В.В. БУБЛИК

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ

Підручник для студентів, які навчаються за напрямами «Комп'ютерні науки», «Комп'ютерна інженерія», «Програмна інженерія», «Системна інженерія», «Інформатика», «Прикладна математика»

Київ
ІТ-книга
2015

УДК 004.432.2
ББК 32.973.26-018.1
Б90

Рецензенти: А.В. Анісімов, член-кореспондент НАНУ, доктор фізико-математичних наук, професор, завідувач кафедри математичної інформатики, декан факультету кібернетики Київського національного університету імені Тараса Шевченка

А.Ю. Дорошенко, доктор фізико-математичних наук, професор кафедри автоматики та управління в технічних системах факультету інформатики та обчислювальної техніки Національного технічного університету України «Київський політехнічний інститут»

Рекомендовано Вченою радою Національного університету «Києво-Могилянська академія» (протокол № 18 від 23.10.2014)

Бублик В.В.

Б90 Об'єктно-орієнтоване програмування: [Підручник] / В.В. Бублик. – К.: ІТ-книга, 2015. – 624 с.: іл.

ISBN 978-966-97182-1-1

Підручник призначено для всіх, хто прагне оволодіти технікою об'єктно-орієнтованого програмування на основі мови програмування C++. Книжка навчить не просто програмувати, а розробляти безпечні та гнучкі програми, придатні до нових застосувань. Основна увага приділяється побудові обґрунтованих ієрархій об'єктів і класів, складанню абстрактних інтерфейсів та універсальних сигнатур методів. Видання допоможе програмістам-початківцям зрости до рівня професіоналів, а також стане чудовою базою для сучасного курсу об'єктно-орієнтованого програмування у профільних вищих навчальних закладах.

ББК 32.973.26-018.1

ISBN 978-966-97182-1-1

© В.В. Бублик, 2015
© ІТ-книга, 2015

Дуже радий, що цей підручник нарешті побачив світ. Неважливо, якою мовою об’єктно-орієнтованого програмування Ви користуєтеся в даний конкретний момент часу, базові проблеми залишаються однаковими. Якщо Ви хочете досягти у своїй професії чогось більшого, ніж вміння нашвидкуруч склепати простий веб-сайт на основі готової CMS – ця книга повинна стати Вам в нагоді. Успіхів!

Ростислав Сліпецький,
випускник НаУКМА 2009 року,
Microsoft

Досвід і глибина розуміння предмету Володимиром Васильовичем завжди вражали. Його прагматизм, коментарі і навіть погляд змушували нас критично оцінювати кожне наше слово, кожну думку. Такого ґрунтовного підходу до викладання курсів я не зустрічав ні до, ні після. Заняття змушували бути уважним до найменших дрібниць, ретельно продумувати всі зв’язки між компонентами. Ми повинні були могти пояснити кожну деталь, кожен символ в наших реалізаціях. Всі потенційні можливості використати наші класи не за призначенням повинні були бути завбаченими і унеможливленими. Таким чином наші програми виходили добре захищеними й стабільними.

Кожен знайшов в курсах Володимира Васильовича щось для себе. Не тільки предмет, але і метод викладання, поєднання теорії з жорсткою практикою і модель поведінки викладача викликали лише захоплення. З точки зору багатьох моїх однокурсників, ці курси стали фундаментом в нашому професійному розвитку, а ВВ (Володимир Васильович) і досі залишається найобговорюванішим викладачем серед колишніх студентів.

Роман Бойко,
випускник НаУКМА 2011 року,
Програміст в TripAdvisor,
Дублін, Ірландія

Бублик – це український Александреску.

Дмитро Красіков,
випускник НаУКМА 2009 року,
Google

Мені відомі випадки, коли старанні студенти Володимира Васильовича, які по краплинках всотували всю цінну інформацію на його парах, вже на своїй першій роботі ставали «законодавцями стилю програмування», запроваджували дисципліну розробки ПЗ у цілій фірмі, бодай і невеликій.

Цей підручник не навчить вас програмувати, якщо до цього програмування було для вас чимось захмарним. Цей підручник навчить вас добре програмувати, якщо ви вже трохи розумієте, про що йдеться. Він навчить вас прискіпливості та критичності, сформує ваш професійний світогляд.

Анна Дьоміна,
випускниця НаУКМА 2011 року,
software engineer в EPAM Systems,
аспірантка НаУКМА

Всі курси Володимира Васильовича були корисними для мене. В першу чергу вони вчать мислити більш структуровано і системно підходити до вирішення завдань. Хоч програмування і не є головним обов'язком для мене, але я постійно користуюсь знаннями з курсів при вирішенні критичних для моєї роботи завдань, підготовці експериментів і симуляцій, обробці і аналізі даних. Крім того, час від часу треба створювати спеціалізовані програми для експериментів або виправляти помилки в чужих розробках, і знання, отримані в Могилянці, мені в цьому дуже допомагають.

Мирослав Бачинський,
випускник бакалаврату НаУКМА 2009 року,
магістр комп'ютерних наук (Університет Саарланду),
докторант у Max Planck Institute for Informatics

Цей підручник стане у нагоді як початківцям, так і читачам із досвідом використання об'єктно-орієнтованої парадигми програмування. Книга висвітлює фундаментальні аспекти та дає свіжий погляд на усталені засади у програмуванні. Приклади, які автор використовує, дозволять легко зрозуміти базові поняття та оволодіти ключовими принципами об'єктно-орієнтованого моделювання. Рекомендую цей підручник до Вашої уваги.

Доктор Артем Полив'яний,
випускник НаУКМА 2004 року,
Queensland University of Technology, Brisbane, Australia

Зміст

Передмова 9
Веб-підтримка курсу 12
1. Парадигми програмування 13
1.1. Парадигми і мови програмування 13
1.2. Процедурне програмування 17
1.3. Об'єктне (модульне) програмування 19
1.4. Об'єктно-орієнтовне програмування 21
1.5. Програмні середовища 24
Завдання до розділу 1 30
2. Базові поняття програмування 31
2.1. Фундаментальні типи даних і операції над ними 33
2.1.1. Розрядність фундаментальних типів даних 35
2.1.2. Операції над даними фундаментальних типів 37
2.2. Літерали та їхні типи 41
2.2.1. Числові літерали 42
2.2.2. Символьні літерали 43
2.3. Сталі та змінні 45
2.3.1. Визначення сталих і змінних 45
2.3.2. Присвоєння 46
2.3.3. Перетворення типів 50
2.4. Указники 51
2.4.1. Указник як засіб непрямого адресування 51
2.4.2. Указники та динамічна пам'ять 56
2.4.3. Указники констант 59
2.4.4. Типізація указників 62
2.4.5. Операції над указниками 67
2.5. Відсилки 68
2.6. Інструкції 71
2.6.1. Прості інструкції 71
2.6.2. Структури керування 72
2.7. Налагодження програм 77
2.7.1. Макрос суджень 77
2.7.2. Інструкція випробувань і блок захоплення 79
2.8. Масиви 81
2.8.1. Масиви та указники 82
2.8.2. Масиви символів і указники 89

2.9. Програмовані структури даних ... 92
2.9.1. Структури ... 92
2.9.2. Бітові поля ... 101
2.9.3. Об'єднання ... 102
Завдання до розділу 2 ... 103
3. Процедурно-орієнтоване програмування ... 107
3.1. Функції ... 107
3.1.1. Визначення й оголошення функцій ... 108
3.1.2. Відкриті та закриті функції ... 111
3.2. Механізми передавання параметрів ... 119
3.2.1. Передавання параметрів значенням ... 120
3.2.2. Параметри-указники ... 124
3.2.3. Параметри-відсилки ... 127
3.2.4. Передавання параметрів-масивів ... 131
3.2.5. Особливості передавання сталих параметрів ... 139
3.2.6. Параметри-указники другого рівня ... 150
3.3. Обчислення значення функції ... 162
3.3.1. Функції, що повертають значення ... 163
3.3.2. Функції, що повертають указники та відсилки ... 166
3.4. Статичний поліморфізм ... 172
3.4.1. Довизначення функцій ... 172
3.4.2. Параметри за замовчуванням ... 176
3.5. Програмування операцій ... 180
3.5.1. Програмування операцій уведення та виведення ... 181
3.5.2. Програмування арифметичних операцій ... 183
3.6. Узагальнені функції ... 185
3.6.1. Шаблони функцій ... 186
3.6.2. Особливості конкретизації шаблонів функцій ... 188
3.7. Рекурсія ... 191
3.7.1. Техніка рекурсивного програмування ... 192
3.7.2. Ефективна рекурсія ... 195
3.8. Функціональна полісемія ... 197
3.8.1. Указники на функції ... 198
3.8.2. Використання указників на функції як параметрів ... 201
3.8.3. Указники на функції для реалізації меню ... 204
Завдання до розділу 3 ... 206
4. Об'єктне програмування ... 210
4.1. Принцип інкапсуляції даних і дій ... 211
4.1.1. Основні поняття об'єктного програмування ... 211

4.1.2. Інкапсуляція указників на функції в структурах 213
4.1.3. Статичні указники на функції в структурах 216
4.2. Конструктори та деструктори 217
4.2.1. Функції створення та видалення екземплярів структур 218
4.2.2. Виклики конструктора та деструктора..................................220
4.2.3. Реалізація конструкторів 225
4.3. Права доступу, селектори та модифікатори..................................230
4.4. Класи та структури..238
4.5. Функції-члени класу та функції-утиліти класу.............................245
4.6. Копіювання об’єктів..249
4.6.1. Конструктор копіювання 249
4.6.2. Конструктор мультикопіювання..258
4.6.3. Оператор копіювального присвоєння.......................................260
4.7. Операції-члени класу й операції-утиліти класу...........................267
4.8. Властивості арифметичних і логічних операцій..........................277
4.9. Конвертори типів...298
4.10. Особливості статичних членів класу 319
Завдання до розділу 4 328
5. Методи об’єктного програмування 332
5.1. Порушення прав доступу...332
5.2. Типізація членів класів і структур 345
5.3. Асоціювання об’єктів...361
5.3.1. Композиція об’єктів 361
5.3.2. Залежності між об’єктами 367
5.3.3. Агрегати об’єктів..379
5.4. Примітивні контейнери..392
5.4.1. Клас масивів...392
5.4.2. Клас обмежених послідовностей 395
5.4.3. Клас необмежених послідовностей 403
5.4.4. Клас стеків на базі масиву 410
5.4.5. Клас черг на основі масиву 413
5.4.6. Клас двобічних черг на базі масиву...417
5.5. Інтелектуальні указники 418
5.5.1. Клас монолітних маніпуляторів...419
5.5.2. Клас гнучких маніпуляторів..428
5.5.3. Клас осібних маніпуляторів 431
5.5.4. Порівняння маніпуляторів з указниками 438
5.5.5. Клас вкладених маніпуляторів...440
5.5.6. Маніпулятори масивів 442

5.6. Функціональні об'єкти 445
5.6.1. Клас функторів 446
5.6.2. Застосування функторів 447
5.7. Асоціативні масиви 453
5.7.1. Клас асоціативних масивів 454
5.7.2. Прихована реалізація 458
5.8. Підрахунок відсилок 460
5.8.2. Відкладене копіювання 460
5.8.2. Метод проксі 468
5.8.3. Узгодження перетворень указників і відсилок проксі 474
Завдання до розділу 5 479
Навчальні проекти 481
6. Узагальнене програмування 482
6.1. Типи, що залежать від сталих параметрів 483
6.2. Параметризовані класи 495
6.3. Вкладені параметризовані типи 504
6.4. Поліпараметризовані класи 516
6.5. Параметризовані вкладені типи 523
6.6. Розповсюдження шаблонів 532
Завдання до розділу 6 542
Навчальні проекти 544
7. Ієрархічне програмування 545
7.1. Ієрархія об'єктів 546
7.2. Ієрархія класів: відкрите успадкування 552
7.3. Ієрархія класів: типізація об'єктів 562
7.4. Уточнення привілеїв доступу до членів базового класу 577
7.5. Захищене успадкування 587
7.6. Закрите успадкування 590
7.7. Комбінована ієрархія 595
7.8. Кратне успадкування 598
7.9. Успадкування інтерфейсу й успадкування реалізації; абстрактні класи 606
7.10. Архітектура ієрархій об'єктів і класів 615
Завдання до розділу 7 624
Навчальні проекти 626
Заключне слово 628
Список літератури 630
Глосарій 632

Передмова

Цей підручник виник як підсумок викладання автором об’єктно-орієнтованого програмування студентам-інформатикам Київського національного університету імені Тараса Шевченка і Національного університету «Києво-Могилянська академія» протягом останніх десяти років. Курс розрахований на 4–6 аудиторних годин на тиждень протягом року та значний обсяг самостійної роботи зі створення, реалізації та випробувань програмних проектів. Практика багаторічного спілкування зі студентами дала можливість сконцентруватися на складніших для розуміння конструкціях, підібрати прозоріші приклади.

Слухачі мали певний власний досвід програмування, здебільшого на Паскалі, а останнім часом мовою Java. Тому ми не зупинятимемося на традиційних проблемах початкового курсу програмування, а виходитимемо з того, що традиційні структури керування й структури даних високого рівня добре відомі, так само як техніка програмування рекурентних співвідношень, індуктивних функцій, дослідження інваріантів тощо. Читачеві мають бути відомі також засоби алгоритмічної декомпозиції, наприклад, на рівні розробки алгоритмів сортування й пошуку, обробки дерев, списків та інших структур даних. Для розуміння курсу цілком достатньо підготовки в обсязі книги Н. Вірта [1], на жаль, незаслужено забутої.

Головна мета курсу – оволодіння технікою об’єктно-орієнтованого програмування на основі мови програмування C++. Ця мова багатопарадигменна, тому її вивчення служить доброю базою для порівняльного аналізу парадигм імперативного програмування. Крім цього, мова C++ базується на мові програмування C. Залишаючись мовою програмування високого рівня, мова C дотримується концепції, вираженої лозунгом повернення до апаратури (*back to hardware*). Остання обставина дозволяє зосередити увагу на важливих питаннях нижчого рівня: роботою з машинними структурами даних (бітами, байтами, машинними словами), указниками, машинною арифметикою, програмованим розподілом пам’яті тощо. Отже, обрана нами мова органічно поєднує підходи нижчого й вищого рівнів і при цьому відповідає тенденції до поглиблення типізації. Саме програмування на C++ найповніше відповідає влучній характеристиці умінь програміста, даній академіком А.П.Єршовим: «Програміст повинен володіти здібністю першокласного математика до абстракції і логічного мислення у поєднанні з талантом Едісона споруджувати все, що

завгодно, з нуля і одиниці. Він повинен поєднувати акуратність бухгалтера з прозорливістю розвідника, фантазію автора детективних романів з тверезою практичністю економіста».

В центрі об'єктно-орієнтованої парадигми перебуває поняття типу, або класу, яке поєднує в собі чотири найважливіші властивості об'єктної моделі: абстракцію, інкапсуляцію, модульність та ієрархію. Кожна окрема властивість притаманна й іншим парадигмам. Так ми маємо абстракцію процедур, інкапсуляцію даних у структурі, ієрархію підструктур, модулі з процедур і даних в Object Pascal або Delphi. Для повноцінної реалізації об'єктно-орієнтованої парадигми залишається тільки доповнити ієрархію успадкуванням, відділити успадкування реалізацій від успадкувань типів, запровадити можливість налаштування поведінки за допомогою динамічного поліморфізму й ввести узагальнене поняття абстрактного класу або інтерфейсу, керуючись тезою Е.Дейкстри про те, що абстрактність не зводиться до невизначеності, а полягає в створенні нового абсолютно прозорого семантичного рівня.

