключове слово __try_cast використовується для перетворення типу виразу. Якщо зроблена спроба виявиться невдалою, згенерує виняток типу System::InvalidCastException.

Ключове слово __typeof дає змогу отримати об'єкт, який інкапсулює інформацію про даного типу. Цей об'єкт є примірником класу System::Type.

Ключове слово __value визначає клас, який є позначенням типу. Будь-яке позначення типу містить власні значення. І цим тип **__value** відрізняється від типу **__gc**, який повинен виділяти пам'ять для об'єкта за допомогою оператора **new**. Позначення типу не представляють інтерес для "складальника сміття".

Д.2. Розширення препроцесора

У мові програмування C++ передбачено використання препроцесора. Одним з таких розширень є конструкція **#define**, іншим – можливість включати під час компіляції вміст інших файлів.

Препроцесор виконує макропідстановку, умовну компіляцію, включення в програму іменованих файлів. Рядки, які починаються знаком # (перед ним можливі символи-роздільники), установлюють зв'язок із препроцесором. Їх синтаксис не залежить від іншої частини мови, вони можуть з'являтися де завгодно і впливати (незалежно від області дії) аж до кінця трансльованої одиниці. Межі рядків беруться до уваги; кожен рядок аналізується окремо (також є можливість "склеювати" рядки). Лексемами для препроцесора є всі лексеми мови і послідовності символів, які задають імена файлів, як, наприк-

лад, у директиві #include. Окрім цього, будь-який символ, не визначений яким-небудь іншим способом, сприймається як лексема. Вплив символів-роздільників, які відрізняються від пробілів і горизонтальної табуляції, усередині рядків препроцесора не визначено.

Саме препроцесування відбувається в декількох логічно послідовних фазах. В окремих реалізаціях деякі фази об'єднані.

1. Тризначні послідовності заміняються їх еквівалентами. Між рядками вставляються символи нового рядка, якщо того потребує операційна система.
2. Викидаються пари символів, які включають символ (\) з наступним символом нового рядка, що здійснює "склеювання" рядків.
3. Програма розбивається на лексеми, розділені символами-роздільниками. Коментарі замінюються одиничними пробілами. Потім виконуються директиви препроцесора і макропідстановки.
4. ESC-послідовності в символьних константах і рядкових літералах заміняються на символи, які вони позначають. Сусідні рядкові літерали конкатенуються.
5. Результат транслюється. Далі встановлюються зв'язки з іншими програмами і бібліотеками за допомогою збирання необхідних програм та даних і з'єднання посилань на зовнішні функції та об'єкти з їх визначеннями.

Для підтримки .NET-середовища компанія Microsoft визначає директиву препроцесора **using**, яка використовують для імпортування метаданих у програму. Метадані містять інформацію про тип і членів класу у формі, яка не залежить від конкретної мови програмування. Таким чином, метадані забезпечують підтримку змішаного використання мов програмування. Всі керовані С++-програми повинні імпортувати бібліотеку <mscorlib.dll>, яка містить необхідні метадані для оболонки .NET Framework.

Компанія Microsoft визначає дві pragma-настанови (використовувані з директивою препроцесора #pragma), які мають відношення до оболонки .NET Framework. Перша (managed) визначає керований код. Друга (unmanaged) визначає некерований (власний, тобто притаманний даному середовищу) код. Ці настанови можуть бути використані усередині програми для селективного створення керованого і некерованого коду програми.

Використання атрибуту attribute. Компанія Microsoft визначає атрибут **attribute**, який використовують для оголошення іншого атрибуту.

Компілювання керованих С++-програм. На момент написання цього навчального посібника єдиний доступний компілятор, який міг обробляти програми, що орієнтуються на роботу у середовищі .NET Framework, постачався компанією Microsoft (Visual Studio.NET). Щоб скомпілювати керовану програму, необхідно використовувати команду /clr, яка передасть Вашу програму "до рук" універсального засобу Common Language Runtime.

ЛІТЕРАТУРА

1. **Александреску А.** Современное проектирование на C++ / А. Александреску. – Сер.: C++ In-Depth. 3. – М. : Изд. дом "Вильямс", 2002. – 336 с.

2. **Аммерааль Л.** STL для программистов на C++ / Л. Аммерааль. – М. : Изд-во ДМК, 1999. – 240 с.

3. **Архангельский А.Я.** Программирование в C++ Builder 6 / А.Я. Архангельский. – М. : Изд-во "Бином", 2004. – 1152 с.