Увага в курсі об'єктно-орієнтованого програмування концентрується не на складних алгоритмах – на них просто не вистачає часу – а на побудові обґрунтованих ієрархій, складанні абстрактних інтерфейсів та універсальних сигнатур методів. Ідеал об'єктно-орієнтованого програмування – «програми-пристосуванці» (*programming for reuse*), придатні до налагодження та функціонування за нових умов лише завдяки дописуванню, але не переписуванню коду. В літературі з програмування часом наводять приклад фрази, яку вважають повною протилежністю алгоритму: «Піди туди, не знаю куди, за тим, не знаю за чим». Але ж ця фраза – прекрасний приклад абстрактного інтерфейсу, не закованого в пута реалізації. Цей інтерфейс має справу з тими, які ходять і приносять, а куди ходять і що приносять – на цьому етапі неважливо. Це деталі, для з'ясування яких пізніше залишається достатньо простору. Наскільки було б гірше, якби довелося виправляти наслідки походу не туди, куди треба, з доставкою непотрібного або навіть шкідливого чи небезпечного. Сьогодні необхідно вчитися програмувати, заглядаючи у майбутнє, розробляти гнучкі програми, придатні до нових, можливо, навіть несподіваних застосувань. Образно кажучи, парадигму *«reuse»* можна сформулювати, як спробу відкласти на завтра все, чого можна не уточнювати сьогодні.

Підручник з об'єктно-орієнтованого програмування не замінить посібників з мови програмування C++. Він допоможе вивчати матеріал у правильній послідовності, виділити найважливіші з погляду розробки програмного забезпечення концепції, опрацювати змістовні приклади. Пе-

редбачено, що деталі синтаксису та семантики мовних конструкцій читачі вивчатимуть самостійно за широко доступною літературою, наприклад, за книгами авторів C [2] та C++ [3]. Мова програмування C++ знаходиться у постійному розвитку, ретроспективу якого можна прослідкувати за книгою Бьярне Страуструпа [4], а з її новітнім станом можна ознайомитися, вивчаючи [5]. Однак автор глибоко переконаний у перспективності навчання методам, а не мовам програмування. Кожна конструкція вивчається й аналізується з погляду доцільності та обґрунтованості її застосування в розробленні програм. Протягом усього курсу явна перевага віддається принципам структурованого програмування [6], а при написанні коду рекомендуємо керуватися міркуваннями, викладеними Стівом Макконнеллом [7].

Для подальшого вивчення можна рекомендувати поглиблений курс методів програмування з науковим семінаром, де увага зосереджуватиметься на рекомендаціях з ефективного програмування Скотта Мейерса [8, 9] , задачах і стандартах програмування Герба Саттера й Андрея Александреску [10, 11, 12], стратегіях проектування Александреску [13], взірцях проектування (*design pattern*) «банди чотирьох» [14] та особливостях нового стандарту C++ [5]. Дуже корисно організувати практикум групової розробки програм для практичного засвоєння набутих знань (*capstone project* в термінології відомих рекомендацій ACM до змісту навчальних програм з інформатики [15]).

Серйозною виявилася проблема української термінології. У програмуванні вона, на жаль, ще не склалася остаточно. Доступні книги російськомовні, термінологія російською мовою загалом склалася, хоча теж не вільна від критики. Деякі терміни, як наприклад, *framework, toolkit, design pattern* ще чекають на влучні переклади, інші перекладають дослівно навіть у тих випадках, коли можна вживати більш звичні слова, наприклад, *довизначення* або *розширення* замість «перегрузки функций». Значну допомогу в перекладі термінів надають англо-українські та російсько-українські словники з інформатики та обчислювальної техніки, зокрема підготовлені в Національному університеті ім. Тараса Шевченка [28], [29]. Але самі вони ще не встигли остаточно визначити мовну норму, а радше розпочали експеримент, відтак проблема термінології ще чекає на своє вирішення. Підбираючи власний варіант перекладу терміну, я керувався здоровим глуздом, власним досвідом та академічними виданнями словників української мови.

Робота над курсами програмування і підручником, зокрема, завдячує постійній підтримці колег і, головне, інтересу і активності студентів, те-

перішніх і колищніх, частина яких вже самі стали колегами автора. Потрібно згадати професора М.М.Глибовця, завдяки чиїй наполегливості цей підручник побачить світ, Ігоря Завадського і Олександра Левченка за їх активну співпрацю над рукописом, Богдана Троценка, Антона Шабінського, Володимира Ляшка, Анну Дьоміну, Назима Сітманбетова, Данила Фітеля, Андрія Давиденка, Андрія Чайку, Василя Куцика, Дмитра Зважія та багатьох інших за допомогу, стимулюючі питання, активність на форумах курсу, і, головне, за підтримку в автора впевненості у потрібності розпочатої справи.

Особливої вдячності заслуговує компанія Інфопульс за фінансування видання підручника, а також постійну допомогу факультету інформатики Києво-Могилянської академії.

Автор буде глибоко вдячний кожному, хто висловить свої зауваження до підручника, їх можна надіслати на адресу **boublik@ukma.kiev.ua**.

Веб-підтримка курсу

http://distedu.ukma.kiev.ua – портал електронного навчання, що супроводжує викладання курсу в Києво-Могилянській академії. Крім навчальних матеріалів, тут містяться вправи та задачі для самостійного розв'язування, завдання для програмних проектів, а також форуми для обговорення проблемних питань. Портал надає гостьовий доступ всім зацікавленим в ознайомленні з практичною частиною електронного курсу. Курс розміщено в розділі «другий рік навчання».

http://oop.in.ua – портал з об'єктно-орієнтованого програмування, створений силами студентів-ентузіастів факультету інформатики Києво-Могилянської академії. Портал супроводжує роботу гуртка любителів програмування, містить презентації студентських доповідей, добірки цікавих матеріалів, розв'язання задач та багато іншого.

http://itknyga.com.ua – сайт видавництва. Тут ви можете замовити книжки, завантажити презентації для лекторів, інтерактивні навчальні модулі до окремих тем курсу, а також отримати доступ до онлайнового навчального середовища курсу для вашого закладу.

1. Парадигми програмування

Слово *парадигма* грецького походження і означає стиль міркування, спосіб дій або набір концепцій у певній галузі знань. В останні десятиліття, після того, як американський вчений Роберт Флойд під час вручення йому Тьюрінгової медалі ACM 1978 року виступив з лекцією «Парадигми програмування» [16], термін «парадигми» став часто вживаним у програмуванні, а тому варто зупинитися на ньому окремо.

1.1. Парадигми і мови програмування

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

Інша програма могла б виглядати так: «Додайте до одиниці три, добудьте із суми квадратний корінь, відніміть його від мінус одиниці. Це буде перший корінь. Потім додайте його до мінус одиниці. Це буде другий корінь». Характерну особливість цієї програми можна визначити як імперативність: **«спочатку роби одне, а потім інше»**. Вона є прикладом *імперативної* (наказової) парадигми програмування. Зазвичай декларативна парадигма потребує досконалішого виконавця, тоді як застосування імперативної парадигми покладає більше відповідальності на програміста, адже він повинен дати більш детальний рецепт розв'язування задачі. Декларативне програмування – це щось на зразок вживання скатертини-самобранки, яка сама приготує за вас обід, тоді як імперативне потребує вправності та майстерності Попелюшки у виконанні забаганок вередливої мачухи.

Поділ на **«що?»** і **«як?»** у програмуванні відображає також природний перебіг процесу розв'язування задач. Спочатку визначаємося з тим, що треба зробити, проектуємо інтерфейси – виникає щось на зразок декларативної програми. Потім з'ясовуємо наявність виконавця, здатного цю програму зрозуміти і, можливо, виконати. Якщо його підібрано, оцінюємо характеристики виконання ним програми та порівнюємо їх на відповідність заданим вимогам. Якщо придатного виконавця немає, беремо найбільш пі-

дхожого з наявних і підсилюємо його можливості додатковими імперативними засобами, необхідними для розв'язання поставленого завдання.

Цікаво, що як перша механічна програмована машина Чарльза Бебіджа в XIX сторіччі, так і сучасні електронні обчислювальні машини зобов'язані своєю появою соціальному запиту на підхожих виконавців. У позаминулому столітті це було зумовлено проблемою перерахунку таблиць тригонометричних функцій у зв'язку із введенням під час Великої французької революції метричної системи вимірювання кутів, що надихнуло Бебіджа на ідею механічних обчислень, які виконуватимуться за наперед складеною програмою. У роки другої світової війни причиною стала постійна загроза бомбардувань та обстрілів Лондона, що потребувало швидкого автоматичного виконання розрахунків траєкторій та дешифрування переговорів німецьких пілотів.

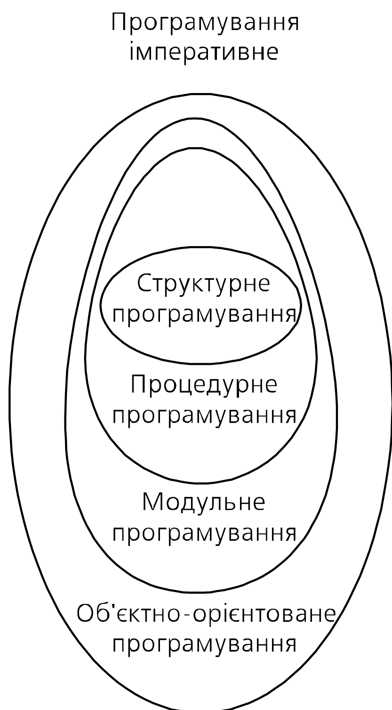


Рис. 1.1. Парадигми імперативного програмування

Всередині кожної з двох основних парадигм – імперативної і декларативної – виділяють парадигми менші. В цьому підручнику йтиметься про наказові парадигми – саме вони становлять основу сучасних виробничих технологій розробки програмних систем, – але ми не забуватимемо й про існування інших. Декларативні парадигми дають нам змогу точніше описувати властивості наказових програм. Зауважимо, що сам професор Джон Бекус – винахідник першої мови наказового програмування Фортран – успішно працював над пошуком нових парадигм, зокрема декларативної парадигми функціонального програмування, що також викладено у його Тьюрінговій лекції 1977 року [17]. Одночасне застосування кількох парадигм, або *мультипарадигменність*, дає можливість подивитися на задачу під різними кутами зору, завдяки чому поєднуються різні підходи і збільшуються шанси на відшукання кращого розв'язку.

Підкреслимо, що парадигми програмування – це моделі, які відтворюють спосіб мислення розробника програми. Мова програмування може підтримувати або не підтримувати ту чи іншу парадигму. В першому випадку застосування парадигми стає зручним, тобто простим, безпечним і ефективним, в іншому – складним і ненадійним. Ми розглянемо три основних наказових парадигми – *процедурне*, *об'єктне* (модульне) і *об'єктно-орієнтоване* (ієрархічне) програмування (рис. 1.1) – та одну допоміжну – *узагальнене* програмування.

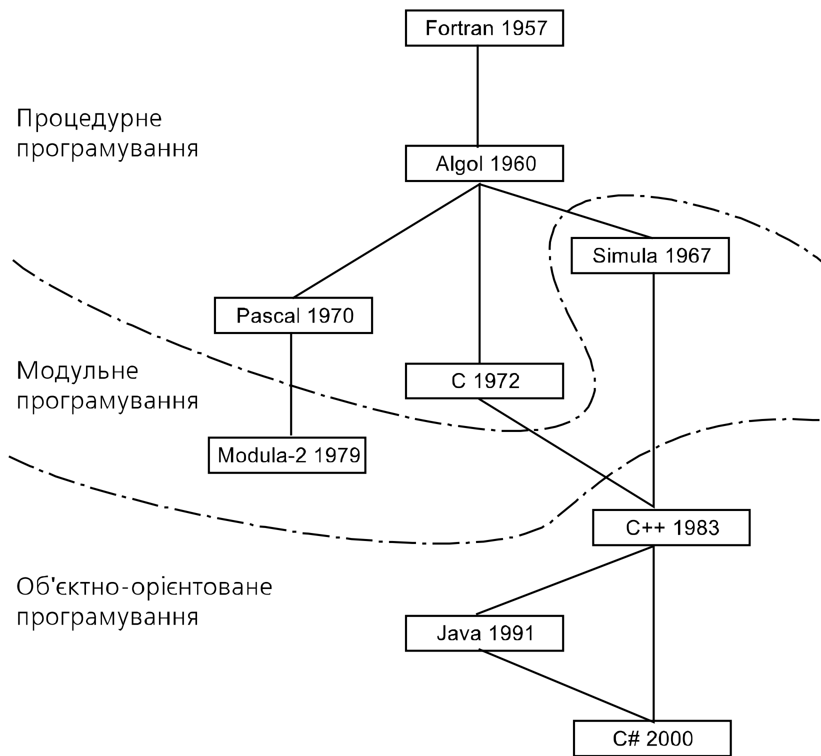


Рис. 1.2. Хронологія мов і парадигм програмування

П'ятдесят років розвитку програмування стали роками пошуків та утвердження виражальних засобів, роками зміни поколінь комп'ютерів, програмного забезпечення і мов програмування. На рис. 1.2 показано мови програмування, які зробили суттєвий внесок у розбудову сучасних парадигм. Так, мова програмування ФОРТРАН (FORmule TRANslator – перек-

ладач формул) уперше розв'язала проблему автоматизації програмування математичних формул, а її компілятори уперше впоралися із задачею роздільної компіляції.

Мова програмування АЛГОЛ (ALGOrithmic Language – алгоритмічна мова), у якій було запроваджено блочну структуру програми, стала першою мовою структурованого програмування із сучасними розгалуженнями і циклами. Майже одночасно з'явилися дві інші знакові мови програмування – Паскаль і С, які дотримувалися майже діаметрально протилежних підходів. Паскаль – прямий нащадок АЛГОЛу – доповнив його структури керування розвиненими структурами даних. Він стандартизував модель обчислювальної Паскаль-машини, призначеної для виконання програм на цій мові, й усі реальні обчислювальні середовища повинні були настроюватися на стандартну модель. Мова програмування С не передбачала стандарту обчислювача, а навпаки, містила засоби настроювання програми на конкретне обчислювальне середовище. Дослідження внеску в системне програмування, зробленого мовою С, стане одним із предметів вивчення цього підручника.

Проблема складності програмування на основі процедурної парадигми стимулювала пошук нових підходів. Так з'явилася концепція абстрактних типів даних як способу об'єднання даних і засобів їхньої обробки та одночасно як методу відокремлення специфікацій (інтерфейсів) від їх реалізації. Типи даних Паскалю були розвинуті в абстрактні типи даних у вигляді модулів мови МОДУЛА-2.

Особливо важливий крок було зроблено внаслідок залучення до мови С засобів мови моделювання СІМУЛА (SIMUlation Language – мова моделювання), в якій свого часу вперше з'явилися класи і об'єкти. З цього союзу виникла мова програмування С++, яка сама пройшла еволюційний шлях вдосконалень і розвитку. Цій темі присвячена одна з книг автора С++ Б'єрна Страуструпа [4].

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

Головне досягнення С++ полягає у *зміні парадигми програмування* (*paradigm shift*) з процедурної на об'єктно-орієнтовану, яка визначає стандарт розробки програмного забезпечення вже упродовж більш ніж чверті століття. Головним доробком об'єктно-орієнтованої парадигми стала ієрархічність програмних структур, яка відображається в агрегації об'єк-

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

Повертаючись до питання про вибір мови для курсу об'єктно-орієнтованого програмування, хочу наголосити, що я дотримуюсь тези, висловленої Стівом Макконнеллом у книзі «Довершений код» [7] (раджу її для подальшого самостійного читання), про те, що програмувати варто не мовою, а з використанням мови програмування. Тому я рекомендую C++ як мультипарадигменну мову, найбільш повну з погляду концепцій сучасного програмування. Наступники C++, а саме Java і C#, поки що парадигму не змінили, зате деякі важливі риси, наявні в C++, втратили.

Саме тому тут вивчатимемо насамперед найважливіші поняття сучасного програмування і зовсім не приділятимемо (або приділятимемо дуже мало) уваги проблемам синтаксису мови, хоча всі приклади програмних текстів перевірені системою програмування Microsoft Visual Studio. Для більшості прикладів достатньо рівня MS Visual Studio 6, але для деяких, що використовуватимуть порівняно нові властивості C++, будуть потрібні новіші версії.

1.2. Процедурне програмування

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

Спочатку процедурне програмування користувалося довільними засобами керування, зокрема *переходом* за міткою (`go to`) – одним із найбільш уживаних операторів керування в Фортрані. Ось приклад програмного тексту мовою Фортран:

ЛІСТИНГ 1.1.

```
C  КВАДРАТНИЙ КОРІНЬ З ДІЙСНОГО ЧИСЛА
   ПРОЦЕДУРА-ФУНКЦІЯ НА ФОРТРАНІ
   REAL FUCTION ROOT(A, EPS)
   S=A*.5
```

```
   IF (A.EQ.0) GO TO 20
10 T=S
   S=(S+A/S)*.5
   IF (ABS((S-T)/S).GT.EPS) GO TO 10
20 ROOT=S
   END
```

В 1968 році голландський вчений Едсгер Дейкстра вперше звернув увагу на проблеми, що виникають у програмах з неконтрольованими переходами, а в 1970 році проголосив новий напрямок, який він назвав структурним програмуванням. *Структурне програмування* (не зовсім вдалий переклад англійського *structured programming* – структуроване програмування) – це варіант процедурного, що використовує лише три типи структур керування: послідовне виконання дій, розгалуження і цикл. Не дивно, що Фортран не підтримував цю парадигму: в наборі його засобів не було циклів за умовами. Починаючи з Алголу, цикли стають основним засобом організації обчислень у програмі.

ЛІСТИНГ 1.2.

```
{ Квадратний корінь з дійсного числа
  Процедура-функція на Паскалі }
function root(a: real, eps: real): real;
   var s, t: real;
begin
   s:=a*0.5;
   if a<>0 then
   repeat
      t:=s; s:=(s+a/s)*0.5;
   until abs(s-t)/s<eps;
   root:=s;
end
```

Вправа 1.1.

Встановіть відповідність між конструкціями процедур, наведених у лістингах 1.1 і 1.2.

Професор Ніклаус Вірт, автор мови програмування Паскаль, відібрав до неї лише прості в поясненні й легкі в реалізації конструкції. Внаслідок сильної типізації програми на Паскалі відзначаються високою надійністю, закладена в них концепція Паскаль-машини робить їх мобільними, їх легко читати і розуміти завдяки дисципліні програмування, яка продиктована вжитою парадигмою.

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

1.3. Об'єктне (модульне) програмування

Процедурна парадигма віддала належне алгоритмічній компоненті програмування. Але зі зростанням обсягу програм і складності даних з'явилася нова проблема, а саме проблема структурної організації даних, найбільш влучно висловлена Віртівською формулою: «*алгоритми + структури даних = програми*» [18].

Поняття *модуля* як абстракції даних було вперше запропоноване Девідом Парнасом у 1972 році [20]; правда, на той час уже існувала мова програмування Симула–67, в якій була використана парадигма модулів у вигляді об'єктів. Найбільш повно поняття абстракції даних було реалізоване в мові програмування Модула–2 [19].

Головна ідея модульності даних полягає в забезпеченні доступу до них і оперування ними незалежно від способу їхнього конкретного кодування у пам'яті комп'ютера. Самі дані разом із процедурами їх обробки вбудовують (*інкапсулюють*) в окрему одиницю програми. Ось простий приклад, який демонструє, як складність обчислень може перетікати у складність даних.

Обчислення математичної функції, скажімо, квадратного кореня, можна виконати, обчислюючи границю відповідної послідовності, наприклад, такої:

$$x_0 = \frac{a}{2};\; x_n = (x_{n-1} + \frac{a}{x_{n-1}});\; n = 1,2,...;\; \lim_{n\to\infty} x_n = \sqrt{a}.$$

Цей же квадратний корінь можна знайти у відповідному рядку чотиризначних математичних таблиць Брадіса. У першому випадку вся складність зосереджена в алгоритмі, який необхідно придумати і запрограмувати (що було зроблено у наведених вище прикладах програм на Фортрані та Паскалі). У другому – в структурах даних, тобто в таблицях, які професор В.М. Брадіс повинен був попередньо розрахувати. Тут алгоритм обчислення зводиться до вибору потрібної клітинки таблиці.

Модулі мають дві головні риси. По-перше, вони об'єднують структури даних з алгоритмами їхньої обробки. По-друге, у них відокремлено специфікацію від реалізації інкапсульованих у модулі конструкцій, і це

перетворює модуль на так званий *абстрактний тип даних (abstract data type)*, на що свого часу звернув увагу Джон Гуттаг [21].

Ознайомитися з абстрагуванням даних можна на прикладі тексту мовою Модула–2, який дає певне уявлення про поділ модуля на дві частини – визначення і реалізацію:

ЛІСТИНГ 1.3.

```
{ Визначення модуля обчислення кількості днів
  у місяці мовою Модула-2 }
definition module Months;
   type Month=
      (jan, feb, mch, apr, may, jun, jul, aug, sep, oct, nov, dec);
   procedure length(m: Month): Cardinal;
end Month;
```

Наведеного у лістингу 1.3 визначення модуля `Months` достатньо для його використання. З тексту бачимо, що модуль містить тип даних `Month` для позначення місяців року і процедуру `length`, яка обчислює кількість днів у місяці, заданому як параметр. Результат цієї процедури є натуральним (кардинальним) числом, тип якого позначено як `Cardinal`. Щось подібне можна було б запрограмувати й на Паскалі, проте зв'язок між процедурою і типом даних, інкапсульований у цьому модулі, втратився б.

Окремо програмують реалізацію модуля (лістинг 1.4). Вона складається із визначення структур даних (у нашому випадку це масив кількості днів у місяцях), реалізації процедури та ініціалізації модуля (вона виконує підготовчу роботу з наповнення структур даних).

ЛІСТИНГ 1.4.

```
{ Реалізація модуля }
implementation module Month;
var
   len: array Month of Cardinal;
   procedure length(m: Month): Cardinal;
   begin
      return len[m];
   end length;
{ Ініціалізація модуля }
begin
   len[jan]:=31;   len[feb]:=28;   len[mch]:=31;   len[apr]:=30;
   len[may]:=31;   len[jun]:=30;   len[jul]:=31;   len[aug]:=31;
   len[sep]:=30;   len[oct]:=31;   len[nov]:=30;   len[dec]:=31;
end Month.
```

Як побачимо далі, поділ програмних конструкцій на визначення інтерфейсу та його реалізацію став типовою рисою сучасних мов програмування.

1.4. Об’єктно-орієнтовне програмування

Як було зазначено раніше, об’єктно-орієнтована парадигма розвиває об’єктне (модульне) програмування засобами створення ієрархій об’єктів і класів. Об’єктно-орієнтоване програмування, за метафорою Б’єрна Страуструпа, – це «*високоінтелектуальний синонім доброго програмування*». Справді, хоча нові парадигми програмування з’являються не так часто – приблизно одна на десятиліття, лише деякі з них, як, наприклад, структурне програмування, стають справжніми довгожителями. Той факт, що об’єктно-орієнтовану парадигму успішно використовують упродовж більш ніж чверті сторіччя, сам собою є вагомим підтвердженням її життєздатності.

Справді, алгоритми, реалізовані у процедурному програмуванні, надто конкретні. Будь-яка модифікація – це вже новий алгоритм, і тому кількість використовуваних процедур і функцій, як і затрати на їхнє розроблення або перепрограмування, надмірно зростають. Модульне програмування групує алгоритми в модулі (класи), інкапсулюючи разом з алгоритмами і відповідні структури даних. Тепер залишається зробити наступний крок – побудувати ієрархічну структуру.

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

Ієрархії можуть бути двох типів. Перший – це бути частиною чогось. Наприклад, грань є частиною многогранника, ребро – грані, вершина – ребра. Цей тип ієрархії дає змогу збирати об’єкти з частин, які самі є об’єктами. Інший, складніший тип дає можливість будувати узагальнення або, навпаки, конкретизації. Наприклад, овал і многокутник є конкретизацією плоскої фігури, коло – овалу, чотирикутник – многокутника; подальшими конкретизаціями чотирикутника можуть бути паралелограм, прямокутник, ромб, квадрат. Те, що квадрат, ромб чи прямокутник є повноцінними паралелограмами, дає їм змогу користуватись усіма програмними засобами, створеними для паралелограмів; паралелограм зі свого боку є повноцінним чотирикутником і так далі. Цей принцип, відомий під

назвою *reusable – знову (або навіть заново) вживаний*, – став одним із найважливіших досягнень об'єктно-орієнтованої парадигми. Щоб знову використати вже наявне програмне забезпечення у більш конкретизованих чи навіть зовсім несподіваних умовах, достатньо дописати лише ту його частину, яка стосується особливостей відповідної конкретизації. Цей принцип отримав назву *programming by difference* або *дописування програм*.

І, нарешті, об'єктно-орієнтована парадигма доводить до логічної завершеності принцип моделювання реального світу, а точніше – тієї його частини, абстракцією якої є програма. За такого підходу програма складається з об'єктів, які відповідають реальним поняттям або предметам, а її виконання зводиться до взаємодії цих об'єктів, що є абстракцією реальної взаємодії їхніх прототипів. Все це разом забезпечує об'єктно-орієнтованому підходу беззаперечне лідерство в галузі розробки програм.

Сьогодні в сімействі мов об'єктно-орієнтованого програмування три найбільш відомі представники: C++, Java і C# (читають як «Сі шарп»). C++ дотепер залишається визнаним лідером у розробленні великих і складних програмних систем. Java і C# виросли з C++, вони мають свою сферу застосування в розподіленому програмуванні.

Ще одна перевага C++ – її мультипарадигменність. C++ містить у собі мову C (з деякими застереженнями), а тому, природно, підтримує процедурну парадигму. Ось приклад функції для обчислення квадратного кореня в C/C++, після якої для порівняння розмістимо аналогічну функцію на Паскалі.

ЛІСТИНГ 1.5.

```
/* Квадратний корінь з дійсного числа
   Функція мовою C  */
double root(double x, double eps) {
   double s=0.5*x;
   double t=s;
   if (x!=0)
      do {
         t=s; s=(s+x/s)*0.5;
      } while ((fabs(s-t)/s)>eps);
   return s;
}
```

ЛІСТИНГ 1.6.

```
{ Квадратний корінь з дійсного числа
  Процедура-функція на Паскалі }
function root(a: real,eps: real): real;
```

```
   var s, t: real;
begin
   s:=a*0.5;
   if a<>0 then
      repeat
         begin
            t:=s; s:=(s+a/s)*O.5;
         end
      until abs(s-t)/s<eps;
   root:=s;
end
```

Обидві функції не відрізняються одна від одної з точністю до позначень для структур керування. Несуттєві відмінності полягають у способі запису заголовка, визначеннях та ініціалізації змінних, позначеннях для розгалуження і циклу (зокрема є відмінність семантики циклу з умовою закінчення), позначеннях операторних дужок і способі повернення результату.

C++ також підтримує традиційні для модульного програмування механізми абстракції даних, доповнені можливостями об'єктно-орієнтованої парадигми. Ось якого вигляду на C++ набуде клас місяців року, відповідний до прикладу з лістингів 1.3-1.4 мовою Модула–2:

ЛІСТИНГ 1.7.

```
// Клас C++ для обчислення кількості днів у місяці
// Специфікація класу
class Month {
public:
   enum month
      {jan=1, feb, mch, apr, may, jun, jul, aug, sep, oct, nov, dec};
   int length();
   Month(month n): _mon(n) {}
private:
   month _mon;
   static int _len[12];
};
// Реалізація класу
int Month::length() {
   return len[mon-1];
}
int Month::len[12] {31,28,31,30,31,30,31,31,30,31,30,31};
```

Відповідність текстів обома мовами у цьому випадку стала не такою очевидною, як для процедурного варіанту. Більше того, відмінність мо-

дульного і об’єктного підходів стає помітною вже на цьому простому прикладі.

На відміну від модуля, клас `Month` – це повноцінний тип даних, який можна використовувати, нарівні зі стандартними, для визначення змінних, наприклад:

ЛІСТИНГ 1.8.

```
Month theMonth=mai;
int i=theMonth.length();
```

Як побачимо далі, на новостворений тип можна навіть поширити набір стандартних операції, наприклад, визначивши операцію збільшення місяця на ціле число, що дасть змогу писати вирази на кшталт `theMonth+5`.

І, нарешті, крім процедурної, об’єктної та об’єктно-орієнтованої парадигм, C++ реалізує також для кожної з них парадигму узагальненого програмування, надаючи можливість використання узагальнених функцій і параметризованих класів, але про це далі.

1.5. Програмні середовища

Від складання програмістом до виконання комп’ютером програма проходить доволі тривалий процес оброблення спеціальними службовими програмами, які становлять систему автоматизації програмування. З часом слово «автоматизація» випало із наведеного словосполучення, внаслідок чого воно перетворилося на «систему програмування». Ця система складається з кількох компонент, а саме *препроцесора (preprocessor), компілятора (compiler)* та *компонувальника (linker)*, а також засобів підтримки етапу виконання, зокрема *налагоджувача (debugger)*, об’єднаних спільним інтерфейсом у так зване універсальне середовище розробки програм. Прикладами таких середовищ можуть бути система програмування Visual C++ та діалогове середовище розробки програм Developer Studio.

Головна особливість підготовки програми до виконання полягає в тому, що програму збирають із багатьох, часом різнорідних, складових частин, об’єднаних у *програмний проект (software project)*. Кожну програмну розробку оформлюють як окремий проект, який зазвичай складається з багатьох різнорідних файлів. Файли першого типу називають *вхідними (source file, исходный файл*), вони містять тексти, написані певною мовою програмування. Робота над проектом може бути достатньо тривалою, в її процесі виникатимуть нові вхідні файли, які додаватимуться до наявних. Тому в кожен момент часу частина вхідних файлів може виявитися вже готовою до використання, тобто попередньо відкомпільованою. Цю частину вико-

ристовують у вигляді готових машинних кодів, що зберігаються в проекті як особливі *об'єктні файли (object file)*, котрі складають другий тип файлів програмного проекту. Система програмування автоматично відстежує необхідність повторного компілювання кожного вхідного файлу. Компіляції підлягає кожен новий файл, долучений до проекту, а також будь-який інший файл після внесення до нього змін.

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

Вже упродовж не одного десятиріччя під час складання великих за розмірами програм використовують принцип структурної декомпозиції. Логічним складовим частинам програми (у процедурному програмуванні це процедури і функції, у більш розвинених парадигмах – модулі або класи) відповідають фізичні складові – файли, що містять ці окремі завершені логічні частини, одну або декілька разом. Одночасне використання в одному програмному проекті багатьох файлів з текстами різних частин програми одержало назву *роздільної компіляції (separate compilation)*, яка, без сумніву, стала одним із найбільших досягнень систем програмування. Завдяки їй програму тепер поділено на частини, які називаються *одиницями трансляції* (*translation unit*). З використанням роздільної компіляції стало можливим колективне розроблення великих програм: окремі розробники зайняті підготовкою кожен своїх файлів.