4. **Бронштейн И.Н.** Справочник по математике для инженеров и учащихся втузов / И.Н. Бронштейн, К.А. Семендяев. – Изд. 13-е, испр. – М. : Изд-во "Наука", Гл. ред. физ.-мат. лит., 1986. – 544 с.

5. **Буч Г.** Объектно-ориентированный анализ и проектирование с примерами на C++ / Г. Буч. – М. : Изд-во "Бином", 1998. – 560 с.

6. **Влиссидес Дж.** Применение шаблонов проектирования. Дополнительные штрихи / Дж. Влиссидес. – М. : Изд. дом "Вильямс", 2003. – 144 с.

7. **Гамма Э.** Приемы объектно-ориентированного программирования. Патерны проектирования / Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. – СПб. : Изд-во "Питер", 2004. – 366 с.

8. **Глинський Я.М.** C++ і C++ Builder / Я.М. Глинський, В.Є. Анохін, В.А. Ряжська. – Львів : Вид-во "Деол", СПД Глинський, 2003. – 192 с.

9. **Грицюк Ю.І.** Програмування мовою C++ : навч. посібн. / Ю.І. Грицюк, Т.Є. Рак. – Львів : Вид-во Львівського ДУ БЖД, 2011. – 292 с.

10. **Кениг Э.** Эффективное программирование на C++. Серия C++ In-Depth. tType. 2 / Э. Кениг, Б. Му. – М. : Изд. дом "Вильямс", 2002. – 384 с.

11. **Ласло М.** Вычислительная геометрия и компьютерная графика на C++ / М. Ласло. – М. : Изд-во "Бином", 1997. – 304 с.

12. **Лафоре, Роберт.** Объектно-ориентированное программирование в C++. Классика Computer Science / Роберт Лафоре : пер. с англ. – Изд. 4-е. – СПб. : Изд-во "Питер", 2005. – 924 с.

13. **Либерти Д.** Освой самостоятельно C++ за 21 день / Д. Либерти. – М. : Изд. дом "Вильямс", 2000. – 816 с.

14. **Липпман С.** Язык программирования C++. Вводный курс / С. Липпман, Ж. Лажойе : пер. с англ. – Изд. 3-е. – СПб.-М. : Изд-во "Невский диалект – ДМК Пресс", 2004. – 1104 с.

15. **Павловская Т.А.** Программирование на языке высокого уровня : учебник / Т.А. Павловская. – СПб. : Изд-во "Питер", 2005. – 461 с.

16. **Павловская Т.А.** C++. Объектно-ориентированное программирование : практикум / Т.А. Павловская, Ю.А. Щупак. – СПб. : Изд-во "Питер", 2005. – 265 с.

17. **Павловская Т.** C/C++. Программирование на языке высокого уровня / Т.А. Павловская. – СПб. : Изд-во "Питер", 2001. – 460 с.

18. Павловская Т. С/С++. Структурное программирование : практикум / Т.А. Павловская, Ю.А. Щупак. – СПб. : Изд-во "Питер", 2005. – 240 с.

19. Павловская Т.А. С++. Объектно-ориентированное программирование : практикум / Т.А. Павловская, Ю.А. Щупак. – СПб. : Изд-во "Питер", 2005. – 265 с.

20. Прата, Стивен. Язык программирования С++. Лекции и упражнения : учебник : пер. с англ. / Стивен Прата. – СПб. : ООО "ДиаСофтБП", 2005. – 1104 с.

21. Саттер Г. Решение сложных задач на С++. – Сер.: C++ In-Depth / Г. Саттер. – Т. 4. – М. : Изд. дом "Вильямс", 2002. – 400 с.

22. Синтес, Антони. Освой самостоятельно объектно-ориентированное программирование за 21 день : пер. с англ. / Антони Синтес. – М. : Изд. дом "Вильямс", 2002. – 672 с.

23. Справочник по элементарной математике: геометрия, тригонометрия, векторная алгебра / под ред. члена-корр. АН УССР П.Ф. Фильчакова. – К. : Вид-во "Наук. думка", 1966. – 444 с.

24. Страуструп Б. Язык программирования С++ / Б. Страуструп. – СПб. : Изд-во "Бином", 1999. – 991 с.

25. Халперн, Пабло. Стандартная библиотека С++ на примерах : пер. с англ. / Пабло Халперн. – М. : Изд. дом "Вильямс", 2001. – 336 с.