Програмування – це діяльність, яка потребує великої організованості. Прийнято не тільки розділяти текст програми на структурні частини, але й розрізняти описові та виконавчі частини текстів. Тому під час складання програм дотримуються певних правил доброго тону, одне з яких полягає в необхідності розподілення визначень і обчислень між вхідними файлами двох типів: *файлами заголовків (header) і файлами реалізації (implementation)*. Особливий компонент системи програмування – *препроцесор* – у відповідності з директивами, розміщеними, як правило, на початку вхідного файлу, приєднує до нього відповідні файли заголовків. Вони містять інформацію про те, які функції, об'єкти або класи, визначені поза цим вхідним файлом, можуть бути використані в ньому. Один файл заголовків можна приєднувати до багатьох вхідних файлів, які ви-

користовуватимуть оголошені в ньому конструкції. Приєднанням потрібних заголовних файлів препроцесор готує вхідні файли до подальшої обробки компілятором. Тому вміст вхідного файлу для компілятора, взагалі кажучи, відрізняється від вмісту вхідного файлу, складеного розробником, оскільки обробка препроцесором може значно його змінити. Далі розглянемо, в яких межах варто зосередити цю обробку.

Набір пов'язаних один з одним заголовних і вхідних файлів, з яких складається проект, в сукупності утворює *вхідну програму*, яку система програмування перетворюватиме на машинний код. Машинний код генерується у два етапи. На першому компілятор генерує так звані *об'єктні коди (object code)*, по одному для кожного файлу реалізації (за заголовними файлами об'єктні не генеруються). Коди називають об'єктними (від слова «objective», що англійською мовою означає «мета»), оскільки їхнє створення є метою роботи компілятора, який становить центральну частину системи програмування. Інша її частина, названа *компонувальником (linker)*, збирає об'єктні файли в одну *виконавчу програму (executable program*, *exe-file)*, приєднуючи до неї також об'єктні коди зі стандартних або власних бібліотек проекту.

Ми розглядатимемо систему програмування такою мірою, наскільки вона впливає на написання та організацію текстів вхідних програм. За детальним ознайомленням із засобами і можливостями систем програмування відсилаємо до системної документації та спеціальної літератури.

Ось типовий сценарій підготовки програми.

Система програмування працює з *програмними проектами*, де розміщують файли вхідної програми. Розглянемо, наприклад, проект **sqrt**. До файлу реалізації **root.cpp** запишемо текст функції `root` для обчислення квадратного кореня, а програмний код для її виклику помістимо у файл реалізації **main.cpp**.

Пам'ятаймо, що компіляція цих двох файлів виконуватиметься нарізно. Виникає запитання: звідки під час компіляції файлу **main.cpp** компілятор візьме відомості про ідентифікатор `root`? У принципі можливі два вирішення. Перше – це перемістити текст функції з файлу **root.cpp** до файлу **main.cpp**, де функцію використано. Проте це було б не дуже вдалим виходом, оскільки незрозуміло, як одну й ту ж функцію розмістити у різних вхідних файлах, де її також можна було б викликати. Друге вирішення, а саме воно прийняте в системах програмування, полягає у створенні допоміжного заголовного файлу **root.h**, який міститиме не всю інформацію про функцію `root`, а лише ту, що необхідна для обробки її виклику компілятором, – так звану *сигнатуру функції*:

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

Проект **sqrt**, наповнений вхідними файлами, зображено на рис. 1.3.

Заголовний файл приєднують до вхідного за допомогою команд препроцесора. Їх записують за допомогою спеціальних директив, які починаються символом **#**. Зокрема, файл **main.cpp** міститиме директиву **#include "root.h"**, виконання якої препроцесором полягає у включенні тексту з файлу заголовку **root.h** до вхідного файлу **main.cpp**. Наведемо текст, доповнений коментарями, покликаними зробити його зрозумілим навіть для читача, який ще зовсім не знає C.

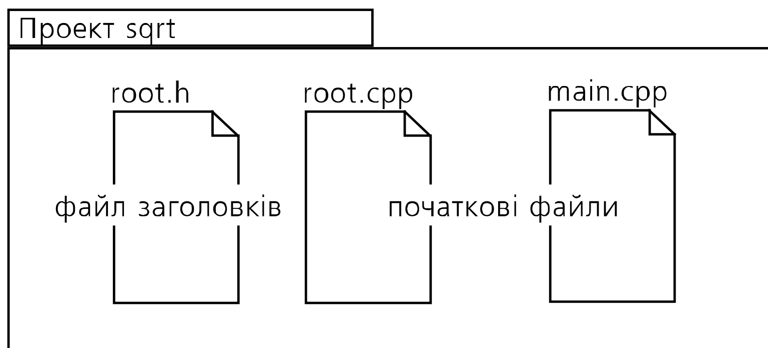


Рис. 1.3. Склад вхідних файлів проекту sqrt

ЛІСТИНГ 1.9.

```
// main.cpp
// Спочатку системні заголовки
// Підключення бібліотеки введення та виведення
#include <iostream>
// Підключення бібліотеки математичних функцій
#include <cmath>
// Посилання на стандартні позначення
using namespace std;
// Тепер власні заголовки
#include "root.h"
// Початок власне програми, вона називається main
int main() {
// cout<<someting; вивести something
// cout<<someting<<endl; вивести те ж саме і перейти на новий рядок
   cout<<"The square root of 2 calculated with"<<
      endl<<" a standart fuction is"<<sqrt(2.0)<<endl;
```

```
   cout<<" your function is "<<root(2.0,0.000001)<<endl;
   return 0;
}
```

Як бачимо, крім заголовного файлу **root.h**, вхідний файл використовує ще кілька інших, службових файлів, приєднаних препроцесором (їх не показано на рис. 1.3). Файли **iostream** і **cmath** описують стандартні бібліотеки: введення-виведення **iostream** (вона потрібна для обслуговування вихідного потоку `cout`) та математичних функцій **cmath**, звідки буде взято, наприклад, функцію обчислення квадратного кореня `sqrt`. Директива `using namespace std` є вказівкою компілятору вживати стандартні імена для системних об’єктів: наприклад, через `cout` позначено стандартний вихідний потік.

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

ЛІСТИНГ 1.10.

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

Тепер коротко про призначення інших компонентів системи програмування. Після того, як буде проінтепретовано команди препроцесора, файли повних текстів надійдуть на вхід компілятора, який створить із них об’єктні коди (рис. 1.4). Об’єктний код ще не призначено для безпосереднього виконання. У проекті таких кодів багато, взагалі кажучи, по одному на кожен вхідний файл, і вони містять взаємні посилання один на одного, а також на бібліотеки. Об’єктні коди надходять на вхід компонувальника, завдання якого – зібрати їх разом, доповнити необхідними компонентами бібліотек та перетворити на виконавчу програму.

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

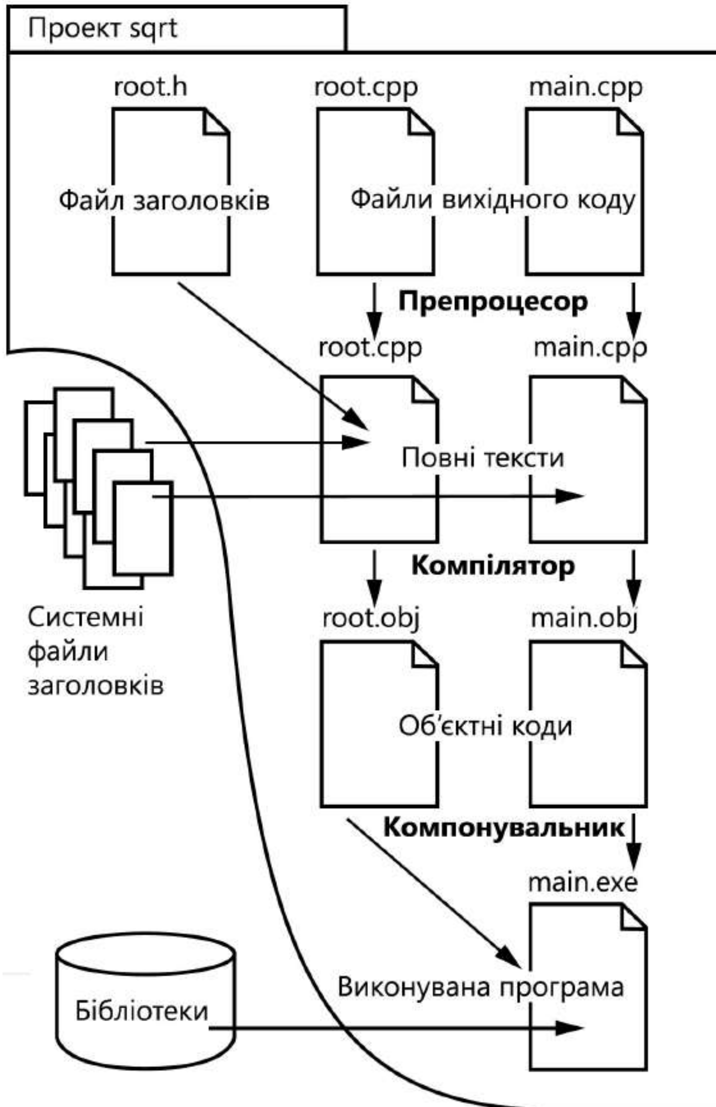


Рис. 1.4. Повний склад файлів проекту sqrt

Завдання до розділу 1

1. Інсталюйте систему програмування C++ (MicrosoftVisualStudio, GCC або іншу). Керуючись підказками системи, створіть консольний порожній проект **sqrt**.
2. Створіть у проекті файл **root.cpp**. Приєднайте до нього системний заголовний файл **cmath**. Розмістіть у ньому текст функції `root` та виконайте його компіляцію.
3. Створіть у проекті файл **root.h**. Розмістіть в ньому заголовок функції **root**.
4. Створіть у проекті файл головної функції **main.cpp**. Приєднайте до нього власний заголовний файл **root.h** та системні файли **iostream** і **cmath**. Включіть директиву `using namespace std`. Запишіть текст функції `main`. Виконайте компіляцію.
5. Користуючись документацією системи програмування, зберіть і виконайте проект в автоматичному режимі.
6. Модифікуйте головну функцію, визначивши в ній змінну `x` та вказавши оператори введення підкореневого значення:

   ```
   double x(2.0);
   cout<<"give your value for x ";
   cin<<x;
   ```

   та відповідно замініть аргументи функцій `sqrt` та `root` на `x`. Виконайте модифіковану в цей спосіб програму.
7. Користуючись налагоджувачем, виконайте функцію `main` в покроковому режимі, спостерігаючи за ходом обчислень.
8. Увійдіть в процесі виконання до функції `root` та простежте за зміненням значень змінних.

2. Базові поняття програмування

Як відомо, пам'ять комп'ютера складається з послідовностей бітів, а тому всі дані, що зберігаються в пам'яті, являють собою двійкові коди. Тривалий час здавалося, що можна обходитися лише ними, даючи програмі змогу під час виконання «на власний розсуд» динамічно інтерпретувати один і той самий код залежно від обставин. Однак такий спосіб безтипового програмування виявився надто небезпечним. Навіть у випадку числових величин, ґрунтуючись лише на двійковому коді, не завжди можна правильно відтворити значення об'єкта: адже один і той самий двійковий код може кодувати різні значення залежно від типу об'єкта. І навпаки: як побачимо далі, одне й те саме, навіть найпростіше числове значення, наприклад число 1, залежно від наданого йому типу, може бути подано різними двійковими кодами. Тому знання типу об'єкта – вирішальний фактор для визначення допустимих операцій над ним.

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

Тяжіння до дедалі сильнішої типізації – загальна тенденція розвитку програмування. Кожному об'єкту програми (і, відповідно, області пам'яті, що зберігає його як елемент даних) приписано певний тип, від якого залежить інтерпретація даних, а саме спосіб їх кодування і декодування та, головне, набір допустимих операцій над ними. Навіть якщо області даних різних типів перетинаються (наприклад, цілі та дійсні числа), то спосіб кодування числової величини залежить від приписаного їй типу. Скажімо, машинний код цілого числа `1` і дійсного числа `1`, як уже було сказано, відрізняються.

Мова C в процесі свого розвитку пройшла шлях від *слабкої типізації* у напрямку посилення цієї властивості, тому тепер C та C++ – сильно типізовані мови, хоча і зберегли деякі риси своєї попередниці (див. неявні перетворення типів у розділі 2.3.3). Кожному об'єкту програми приписують певний тип, у найпростішому випадку один. Від нього залежить, що дозволено робити з цим об'єктом. Можна не лише використовувати стандартні типи, наперед задані мовою, а й створювати власні. Вони можуть утворювати ієрархії, що ускладнює проблему типізації. Однак це вже пи-

тання іншого рівня, відоме під назвою поліморфізму. Про це йтиметься в заключних розділах підручника.

Перш ніж розглядати стандартні типи, зробимо коротке зауваження про спосіб запровадження власних позначень для типів, як програмованих, так і стандартних.

Зауваження. *Створення позначень для типів за допомогою оператора* `typedef` *– ще один атавізм безтипової мови С, оскільки задані у такий спосіб типи лише відіграють роль синонімів для своїх базових типів і можуть бути довільно змішані один з одним. Проте в одному заголовному файлі зазвичай міститься чимало визначень типів, а тому корисно принаймні вміти їх читати. Визначення повноцінних типів за допомогою класів розглянемо пізніше.*

Проаналізуємо простий приклад.

Лістинг 2.1

```
typedef int length;   // Тип довжина
typedef int weight;   // Тип вага
length f(length x) {return x;}   // Приймач довжини
weight g(weight x) {return x;}   // Приймач ваги
int main() {
   int a=123;
   a=f(a); a=g(a);
   return 0;
}
```

Його безпроблемна компіляція свідчить, що компілятор не лише не розрізняє типи `length` і `weight`, але й не відрізняє їх від стандартного цілого типу **`int`**.

Наведемо приклади кількох системних типів, які введено як альтернативні позначення, що використовуються як підказки щодо їх призначення.

Лістинг 2.2

```
typedef unsigned short wchar_t;
typedef long time_t;
typedef unsigned int size_t;
```

У першому рядку наведено альтернативне позначення `wchar_t` для типу **`unsigned short`**. Воно використовується, коли коротке ціле без знаку кодує двобайтові символи. У другому рядку позначення `time_t` вжито для типу **`long`**, як підказка до його застосування для часових величин. У третьому – позначення **`size_t`** застосовано для типу **`unsigned int`**, що служить натяком на його застосування для позначення розмірів виділеної пам’яті. Типи **`unsigned short`**, **`long`**, **`unsigned int`** і багато інших належать до стандартних типів мови С, вивчення яких ми зараз і почнемо.

2.1. Фундаментальні типи даних і операції над ними

Повний список *фундаментальних типів* (fundamental types) наведено далі в табл. 2.1.

Таблиця 2.1. Фундаментальні типи даних

Назва типу	Позначення	Діапазон значень
Логічний (булевий)	`bool`	`false, true (true+true==true)`
Символьний	`char, unsigned char`	Від `0` до `255` (`255+1=0`)
	`signed char`	Від `-128` до `127` (`127+1=-128`)
Цілий	`short (short int)`	Визначається з умов: `1≡sizeof(char)≤sizeof(short)` `≤sizeof(int)≤sizeof(long);` `1≤sizeof(bool)≤sizeof(long);` `sizeof(N)≡sizeof(signed N)≡` `sizeof(unsigned N);`
	`unsigned short (unsigned short int)`	
	`int (signed int)`	
	`unsigned (unsigned int)`	
	`long (long int)`	
	`unsigned long (unsigned long int)`	
	`long long (long long int)`	
Довгий символ	`wchar_t`	`unsigned short`
Дійсний, звичайна точність	`float`	`sizeof(double)≤sizeof(double)` `≤sizeof(long double)`
Дійсний, подвійна точність	`double`	
Дійсний, розширена точність	`long double`	
Без типу	`void`	Немає

Почнемо з типу **`void`**, який лише формально віднесено до типів – передусім для того, щоб одним і тим самим способом розглядати процедури

та функції, а також функції без параметрів і безтипові указники (підрозділ 2.4.4). Цей тип не має множини значень і операцій.

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

Зауваження. *Термін floating point доволі незвичний для узвичаєного в нас способу відокремлення дробової частини в десяткових дробах комою, а не крапкою. В Америці вживають десяткову крапку, і саме її використовують у мовах програмування.*

Для програміста, вихованого на Паскалі, арифметичні операції над булевими типами є більш ніж незвичними. Чому булевий або символьний типи належить до арифметичних? Як виконувати логічні операції над цілими значеннями? Відповідь проста. Спочатку в мові C зовсім не було булевого типу. Забігаючи наперед (а нам доведеться робити це доволі часто), можна сказати, що назва цього типу – просто синонім, інше ім'я цілого типу.

Хоча в нових версіях мови C++ уже визначено булевий тип, для збереження сумісності версій він залишається арифметичним. Те саме стосується й символьного типу, як це видно з наступного фрагмента коду.

Лістинг 2.3

```
int f(int x) {return x+1;}
int main() {
   bool t=false; cout<<t<<endl; t=f(t); cout<<t<<endl;
   char ch='a'; cout<<ch<<endl; ch=f(ch); cout<<ch<<endl;
   return 0;
}
```

З лістингу 2.3 видно, що над булевими величинами можна (проте чи потрібно?) виконувати звичайні арифметичні операції. Правда, цей тип має одну особливість, пов'язану з наявністю лише двох допустимих значень – істинного та хибного. Чи вдається у такому разі проінтерпретувати довільне ціле значення як булеве? Домовилися так: усі цілі значення, відмінні від нуля, уважати істинними. Як приклад наведемо фрагмент програми.

Лістинг 2.4

```
for(int i=0; i<=255; ++i)
   if(i) cout<<true<<' ';
```

```
   else cout<<false<<' ';
cout<<endl;
```

Тепер уже можна не дивуватися, що символьний тип теж уважають арифметичним. Кодову таблицю символів у десяткових, вісімкових і шістнадцяткових кодах можна вивести за допомогою циклу, наведеного в лістингу 2.5.

Лістинг 2.5

```
for(int k=0; k<256; ++k)
   cout<<dec<<k<<':'<<oct<<k<<','<<hex<<k<<','<<char(k)<<endl;
```

Кількість арифметичних типів у мові C++ дещо надмірна. Для більшості задач достатньо чотирьох із них: `bool`, `char`, **`int`** і **`double`** (зазвичай застосовують подвійну, а не звичайну точність). Інші арифметичні типи призначені для застосування в особливих випадках (оптимізація, системне програмування тощо), проте ми їх майже не використовуватимемо.

2.1.1. Розрядність фундаментальних типів даних

Мова програмування не регламентує розрядність типів даних. Вона залежить від архітектури апаратного забезпечення та компілятора. Точні значення розміру пам'яті, що відводиться системою програмування для кожного з типів, можна обчислити програмно. Наприклад, **`sizeof(bool)`** або **`sizeof(true)`** – розмір значення булевого типу (**`sizeof`** – це операція, значенням якої є розмір в байтах пам'яті, відведеної для типу чи об'єкта). Наведемо програму, яка відображає кількість байтів, використаних для зберігання величин кожного з типів. Дещо забігаючи наперед ми запишемо функцію обчислення ємності типу даних у вигляді шаблону (за поясненнями відсилаємо читача до підрозділу 3.6.1).

Лістинг 2.6

```
// Обчислення ємності довільного типу даних T у бітах
template <typename T>
unsigned int size_in_bits() {
   return sizeof(T)*8;
}
```

Ось приклад викликів функції з лістингу 2.6, (в коментарях наведені значення видані системою програмування Microsoft Visual Studio 2008).

```
cout<<size_in_bits<bool>()<<endl; // 8
cout<<size_in_bits<char>()<<endl; // 8
```

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

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

Чотирибайтові величини можуть набути 2^{32} = 4 294 967 296 значень із діапазону від 0 до 4 294 967 295. Зрозуміло, що чотирибайтових цілих типів не вистачить, наприклад, для точного підрахунку жителів земної кулі. Зауважимо, що пам'ятати точне значення кожного степеня двійки не обов'язково, оскільки в багатьох випадках достатньо його наближення, обчисленого за допомогою так званої *основної тотожності інформатики*

$$2^{10} \approx 10^3.$$

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

Вам буде корисно самостійно розібратися зі способом кодування дійсних чисел у вигляді порядку й мантиси та скласти програму, що підраховує максимальну кількість десяткових цифр у мантисі дійсного числа.

Наведеним вище способом нескладно визначити порядок значень восьмибайтових величин $2^{64} = 2^4 \cdot 2^{60} = 16 \cdot (2^{10})^6 \approx 16 \cdot (10^3)^6 = 16 \cdot 10^{18}$ = 16 000 000 000 000 000 000.

Вправа 2.1

Програмно визначте кількість десяткових цифр мантиси в дійсному числі з подвійною точністю.

Важливо пам'ятати*! Пам'ять комп'ютера – це не просто послідовність байтів; її поділено на півслова, слова та подвійні слова. У 32-розрядній архітектурі адреса півслова кратна 2, адреса слова – 4, а подвійного слова – 8. Однобайтові величини можуть міститися в будь-якому байті, двобайтові – лише в півсловах, чотирибайтові – у словах, восьмибайтові – у подвійних словах. Тому в пам'яті можуть бути «прогалини».*

Можна також створювати власні *перелічувані типи* – *переліки* (enumeration). Наприклад, перелік

```
enum week
   {monday=1, tuesday, wednesday, thursday, friday,saturday, sunday};
```

задає новий тип, складений з іменованих констант, значення яких – це цілі числа, починаючи з 1. На жаль, ідентифікатори, вжиті для іменування переліку в програмі, так само, як інші імена змінних або сталих величин не призначені для відтворення на зовнішніх пристроях. Тому спроба вивести константу `wednesday` призведе до відображення числа 3, а команда

```
cout<<sunday<<endl;
```

просто виведе число 7. Щоб зберегти символьні позначення переліків для виведення, знадобиться додаткове програмування, наприклад, створення текстового масиву з відповідних позначень. Ми проілюструємо це пізніше, у лістингу 2.63.

За необхідності вживати переліки, складені не з послідовних цілих чисел, їх значення слід задавати явно:

```
enum number {two=2, eight=8, ten=10, hex=16};
```

Якщо значень констант з переліку не задано явно, перелік розпочнеться з нуля:

```
enum cardinal {zero, one, two};
```

Над величинами перелічуваних типів можна виконувати арифметичні операції як над значеннями цілого типу.

2.1.2. Операції над даними фундаментальних типів

Арифметичні операції мови C++ подано в табл. 2.2. Усі арифметичні операції, окрім операції обчислення остачі від ділення, яка має сенс лише у разі цілого типу, застосовують до всіх арифметичних типів, зокрема булевих. Можливість виконання арифметичних операцій над булевими значеннями збережена для забезпечення сумісності старіших і новіших версій мови. Так само, як людина з певним рівнем освіти, котра повсякденно вживає сучасну літературну мову, а не староукраїнську чи церковнослов'янську, у разі потреби (чи за бажанням) здатна прочитати і зрозуміти тексти, складені Григорієм Сковородою, преподобним Нестором-літописцем або невідомим автором «Слова о полку Ігоревім», прихильники найсучасніших технологій програмування можуть читати і використовувати архаїчні програмні тексти, нерідко схожі на ребуси чи головоломки. За певних умов це може навіть принести задоволення.

Таблиця 2.2. Арифметичні операції

Арифметичні операції		
Символ операції	***Назва***	***Спосіб використання***
`*`	Множення	`op1 * op2`
`/`	Ділення	`op1 / op2`
`%`	Остача	`op1 % op2`
`+`	Додавання	`op1 + op2`
`-`	Віднімання	`op1 - op2`
`,`	Кома	`op1 , op2`

Усі операції, крім останньої, загальновідомі. Операцію ж «*кома*» (sequence operator) використовують тоді, коли на місце, призначене для одного виразу, потрібно розмістити послідовне обчислення значень двох виразів один за одним у порядку їх слідування: спочатку першого операнда, а за ним другого. Значенням виразу, побудованого за допомогою цієї операції, уважають значення її другого операнда, обчислення першого операнда вживається для створення *стороннього ефекту* (side effect; рос. – побочный эффект). Саме завдяки цьому ефекту обчислене та начебто забуте значення може змінити стан пам'яті програми. Типовими прикладами першого операнда коми служать вирази, побудовані за допомогою операції присвоєння, а також операцій інкременту й декременту (див. підрозділ 2.3.2). Наведемо простий приклад застосування операції «кома» (порівняйте два розгалуження).

Лістинг 2.7

```
// Умова стосується обох присвоєнь
if(i<j)
   i=i+1, j=j+1;
// Умова стосується лише першого присвоєння
if(i<j)
   i=i+1; j=j+1;
// Останнє розгалуження доцільніше переписати так:
if(i<j)
   i=i+1;
j=j+1;
```

До величин арифметичних типів можна застосовувати не лише арифметичні операції, а й операції порівняння (табл. 2.3). Їх результат завжди має булевий тип.

Таблиця 2.3. Операції порівняння

Операції порівняння		
Символ операції	***Назва***	***Спосіб використання***
`<`	Менше	`op1 < op2`
`<=`	Менше чи дорівнює	`op1 <= op2`
`>`	Більше	`op1 > op2`
`>=`	Більше чи дорівнює	`op1 >= op2`
`==`	Дорівнює	`op1 == op2`
`!=`	Не дорівнює	`op1 != op2`

Слід звернути особливу увагу на специфічне позначення рівності. Уживання знака присвоєння «=» замість знака рівності «==» – одна з най-

типовіших помилок початківців, які пишуть на C/C++. Такі помилки компілятор не в змозі виявити.

Лістинг 2.8

```
int k=0;
if(k=0)
   cout<<"You never come here"<<endl;
else
   cout<<"Still k is equal "<<k<<endl;
```

Зовсім інша справа –

```
if(k==0)
   cout<<"That is OK this way"<<endl;
```

Крім загальновживаних логічних операцій – заперечення, кон’юнкції та диз’юнкції, – мова містить також тримісну операцію імплікації (табл. 2.4).

Таблиця 2.4. Логічні операції

Логічні операції		
Символ операції	***Назва***	***Спосіб використання***
`!`	Заперечення	`! op`
`&&`	Кон’юнкція	`op1 && op2`
`||`	Диз’юнкція	`op1 || op2`
`? :`	Імплікація	`(op1 ? op2 : op3)`

Аргументи й результат логічних операцій, окрім імплікації, мають булевий тип. Імплікацію використовують для створення умовних виразів. Її перший операнд булевий, а типи двох інших повинні збігатися чи зводитись один до іншого; результат має тип другого або третього операнда. Імплікацію можна визначити тотожностями

```
(true ? x : y)==x
(false ? x : y)==y
```

Окрім названих у мові C++ є також побітові логічні операції (табл. 2.5). Це операції низького (машинного) рівня. Їх застосовують до операндів інтегральних типів, і вони дають результат цього ж типу. У цьому разі інтегральний тип інтерпретують не як ціле число зі знаком, а як послідовність нулів і одиниць. Тому такі операції доцільно застосовувати до даних беззнакових типів (наприклад, `unsigned int` або `unsigned long long`), що відповідає машинному слову чи подвійному слову. Для

величин коротших типів або зі знаком результат може залежати від типу компілятора. Це означає, що одна й та сама програма після підготовки її до виконання різними компіляторами може давати різні результати. Які саме, легко визначити за допомогою спеціальної тестової програми (подумайте, як її скласти). Орієнтуючись на сучасні технології програмування, ми надаватимемо перевагу програмам, незалежним від компілятора і навіть від платформи; їх називають *переносними* (portable). Тому побітові логічні операції застосовуватимемо лише до машинних слів.

Таблиця 2.5. Побітові логічні операції

<table>
<tr><th>Символ операції</th><th>Назва</th><th>Спосіб використання</th></tr>
<tr><td>~</td><td>Заперечення</td><td>~ op</td></tr>
<tr><td><<</td><td>Зсування вліво</td><td>op1 << op2</td></tr>
<tr><td>>></td><td>Зсування вправо</td><td>op1 >> op2</td></tr>
<tr><td>&</td><td>Кон’юнкція</td><td>op1 & op2</td></tr>
<tr><td>^</td><td>Виключна диз’юнкція</td><td>op1 ^ op2</td></tr>
<tr><td>|</td><td>Диз’юнкція</td><td>op1 | op2</td></tr>
</table>

Використання побітових логічних операцій – це програмування на низькому, близькому до машинного, рівні.

Наведемо простий приклад. Візьмемо ціле число `a` = 4 227 661 320. Воно менше за число $2^{32}-1$ = 4 294 967 295, а тому для його зберігання достатньо одного машинного слова, що відповідає типу **`unsigned int`**). Тепер послідовно розглянемо значення виразів `(a>>k)` і `(a>>k)&255` за різних значень `k` (0, 8, 16, 24) (табл. 2.6).

Таблиця 2.6. Результати зсуву і кон’юнкції

k	Результат зсуву		Результат кон’юнкції	
	Десятковий	Шістнадцятковий	Десятковий	Шістнадцятковий
	(4227661310 >>k)	(fbfcfdfe >>k)	(4227661310 >>k)&255	(fbfcfdfe >>k)&ff
0	4227661310	fbfcfdfe	254	fe
8	16514301	00fbfcfd	253	fd
16	64508	0000fbfc	252	fc
24	251	000000fb	251	fb

Прокоментуємо наведені в цій таблиці результати.

Після обчислення першого виразу `(a>>k)` маємо зсув двійкового коду на нуль (тобто без зсуву), один, два чи три байти. Отримані значення на-

ведено в другому стовпчику табл. 2.6. Ужита тут десяткова форма запису дає слабке уявлення про спосіб виконання операції зсуву. Його краще відстежити за двійковим записом або компактнішим шістнадцятковим, який наведено в третьому стовпчику таблиці.

Застосована далі операція кон'юнкції `(a>>k)&255` – типовий приклад виділення частини двійкового коду за маскою. Тут шістнадцятковий запис маски `255` має вигляд `ff` або, у форматі повного слова, – `000000ff`. Із результатів видно, що застосування порозрядної кон'юнкції за маскою 255 «виділяє» молодший байт слова, обнулюючи решту його розрядів, що відображено у двох останніх стовпцях таблиці в десятковому та шістнадцятковому записах.

Вправа 2.2

За допомогою яких масок можна виділити другий, третій, четвертий байт машинного слова? А як виділити ліве півслово? А праве півслово?

Вправа 2.3

Запрограмуйте обчислення виразів і виведення їх значень:

```
4227661310 & 65280;
4227661310 & 16711680;
4227661310 & 4278190080;
~0l;
~2;
~(-1);
0&2;
2&(-1).
```

Правила побудови виразів за допомогою розглянутих операцій загалом є звичними, якщо не брати до уваги кілька особливостей мови С, зокрема використання присвоєння як операції. Ми ознайомимось із ними пізніше.

2.2. Літерали та їхні типи

Літерал (від лат. literal – буквальний) – це найпростіший об'єкт, що позначає сам себе. Наприкінці попереднього розділу ми розглядали числові літерали – десяткові та шістнадцяткові записи чисел. Літерал не потрібно попередньо оголошувати чи визначати. Тому, коли він зустрічається в тексті вхідної програми, компілятор має самостійно розпізнати його тип і виділити для його зберігання пам'ять відповідного розміру: байт, півслово, слово чи подвійне слово.