26. Шаллоуэй А. Шаблоны проектирования. Новый подход к объектно-ориентированному анализу и проектированию / А. Шаллоуэй, Д. Тротт. – М. : Изд. дом "Вильямс", 2002. – 288 с.

27. Шилдт, Герберт. Искусство программирования на С++ : пер. с англ. / Герберт Шилдт. – СПб. : Изд-во БХВ-Петербург, 2005. – 496 с.

28. Шилдт, Герберт. Полный справочник по С++ : пер. с англ. / Герберт Шилдт. – Изд. 4-ое. – М. : Изд. дом "Вильямс", 2010. – 800 с.

29. Шилдт, Герберт. С++: Базовый курс / Герберт Шилдт : пер. с англ. – Изд. 3-е. – М. : Изд. дом "Вильямс", 2005. – 624 с.

30. Шилдт, Герберт. Самоучитель С++ / Герберт Шилдт : пер. с англ. – Изд. 3-е. – СПб. : Изд-во БХВ-Петербург, 2005. – 688 с.

31. Штерн В. Основы С++. Методы программной инженерии / В. Штерн. – М. : Изд-во "Лори", 2003. – 860 с.

32. Элджер Д. С++ библиотека программиста / Д. Элджер. – СПб. : Изд-во "Питер", 2000. – 320 с.

33. Грейди Буч. Язык UML. Руководство пользователя = The Unified Modeling Language user guide / Буч Грейди, Рамбо Джеймс, Айвар Джекобсон. – Изд. 2-ое. – М.-СПб. : Изд-во "ДМК Пресс", Питер, 2004. – 432 с.

34. Буч Г. UML. Классика CS / Г. Буч, А. Якобсон, Дж. Рамбо : пер. с англ. / под общ. ред. проф. С. Орлова. – Изд. 2-ое. – СПб. : Изд-во "Питер", 2006. – 736 с.

Навчальне видання

ГРИЦЮК Юрій Іванович,
РАК Тарас Євгенович

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ МОВОЮ С++

Навчальний посібник

Літературний редактор *В.В. Дудок*

Редактор *Г.М. Падик*
Технічний редактор *О.В. Хлевной*
Авторські комп'ютерний набір та верстка

Підписано до друку 25.05.2011. Формат 60×84/16.
Папір офсетний. Гарнітура *Times*. Друк на різографі.
Ум. др. арк. 23,48. Ум. фарбо-відб. 23,72.
Наклад 350 прим. Зам. № *29/2011*

Видавництво ЛДУ БЖД, Україна, 79007, м. Львів, вул. Клепарівська, 35
Тел./факс: (032) 233-14-77; E-mail: mail@ubgd.lviv.ua; Wed-адреса: http://www.ubgd.lviv.ua
Свідоцтво про внесення суб'єкта видавничої справи до державного реєстру видавців, виготовників і розповсюджувачів видавничої продукції, серія ДК, № 368 від 20.03.2001 р.

Грицюк Ю.І., Рак Т.Є.

Г 85 Об'єктно-орієнтоване програмування мовою С++ : навчальний посібник. – Львів : Вид-во Львівського ДУ БЖД, 2011. – 404 с. – Статистика: іл. 18, табл. 12, бібліогр. 34.

ISBN 978-966-3466-86-3

Розглядаються основні особливості розроблення об'єктно-орієнтованих програм мовою С++. На конкретних прикладах вивчаються класи та робота з ними, перевизначення операторів і успадкування в класах, віртуальні функції та поліморфізм, шаблони в класах і оброблення виняткових ситуацій, С++-система введення-виведення, динамічна ідентифікація типів і оператори приведення типу, простір імен і інші ефективні програмні засоби, введення в стандартну бібліотеку шаблонів і особливості роботи препроцесора С++. На завершення подано матеріал, який стосується формалізації процесу розроблення об'єктно-орієнтованого програмного забезпечення.

Викладений матеріал базується на стандарті ANSI/ISO мови програмування С++, а також зазначено нововведення, які затверджені в стандарті ISO/IEC 14882:2003. Наведено важливу для практичного використання та програмування інформацію про додаткові можливості компілятора, середовища та бібліотек Borland C/C++.

Видання призначено для курсантів і студентів, які вивчають програмування в рамках різних навчальних дисциплін, а також для всіх, хто бажає самостійно опанувати технологію програмування мовою С++.