2.2.1. Числові літерали

Потреба розпізнавати тип за записом літералу призводить до певних умовностей: наприклад, у мові C літерали `10` і `010` різняться своїми значеннями, тому що вони позначають різні числа – 10 і 8. Водночас літерал `08` компілятор сприйме як помилку. Річ у тім, що за традицією, успадкованою від мови C, числові константи різняться за системами числення: десятковою, вісімковою та шістнадцятковою (хоча це стосується лише цілих констант). Двійковий уміст машинного слова простіше відтворити за шістнадцятковим записом числа: один байт відповідає двом шістнадцятковим цифрам, півслово – чотирьом, слово – восьми. Використання вісімкового коду зумовлене ще давнішою традицією: раніше було прийнято називати байтом 9 бітів, а це рівно три вісімкові цифри. Зазначимо, що при цьому знову йдеться про програмування на низькому рівні.

Цілі числа у вісімковому записі починаються нулем (математична традиція нехтування передніми нулями в програмуванні мовами C та C++ не діє). Це мабуть, пояснюється тим, що нуль нагадує літеру «O» (від ***oc**tal* – вісімковий, скорочено `oct`). Шістнадцятковий запис цілого числа починається символами `0x` (цифрою «0» і літерою «x», яка має нагадувати слово *he**x**adecimal* – шістнадцятковий, скорочено `hex`). Нижче наведено три різні записи числа 100:

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

Вісімковими та шістнадцятковими константами слід користуватися лише для типів без знака, щоб на комп'ютерах з *доповнювальним кодом* (complement-on-two), який зазвичай використовують для подання від'ємних чисел, не стикатися з несподіванками на кшталт значення `-1` константи `0xffff` (для 16-бітового слова) чи `0xffffffff` (для 32-бітового) (поясніть, чому).

Можна уточнити тип цілого літералу, додавши до його запису суфікс `U` для беззнакового, `L` – для довгого, `UL` – для беззнакового довгого типу; наприклад, записи `2147483648U` та `2147483648UL` позначають число 2^{31}. Спроба ж розглянути `2147483648` як число зі знаком, скажімо

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

може мати досить несподівані наслідки (поясніть, чому).

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

Вправа 2.4

Виведіть літерал π:

```
cout<<"pi="<<3.1415926535897932L<<endl;
```

Задайте значення точності виведення за допомогою функції `precision()` і повторіть виведення:

```
cout.precision(17);
cout<<"pi="<<3.1415926535897932L<<endl;
```

Поясніть результат.

А що станеться, якщо задати в числі π більше цифр, узявши, скажімо, 40 цифр після коми, – `3.1415926535897932384626433832795028841972L` (їх можна знайти, наприклад, у [22])? Результат не зміниться. Шанси зберегти 38 десяткових цифр з'являться лише тоді, коли мантиса розширеного діапазону займатиме більше чотирьох машинних слів. Справді, 16 байтів забезпечать майже 40 десяткових цифр: $2^{8\cdot16} = 2^{128} = 2^{130-2} = (2^{10})^{13} : 4 \approx (10^3)^{13} : 4 \approx 10^{38}$.

Отже, визначаючи числовий літерал, важливо пам'ятати про можливу точність.

2.2.2. Символьні літерали

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

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

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

Якщо символ задають його кодом, зокрема це стосується символів керування, то коду символу передує знак *зворотної скісної риски* (backslash) `'\'`. Символ може бути закодовано вісімковим або шістнадцятковим кодом. Наведемо приклади вісімкових кодів:

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

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

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

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

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

Як бачимо, деякі символи можна записати еквівалентними способами:

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

Далі наведено приклад виведення літералів у різних форматах.

Лістинг 2.9

```
// Формат виведення залежить
// від маніпулятора dec, oct або hex
cout<<"a number in decimal: "<<dec<<15<<endl;
cout<<"in octal: "<<oct<<15<<endl;
cout<<"in hexadecimal: "<<hex<<15<<endl;
cout<<"non-printing chararters(tab and escape): "<<'\t'<<'\33'<<endl;
```

Багатосимвольні літерали – *рядки символів* – записують у подвійних лапках. Вони зберігаються в пам'яті як послідовності символів, що закінчуються нульовим кодом `'\0'`. За такого способу кодування не потрібно пам'ятати довжину рядків: діставшись першого від початку нульового коду, потрапляємо в кінець рядка. Тому довжина кожного рядка на одиницю більша за кількість його символів. Отже, порожній рядок містить один символ – нульовий код:

```
""   // порожній рядок: код '\0'
" "  // рядок, що складається з пробілу '\x20' і коду '\0'
"A"  // рядок, складений з літери 'A' та коду '\0'
"Це константа, що складається з символів"
"This is another string"
"Наступний рядок містить службові символи"
"\nНомер\tНазва\tКількість\n"
```

2.3. Сталі та змінні

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

2.3.1. Визначення сталих і змінних

Окрім літералів, що іменують самі себе, можна використовувати *іменовані константи* (const variable), яким надають особливі імена у вигляді ідентифікаторів. Характерна особливість їх визначень порівняно з визначеннями змінних – наявність *специфікатора* (qualifier) сталої `const`.

Лістинг 2.10

```
const double pi=3.14159F;
const double nAvogadro=6.02214e23F;
const double dpi=3.141592653589793;
const char title[]="C++Programming Language";
```

Вправа 2.5

Передайте на компіляцію визначення

```
const double nAvogadro=6.02252e23;
```

Прокоментуйте застережне повідомлення компілятора. Зверніть увагу на визначення багатосимвольної константи `title`. Воно є визначенням масиву (див. підрозділ 2.8).

Правило гарного тону. *Під час складання програм будемо дотримуватися такої рекомендації: визначення змінної* має *(хоча загалом це необов'язково) містити її ініціалізацію відповідним початковим значенням.*

Розглянемо приклади ініціалізації.

Лістинг 2.11

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

2.3.2. Присвоєння

Не слід плутати поняття «ініціалізація» та «присвоєння». Із причин, які стануть зрозумілими трохи пізніше, для позначення ініціалізації використовують не тільки знак рівності (символ присвоєння), але й дужки. Використання дужок видається слушним, оскільки таке позначення менше нагадує присвоєння, ніж знак рівності.

Лістинг 2.12

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

Заміну значення змінної виконують зазвичай за допомогою *оператора присвоєння* (assignment operator).

Мова C++ успадкувала від C кілька варіантів присвоєнь:

- *просте присвоєння* (simple assignment), яке позначають знаком рівності «=»;
- *складене присвоєння* (compound assignment) – присвоєння, суміщене з операцією, позначеною двома символами, а саме знаком операції, за яким іде знак присвоєння, наприклад «+=», «&=», «/=».

У другому випадку змінна, якій присвоюють значення, водночас стає першим операндом відповідної операції, тобто фактично з'являється в присвоєнні як зліва, так і справа. Розглянемо приклади.

Лістинг 2.13

```
s=pi*r*r;
i=j+k; i+=k;   // рівнозначно i=i+k
i>>=k;         // рівнозначно i=i>>k
```

Зауваження. *Незважаючи на певну схожість, код ініціалізації*

```
int i=1;
```

принципово відрізняється від коду визначення змінної з наступним присвоєнням їй значення

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

У першому випадку одиниця записується до комірки пам'яті під час її призначення змінній, а в другому спочатку ця комірка спочатку містить якесь значення, а лише потім до неї буде записано одиницю. Як «поводиться» неініціалізована змінна, можна дізнатися, наприклад, у такий спосіб:

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

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

Зазвичай оператор присвоєння складається з двох частин. Права частина – вираз, значення якого буде обчислено та присвоєно змінній із лівої частини, тобто записано в певному місці пам'яті, відведеному для зберігання значень цієї змінної. Тому кожній змінній відповідає два значення – *іменувальне* й *іменоване*. Перше з них задає місце для змінної, а друге – саме значення, яке слід записати у відведеному для цієї змінної місці. У реалізації програми іменувальне значення фактично являє собою адресу в пам'яті. Його також називають *лівобічним значенням* змінної (lvalue, left value), тому що змінна набуває його, перебуваючи ліворуч від знака присвоєння. Іще одна його англомовна назва – *location value*, позаяк ідеться про місце для значень змінної.

Під час обчислення значення виразу з правої частини присвоєння використовують другий тип значень величин, що входять до нього, – іменовані значення. Їх називають також *правобічними* (rvalue, right value) за місцем в операторі присвоєння, хоча правобічні значення зустрічаються також і в інших контекстах (наприклад, у виразах для фактичних параметрів функцій).

Лістинг 2.14

```
// Контекст присвоєння
const double a=0.0, b=1.0;
int n=1;
double h=b-a;
// Присвоєння нового значення змінній
n=n*2;   // Обчислити rvalue(n)*2 та розмістити його в lvalue(n)
h=h/2;   // Обчислити rvalue(h)/2 та розмістити його в lvalue(h)
```

Цікаво, що спроби виконати присвоєння сталим величинам діагностуватиметься по-різному для літералів та іменованих констант, оскільки перші взагалі не мають лівобічного значення, а для других воно стале, непридатне для заміщення.

Як уже згадувалося, одна з типових помилок програміста мовою C полягає в уживанні знака присвоєння замість знака рівності. Якщо пере-

писати приклад з лістингу 2.8, помінявши місцями літерал і змінну, то компілятор відразу повідомить про помилку, спричинену тим, що немає лівобічного значення літералу.

Лістинг 2.15

```
int k=0;
if(0=k)
   cout<<"You never come here"<<endl;
else
   cout<<"Still k is equal "<<k<<endl;
```

Та сама спроба, але з використанням іменованої сталої, знову ж спричинить помилку, але тепер через сталість лівобічного значення.

Лістинг 2.16

```
const double dpi=3.141592653589793;
double x=0;
if(dpi=x);
```

Окрім присвоєння є й інші операції, які обробляють лівобічні значення; це, зокрема, унарні операції інкременту `++` й декременту `--`. Перша збільшує, а друга – зменшує значення свого аргументу на одиницю. Кожна з них має два варіанти – префіксний і постфіксний. У префіксному варіанті операції спочатку змінюється значення її аргументу, а потім виконується оператор, до якого вона входить. У постфіксному варіанті спочатку виконується оператор, а після цього змінюється значення аргументу. Наприклад, присвоєння `j=k++` спочатку змінює значення величини `j`, а потім – значення величини `k`.

Лістинг 2.17

```
// Нехай k==1
j=k++; // j==1; k==2
```

У префіксному ж варіанті `j=++k` спочатку змінюється значення величини `k`, а після того – значення величини `j`.

Лістинг 2.18

```
// нехай k==1
j=++k; // j==2; k==2
```

Вправа 2.6

Проаналізуйте вираз `(i=++(j=++k))++` і обчисліть значення всіх змінних, які він містить.

Зрозуміти механізми виконання операцій інкременту й декременту не так просто. Зазвичай їх використовують для компактного позначення зміни індексів (наприклад, у циклах). За інших умов захоплюватися ними

не варто, оскільки результат виконання операції може суттєво залежати від компілятора. Розгляньте головоломку з лістингу 2.19 і спробуйте відгадати, якими будуть виведені значення змінної m.

Лістинг 2.19

```
int i=100, m=100;
cout<<i<<':'<<i++<<"+1="<<i<<endl;
cout<<m<<':'<<m++<<"+1="; cout<<m<<endl;
m=2; cout<<m<<endl; cout<<m<<':'<<m++*m++<<endl; cout<<m<<endl;
m=2; cout<<m<<endl; cout<<m<<':'<<(++m)*(++m)<<endl; cout<<m<<endl;
```

Вправа 2.7

Відкомпілюйте та виконайте програму з лістингу 2.19. Поясніть одержані результати.

Зазначимо, що саме присвоєння теж вважається арифметичним виразом, як, наприклад, сума або частка. Як і будь-який інший вираз, воно має набувати правобічного значення, яким уважають значення його правої частини. Тепер воно «на правах виразу» може стати правою частиною іншого присвоєння. Виходячи з цього, вираз i=j=k трактують як присвоєння змінній i значення виразу j=k. Таке тлумачення відрізняється від звичайного розуміння порядку виконання операцій у виразах. Скажімо, у виразі a+b+c послідовність операцій додавання буде такою, ніби дужки стоять зліва – (a+b)+c: до суми перших двох доданків додається третій. У виразі i=j=k дужки начебто стоять справа – i=(j=k). Тому спочатку буде обчислено значення змінної k, потім його буде передано змінній j, а в завершення значення присвоєння j=k (у нашому випадку – k) стане значенням змінної i.

Присвоєння i=(j=e), де e – вираз підхожого типу, та послідовність присвоєнь

```
i=e; j=e;
```

можуть виявитися нерівнозначними. Як побачимо пізніше, значення виразу e може змінитися під час переходу від першого присвоєння до другого. А зараз, для самостійного ознайомлення, розгляньте такий приклад:

Лістинг 2.20

```
int i=0, j=0, k=0;
i=j=k++; cout<<i<<','<<j<<endl;
i=k++; j=k++; cout<<i<<','<<j<<endl;
```

Вправа 2.8

Що б могло визначати присвоєння (i=j)=k? Перевірте його коректність і результат виконання на комп’ютері.

Зауваження. *Присвоєння* `x=x` *інколи вважають синонімом порожнього оператора. Чи завжди воно безпечне? Для фундаментальних типів – так. Проте з оглядом на майбутнє зауважимо, що у разі програмованих типів обчислення лівобічного значення може призвести до руйнування правобічного (див. підрозділ 4.6.3).*

2.3.3. Перетворення типів

Тепер звернімося до проблеми, якої за строгої типізації не мало б бути, а саме до *перетворення типів*. Воно виникає, зокрема, у мішаних арифметичних виразах, що водночас містять об'єкти різних числових типів. Навіть у мовах програмування зі скромним набором числових типів (один цілий і один дійсний тип) виникає проблема обчислення мішаних виразів вигляду `k*x`, де `k` – ціле число, а `x` – дійсне. Оскільки результат має бути дійсний, то доцільно виконати перетворення величини `k` до дійсного типу та застосувати операцію множення дійсних чисел. До того ж навряд чи можна очікувати, що в наборі команд комп'ютера є арифметичні операції для всіх можливих комбінацій арифметичних типів. Тому для виконання будь-якої операції над числовими даними різних типів доводиться здійснювати потрібні зведення типів. Так, щоб додати коротке ціле число до довгого, його перетворюють на довге. Щоб помножити ціле число на дійсне, його слід перетворити на дійсне. Як виконати додавання короткого та довгого дійсних чисел? Спочатку необхідно зробити з короткого дійсного числа довге, потім визначити тип результату залежно від типу об'єкта, що його зберігатиме.

Так у програмі виникають *неявні перетворення типів* (implicit type conversion), про окремі з яких компілятор може попереджати, водночас залишаючи інші непоміченими. Строга типізація, узагалі кажучи, мала б заборонити використання мішаних виразів і, відповідно, неявних перетворень. Однак традиція перетворень так укоренилися, що типізації довелося поступитися. Із перетвореннями типів ми знову зустрінемося в об'єктному програмуванні.

У мові C++ діє концепція *розширення типу* (promotion), згідно з якою в разі змішування типів простіші перетворюються на складніші, які забезпечують ширший діапазон значень. У разі одночасного використання аргументів різних типів у виразах компілятор, де це потрібно, автоматично перетворює типи. Так, якщо арифметичні операції виконують над інтегральними типами (наприклад, `char` і **`short`**), вони розширюються до цілого типу **`int`**; коротші дійсні типи перетворюються на відповідні довші. Якщо йдеться про виконання присвоєння, то обчислений у правій частині результат перед зберіганням набуде типу лівобічного значення `lvalue`.

Окрім неявних можливі також явні *перетворення типів* (type cast) за допомогою кількох особливих операцій перетворення. Зокрема, мова C++ успадкувала від C дві «старомодні» операції: вирази `int(monday)` та `(int)monday` перетворюють константу `monday` типу `week` на значення цілого типу. Обернене перетворення – `(week)i` чи `week(i)`.

Зауваження. *Перетворення типів у програмах, узагалі кажучи, небажані – передусім тому, що вони не завжди коректні. Хотілося б принаймні мати можливість контролювати всі перетворення, які відбуваються в програмі. При цьому особливо небезпечними вважаються «старомодні» перетворення. Проблема в тому, що вони замасковані: немає іншого ефективного способу виявляти місця, де вони виконуються, окрім як перечитати всі тексти.*

Власні перетворення мова C++ виконує більш диференційовано за допомогою операцій **`const_cast`**, **`dynamic_cast`**, **`reinterpret_cast`** і **`static_cast`**, кожна з яких має спеціальне призначення. Останнє перетворення, наприклад `static_cast <int>(monday)`, діє так, як аналогічне перетворення в мові C. Його статичність означає, що всі потрібні перевірки виконуються на етапі компіляції. Це найнадійніше з перетворень. Призначення інших перетворень розглянемо пізніше.

Перевага явного позначення для операцій перетворення полягає в тому, що їх можна легко знайти в програмі, виконавши простий пошук за словом «_cast».

2.4. Указники

Програмування роботи з пам'яттю – найважливіша і разом з тим найнебезпечніша з погляду надійності частина C/C++-програм. Якщо до розглянутих раніше сталих і змінних можна застосувати термін *пряме адресування*, оскільки імена адресують значення безпосередньо (direct addressing), і відповідальність за зв'язок змінної з її адресою несе компілятор, то *указники* (pointers) роблять це *непрямо* (indirect addressing), а за коректність адреси відповідає сама програма. Якщо проаналізувати можливі джерела помилок при застосуванні прямої адресації порівняно з непрямою, то побачимо, що до помилок, викликаних неініціалізованими змінними чи указниками, будуть додані помилки обчислення адрес. Володіння технікою програмування указників – своєрідний «вищий пілотаж» у процедурному програмуванні.

2.4.1. Указник як засіб непрямого адресування

Щоб указник мав значення, попередньо потрібно надати йому це значення, зв'язавши указник з адресою певного місця пам'яті. Якщо воно

невідоме на момент визначення указника, то слід ініціалізувати його нулем. На рис. 2.1 наведено приклад визначення й ініціалізації указника, значення якого – нульова адреса.

```
double* px=0;
```

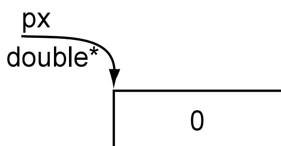


Рис. 2.1. Невизначений указник

Говорять, що такий указник на жодне місце в пам’яті не вказує, а тому його значення невизначене. Невизначений указник певною мірою безпечний, оскільки рівність нулеві можна перевірити. Він стає небезпечним (призводить до аварійного завершення програми) у разі спроби звернутися до значення, що міститься за нульовою адресою.

Невизначений указник не слід плутати із «засміченим» (теж у певному розумінні невизначеним або, точніше кажучи, визначеним некоректно). Якщо звичайну змінну було визначено без ініціалізації, наприклад так:

```
double x;
```

чого за правилами гарного тону слід уникати, то для неї відведено місце, до якого ще не потрапило змістовне значення: відповідна область пам’яті просто містить «сміття». Маємо коректне лівобічне та невизначене правобічне значення змінної. Так само незаданим або, інакше кажучи, «засміченим» буде значення неініціалізованого указника px. Однак у такому разі воно набагато небезпечніше, тому що може виявитись адресою області пам’яті, не призначеної чи навіть забороненої для використання за допомогою цього указника. Некоректним у такому разі стає лівобічне значення, а питання коректності правобічного за цієї умови вже навіть не виникає: наслідки можуть стати непередбачуваними (рис. 2.2). Указники, що ведуть «у нікуди», називають *завислими* (dangling pointer):

```
double* px;
```

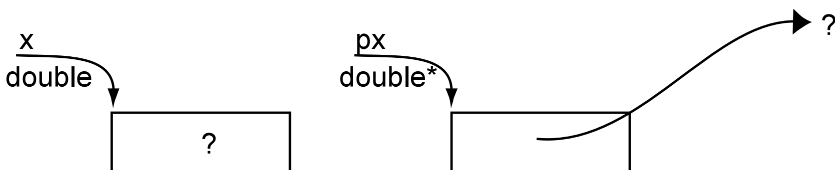


Рис. 2.2. Неініціалізований указник

Зауваження. *Розглядаючи техніку роботи з указниками, можна провести аналогію між структурами керування та структурами даних. Як уже було сказано під час вивчення парадигм, структуроване програмування повністю витіснило з практики вживання оператора переходу* `goto`*. Так само, як оператор переходу призводить до важко контрольованих передавань керування в програмі, використання указників може спричинити виникнення несистематизованих зв'язків між даними. Тому під час роботи з пам'яттю потрібно скрізь, де це можливо, надавати перевагу безпечнішим рішенням, наприклад відсилкам (підрозділ 2.5), особливо якщо мова йде про передавання параметрів. Проте це виходить за межі можливостей мови C. Тому, дотримуючись її правил, доводиться використовувати звичайні указники, особливо для обміну значеннями параметрів. При цьому варто дотримуватися дисципліни програмування, яка б виключала виникнення некоректних значень указників. Від багатьох помилок убезпечують найпростіші правила:* ***указник потрібно ініціалізувати в момент створення (невизначений указник ініціалізується нулем) і знову робити невизначеним у разі видалення значення, на яке він указував.*** *Далі (у підрозділі 5.5) будуть запропоновані безпечніші рішення, що ґрунтуються на використанні вдосконалених безпечних інтелектуальних указників.*

Указники, подібно до будь-яких інших змінних, мають значення, але це лише адреси, що вказують на самі дані. У попередньому прикладі указник `px` містить «сміття», а тому вказує невідомо на що. Важко уявити, до яких наслідків може призвести спроба записати дані за такою адресою. У кращому випадку це спричинить вихід за межі пам'яті, в гіршому – потрапляння до не призначеної для програми області й, можливо, ненавмисного псування життєво важливих даних.

Потрібно дотримуватися дисципліни безпечного програмування: якщо указник має не нульове значення, то з ним мають бути пов'язані два елементи пам'яті – коректна адреса та розміщене за нею значення!

Розглянемо приклад. Означимо дійсну змінну

```
double x=0.25;
```

і указник `px`, що вказує на її значення (рис. 2.3):

```
double* px=&x;
```

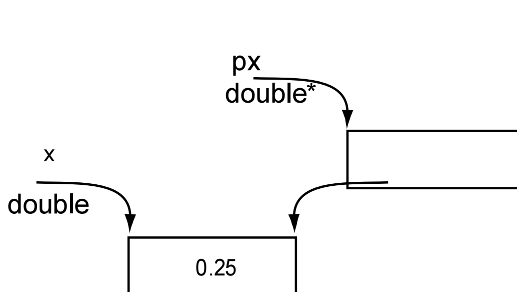


Рис. 2.3. Ініціалізований указник

Указник пов'язують зі змінною за допомогою *операції адресування* (address operator), яку позначають знаком `&`. Вираз `&x` слід читати як «адреса, за

якою розміщено значення змінної x». Інакше кажучи, це lvalue(x).

Операція адресування дає можливість записати *основну адресну тотожність*, що пов'язує лівобічне значення з правобічним. Читають її так: правобічне значення результату застосування операції адресування до змінної x дає її лівобічне значення:

```
rvalue(&x) == lvalue (x)
```

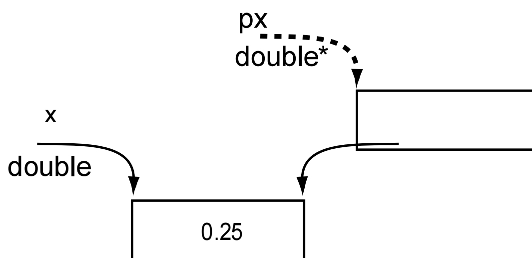


Рис. 2.4. Лівостороннє значення указника

Тепер lvalue(px) позначає місце, придатне для зберігання адрес дійсних величин. У нашому випадку це lvalue(x) (позначено пунктирною стрілкою на рис. 2.4) чи нульової адреси.

На місці lvalue(px), як і має бути, зберігається правобічне значення rvalue(px), яке своєю чергою виявляється лівобічним значенням lvalue(x) (їх позначено пунктирними стрілками на рис. 2.5). Справді,

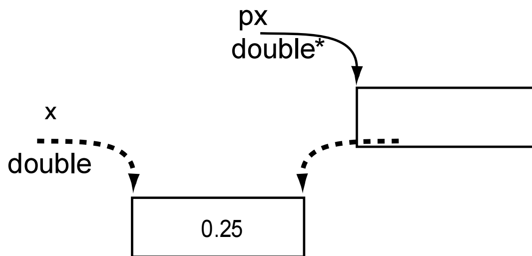


Рис. 2.5. Правостороннє значення указника

```
rvalue(px) == rvalue(&x)
```

відповідно до ініціалізації. Застосувавши основну адресну тотожність, маємо

```
rvalue(px) == lvalue(x)
```

Що можна робити зі значеннями указників? Передусім слід пам'ятати, що, хоча адреси пам'яті можна позначати цілими числами, значення указників *не належать* до жодного цілого типу. Вони дуже схожі на довгі цілі числа типу **unsigned long int**, проте компілятор не пропустить коду

Лістинг 2.21

```
unsigned long int longi;
double* p=new double(1.0);
longi=p;
```

повідомивши:

```
error C2440: '=' : cannot convert from 'double *' to 'long'
This conversion requires a reinterpret_cast, a C-style cast or function-
style cast
```

Причина такої реакції компілятора зрозуміла. Типи даних – це не просто множини, а структури, наділені операціями. До указників не можна застосовувати операції над цілими числами, наприклад множення.

Проте значення указника можна звести до типу **unsigned long**:

```
li=reinterpret_cast<long int>(p);
```

його можна навіть вивести (у шістнадцятковому вигляді):

```
cout<<li<<':'<<p<<endl;
```

Над значеннями указників виконують особливі (відмінні від застосовуваних для інтегральних типів) арифметичні операції, які утворюють так звану *адресну арифметику (address arithmetic)*. Докладніше її буде розглянуто в підрозділі 3.8.

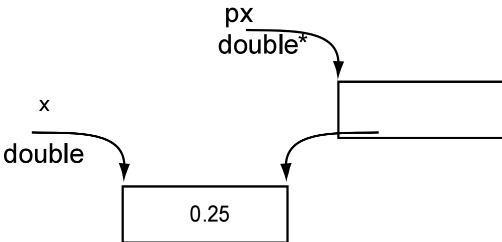


Рис. 2.6. Правостороннє значення змінної

Залишилося зрозуміти, як скористатися указником px для того, щоб добратися до значення змінної x, тобто до rvalue(x)? Для цього є ще одна операція над адресами – *операція розіменування (dereferencing operator)*, яку позначають «*». Тепер маємо ще одну залежність:

```
rvalue(*px) == rvalue(x)
```

По суті, операції адресування та розіменування взаємно обернені.

Лістинг 2.22

```
&(*px) == px;
px == *(&px);
```

Змінити значення x можна тепер одним із двох способів: або присвоєнням змінній x, або, що те саме, присвоєнням розіменованому px:

```
*px=0.5;   // те саме, що x=0.5;
```

У розглянутому прикладі указник було встановлено на значення змінної x. Ніщо не забороняє перевстановити його на іншу змінну, наприклад:

Лістинг 2.23

```
double y=1.6;
px=&y;
```

Тепер указник px відсилатиме до змінної y, а не x (рис. 2.7), тобто

```
rvalue(px) == lvalue(y).
```

Проінтерпретуємо у використовуваних позначеннях невизначені указники. Якщо є визначення

```
double* px=0;
```

то існують як lvalue (px), так і rvalue (px), але

```
rvalue (px) == 0
```

і, відповідно,

```
lvalue (*px) == 0,
```

тому rvalue(*px) не існує.

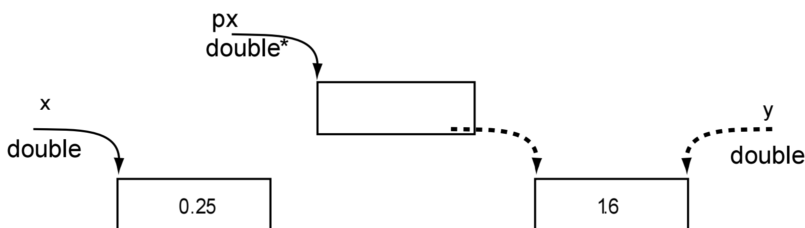


Рис. 2.7. Перевстановлений указник

2.4.2. Указники та динамічна пам'ять

У попередніх прикладах було показано, як пов'язувати указники (імена другого рівня) зі звичайними іменами. Але насправді важливим для указника є наявність не зовнішнього імені – ідентифікатора, а місця для розміщення даних, яке називають лівостороннім значенням. У разі визначення імен змінних це місце виділяє компілятор. Інший спосіб виділення області пам'яті полягає в застосуванні операції **new** T, де T – тип даних. За її допомогою (тут ця операція має вигляд нульарної) виділяється область пам'яті для значення вибраного типу T. Адресу цієї області указнику буде присво-

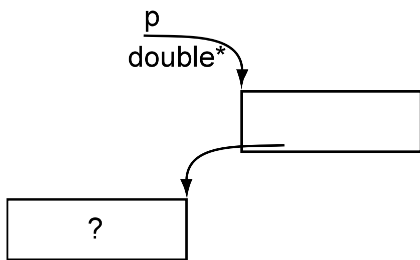


Рис. 2.8. Неініціалізоване значення указника

єно, наприклад, під час його ініціалізації чи присвоєння. Результат ініціалізації

```
double* p=new double;
```

наведено на рис. 2.8.

Як і раніше, знову маємо набір ліво- та правобічних значень, пов'язаних із указником `p`, до того ж визначено як `lvalue(p)`, так і `rvalue(p)`. Зазначимо, що `lvalue(*p) == rvalue(p)` залишилося без зовнішнього ідентифікатора, але його можна позначати за допомогою виразу `*p`. Гірше те, що значення `rvalue(rvalue(p)) == rvalue(*p)`, яке, власне, і міститься у виділеній області пам'яті, не визначено. Тому такий спосіб виділення пам'яті вважають не зовсім удалим, оскільки створений елемент `*p` містить «сміття». Тому, послідовно дотримуючись правила гарного тону з підрозділу 2.3.1, його слід ініціалізувати безпосередньо під час створення. Для цього використовують унарний варіант операції виділення пам'яті, параметром якого є значення, призначене для `rvalue(*p)`:

```
double* p=new double(0.333333);
```

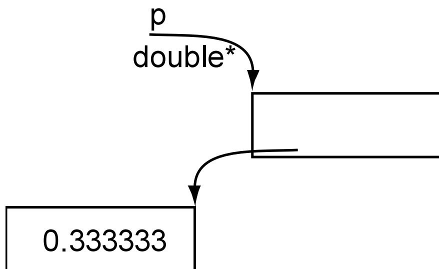


Рис. 2.9. Повністю ініціалізований указник

Результат зазначених дій наведено на рис. 2.9.

Об'єкти, створені розглянутим способом, утворюють так звану динамічну пам'ять програми. Як за створення, так і за видалення її елементів відповідає сама програма. Команду видалення даних (звільнення пам'яті), симетричну до команди виділення пам'яті **`new`**, називають **`delete`**. Її особливість полягає в тому, що вона повертає пам'ять операційній системі, але не змінює значення указника: він і надалі вказує на область пам'яті, яка йому більше не належить, а значить, знову ж таки містить «сміття». Цю пам'ять система може виділити комусь іншому. Недбало складена програма може не помітити цього, продовжую-

чи використовувати вже не чинний указник, а ще гірше – змінювати область пам'яті, як їй не належить.

Дуже повчально виконати наведений далі фрагмент програми та проаналізувати отримані результати.

Лістинг 2.24

```
int* pint=new int(1);
cout<<pint<<':'<<*pint<<endl;   // Справжнє значення *pint
delete pint; cout<<pint<<':'<<*pint<<endl;   // «Сміття»
int* fint=new int(3);
cout<<fint<<':'<<*fint<<endl;   // Справжнє значення *fint
*pint=2; // Використання нечинного указника: втручання в чужу пам'ять
cout<<pint<<':'<<*pint<<endl;   // Незаконно створене значення *pint
cout<<fint<<':'<<*fint<<endl;   // Зіпсоване значення *fint
```

Правило гарного тону. *Якщо ви виділили динамічну пам'ять за допомогою команди* ***new****, то не забудьте своєчасно звільнити її командою* ***delete*** *і самостійно обнулити указник. Звільнення запобігає втратам пам'яті (memory leak), а обнулення (розозначення) указника – спробам використати звільнену пам'ять.*

Лістинг 2.25

```
double* p=new double(0.333333);
// Можна робити все, що потрібно
...........................
delete p; p=0;
// Тепер указник невизначений: він більше не вказує на елемент,
// пам'яті що йому не належить; спробу вжити вираз *p буде покарано
// аварійним завершенням програми
```

Вправа 2.9

Виконайте та проаналізуйте такі тексти програм.

Лістинг 2.26

```
// 1. Некоректний сценарій
double* pd=new double(5.2); // Ініціалізуємо указник і елемент пам'яті
cout<<pd<<endl;       // 0x004419B0
cout<<*pd<<endl;      //5.2
delete pd;   // Пам'ять звільнено, значення указника не змінилося
cout<<pd<<endl;       // 0x004419B0, але він указує на «сміття»
cout<<*pd<<endl;      // -1.45682e+144: «сміття»
double* new_pd=new double; // Ініціалізуємо інший указник
// Йому дістається та сама область пам'яті
cout<<new_pd<<endl;   // 0x004419B0, але «сміття» змінилося
cout<<*pd<<endl;      // -6.27744t+066: «сміття»
cout<<*new_pd<<endl;
```

```
// 2. Коректний сценарій                                    Лістинг 2.27
double* pd=new double(5.2); // Ініціалізуємо указник і елемент пам'яті
cout<<pd<<endl;      // 0x004419B0
cout<<*pd<<endl;     // 5.2
delete pd;   // Звільняємо пам'ять
pd=0;        // Правило гарного тону: обнулюємо указник. Тепер спробу
// скористатися невизначеним указником буде діагностовано як помилку
cout<<*pd<<endl;      // Помилка етапу виконання!!!
```

2.4.3. Указники констант

Тепер зауважимо, що крім змінних у програмах використовують також величини двох типів: літерали й іменовані константи. Чи може адреса сталої бути значенням указника? Якщо так, то доступ до пам'яті за допомогою указників створює проблему захисту констант. Справді, указник забезпечує альтернативний доступ до області пам'яті. Якби звичайний указник, наприклад px, можна було встановити на константу, то присвоєння розіменованому указнику *px нового значення призвело б до спотворення константи. На щастя, мовою передбачено відповідний захист. Наприклад, компілятор не пропустить такого тексту.

```
const double pi=3.14159F;                                   Лістинг 2.28
double* ppi=&pi;
*ppi=0.2;   // Error: змінилося б значення константи
```

Після його компіляції буде виведено приблизно таке повідомлення про помилку, виявлену в другому рядку тексту:

```
error C2440: 'initializing' :
cannot convert from 'const double *' to 'double *'
```

Щоб уникнути проблеми несанкціонованого доступу до констант, мова C++ точніше розрізняє типи, застосовуючи особливі указники для сталих об'єктів. Для адресування констант існує особливий тип указника, а саме *указник константи* (pointer to constant).

```
const double* cpx=&pi;   // Указник константи
```

У такому разі cpx стає указником на дійсну константу. Його розіменування можна використовувати лише як правобічне значення. Тому стає неможливим змінити значення *cpx, а отже, ненавмисно змінити константу pi (рис. 2.10).

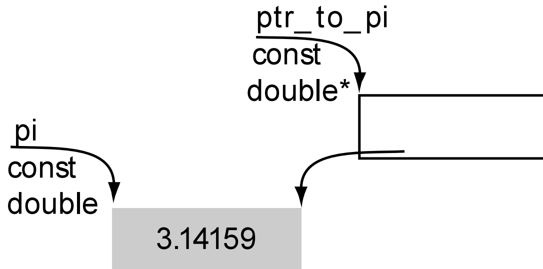


Рис. 2.10. Указник на константу

Для сталих указників виконуються основні адресні тотожності, але без права заміни rvalue(*cpx). Це цілком відповідає семантиці константи, адресованої указником:

```
rvalue(cpx) ==
   lvalue(pi);
rvalue(*cpx) ==
   rvalue (pi);
```

Зауваження. *Адресування указником літералів неможливе взагалі, оскільки для літералів не визначено лівобічне значення, потрібне для встановлення указника.*

Зазначимо, що указник константи може вказувати і на змінну:

Лістинг 2.29

```
double x=0;
const double* cpx=&x;
```

Чи коректне в цьому разі використання указника константи? Знову маємо

Лістинг 2.30

```
rvalue(cpx) == lvalue(x);
rvalue(*cpx) == rvalue(x);
```

Однак тепер можна змінити значення rvalue (x) змінної x, унаслідок чого зміниться значення вмісту rvalue(*cpx) указника, наведеного на константу *cpx.

Лістинг 2.31

```
cout<<*cpx<<endl;        // Буде виведено 0
x=1; cout<<*cpx<<endl;   // Буде виведено 1
```

Ситуацію, що склалася, інтерпретуємо так. «Власник» області пам'яті – сама змінна x, і вона має право змінювати значення. Указнику константи делеговано частину прав доступу: за його допомогою можна бачити, але не можна змінювати значення rvalue(*cpx). Завдяки таким указникам не можна змінювати значення об'єкта за допомогою альтернативного доступу до нього із застосуванням операції розіменування. Справді, спроба виконати присвоєння

```
*cpx=1;
```

призведе до помилки

```
error C2166: l-value specifies const object
```

Отже, указник константи має *обмежені права доступу* до області пам'яті. Однак сам власник цієї області використовує її відповідно до наданих йому прав. Таке рішення цілком виправдане. Доступ до об'єктів лише за допомогою указників на константи спрощує процедуру контролю за зміненням значень об'єкта.

Утім у указника константи все ще залишається один «степінь вільності». Наприклад, указник константи `cppi` жорстко не пов'язаний зі своєю константою:

Лістинг 2.32

```
const double pi=3.14159F;
const double* cppi=&pi;
```

Значення `rvalue(cppi)` можна змінити, переспрямувавши указник на інше місце в пам'яті – наприклад, присвоївши йому адресу іншої константи чи навіть змінної:

Лістинг 2.33

```
const double nAvogadro= 6.0221412927e23;
cppi=&nAvogadro;
```

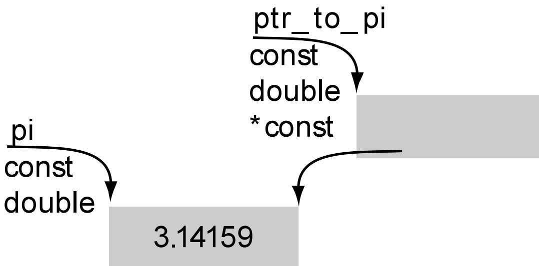


Рис. 2.11. Сталий указник на константу

Щоб `cppi` та `pi` ніколи не могли бути роз'єднані, `cppi` слід визначити ще точніше – як *сталий указник на константу* (constant pointer to constant) (рис. 2.11):

```
const double* const
cpcpi=&pi; // Сталий
// указник на константу
```

Тепер `cpcpi` та `pi` пов'язані один з одним нерозривно. Спроба змінити значення `rvalue(cpcpi)` за допомогою присвоєння

```
cpcpi=&nAvogadro;
```

призведе до виведення вже відомого повідомлення

```
error C2166: l-value specifies const object
```

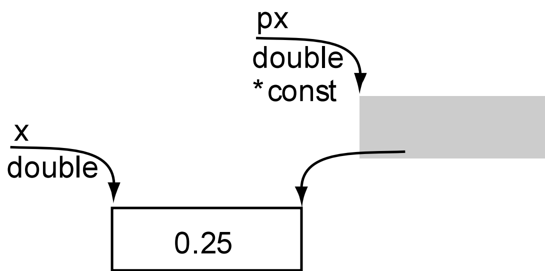


Рис. 2.12. Сталий указник

Нерозривно пов'язати з об'єктом можна указник не лише на сталу, а й на змінну. З цією метою використовують *сталі указники* (constant pointer) (рис. 2.12). Особливо важливе значення вони мають в об'єктному програмуванні, де їх застосовують як указники на поточний об'єкт **this** (підрозділ 4.5).

```
double* const px=&x; // Сталий указник на змінну
```

Значення x і *px, як і раніше, можна довільно змінювати, але спрямувати на інше місце сам указник px неможливо.

2.4.4. Типізація указників

Як бачите, кожен указник не лише вказує на певне місце в пам'яті, але й приписує тип об'єкту, що там зберігається. Враховуючи те, що ціле значення можна неявним перетворенням звести до дійсного, дійсне – до довгого дійсного тощо, поставимо собі запитання: як перетворюються типи указників? Скажімо, як звести указник на ціле число до указника на дійсне? Відповідь виявляється на диво простою: ніяк! Неявні перетворення типів указників заборонено. Так, скажімо, коректне перетворення, наведене в лістингу 2.34, стає недопустимим і некоректним у лістингу 2.35.

Лістинг 2.34

```
int in=10; double fl=in;
cout<<fl<<endl; // Буде виведено 10
```

Лістинг 2.35

```
int* pInt=&in;
double* pf=&fl; pf=pInt;
```

Буде виведено повідомлення про помилку:

```
error C2440: '=' : cannot convert from 'int* ' to 'double* '
```

Перш ніж пояснити, чому перетворення типів указників недопустимі, опишемо два винятки: ініціалізацію указника нулем (або присвоєння йому нульового значення) і *безтипові указники* (void pointer).

Як уже було сказано, ініціалізація указника нулем означає, що указник не визначено, тобто він не вказує на жоден об'єкт:

```
double* px=0;
```

Правила гарного тону настійливо рекомендують перетворювати указник на невизначений, тобто обнулювати після видалення пов'язаного з ним елемента пам'яті:

Лістинг 2.36

```
delete px; px=0;
```

Розглянемо операцію виділення пам'яті **new**. Коли вільної пам'яті немає, вона дає один із двох результатів. Стандартна реакція системи полягає у створенні аварійної ситуації bad_alloc (обробку таких ситуацій розглянемо пізніше, в підрозділі 2.7.2). Однак можна й самостійно перевіряти значення указника, який дорівнюватиме нулю, якщо виділення пам'яті неможливе. У такому разі операцію **new** застосовують у форматі **new**(nothrow), де аргумент nothrow – це об'єкт простору імен std, означений у заголовному файлі <new>:

Лістинг 2.37

```
#include <new>
double* px=new(std::nothrow)double(1.0);
if(px==0) {   // Вільної пам'яті немає
   cout<<"Вільної пам'яті немає"<<endl;
   ……………………………….
}
```

Щоб виявити факт переповнення пам'яті, потрібно створити достатньо великий запит – наприклад, такий:

Лістинг 2.38

```
#include <new>
double* px;
for(int i=0; ;++i) {   // Повторюємо, доки вдається
// Розміщуємо 134217727 дійсних значень
   px=new(std::nothrow)double [0x7ffffff];
   if(px==0) {   // Вільної пам'яті немає
      cout<<"Вільної пам'яті немає"<<endl;
      break;
   }
}
```

Нульове значення указника – певне порушення типізації, оскільки нуль формально не належить до типу **double***, а операцію розіменування нульового указника не визначено.

Інше порушення типізації є набагато небезпечнішим. Справа в тому, що мова C містить так звані безтипові указники **void***, а C++ для досягнення сумісності допускає їх. Це знову ж таки данина традиціям безтипової мови C, такий собі «хакерський діалект» у програмуванні.

Безтиповий указник (void pointer) указує на об'єкт, тип якого не відомий. До стандартних типів даних було додано тип **void**, множина значень якого порожня. Змінних типу **void** не буває, констант – теж, але є указник типу **void***, який може набувати значень указників будь-яких інших типів:

Лістинг 2.39

```
void* voidPtr;
int* intPtr=new int(10);
voidPtr=intPtr;   // OK
```

Важливо! *Безтиповий указник* `voidPtr` *не можна розіменувати:*

```
cout<<*voidPtr<<endl;    // Error
```

бо невідомий тип значення, що зберігається за його адресою. Як зараз побачимо, інтерпретація значень указників суттєво залежить від їхніх типів.

Безтиповий указник не можна безпосерендньо присвоїти звичайному типізованому, наприклад указнику на ціле:

```
intPtr=voidPtr;   // Error
```

адже немає гарантії, що безтиповому указнику було свого часу виділено чотири байти пам'яті, до яких може бути записане ціле число. Програма сама несе відповідальність за подібне присвоєння, а тому потрібне явне перетворення типу:

```
intPtr=static_cast<int*>(voidPtr);
```

Зауваження. *Нерідко виникає запитання, для чого в операторі виділення пам'яті*

```
double* doublePtr=new double (10);
```

повторювати тип указника, відомий з його визначення. Дійсно, спроба виконати щось на зразок

```
double* doublePtr=new int (10);
```

завершується діагностуванням помилки. Проте застосування безтипових указників свідчить про те, що тип указника та тип його значення можуть не збігатися. Тому для ініціалізації безтипового указника потрібно явно визначити тип виділеної пам'яті:

```
void* voidPtr=new int(10);
```

Є й вагоміша мотивація такого синтаксису, і ми повернемося до неї у підрозділі 7.9.

Вправа 2.10

Розгляньте приклад з дуже несподіваним результатом. Виконайте його на комп'ютері та поясніть.

Лістинг 2.40

```
void* voidPtr;
int* intPtr=new int(10);
double* dPtr=new double(10); voidPtr=dPtr;
intPtr=static_cast<int*>(voidPtr); cout<<*intPtr<<endl;
```

Загалом, використання безтипових указників – невдала традиція, знову ж таки успадкована від мови С. Застосування методів поліморфізму в об'єктно-орієнтованому програмуванні дасть нам змогу в майбутньому складати надійні програми з використанням указників наперед невідомих типів.

Тепер подивимося, до чого призводять спроби змінити типи повноцінних типізованих указників. Спочатку розглянемо указник на рядок символів:

```
char* str="\nabcdefghij";
```

В пам'яті він відображається як

0A	61	62	63	64	65	66	67	68	69	6A	00

На початок текстового рядка можна спрямувати безтиповий указник, незважаючи на те, якого типу значення було до цього:

```
void* voidPtr=str;
```

Тепер установимо на цю область пам'яті указник на коротке ціле число:

```
short int* pShInt=static_cast<short int*>(voidPtr);
```

Він вказуватиме на перші два байти:

0A	61	62	63	64	65	66	67	68	69	6A	00

Вивівши його значення, одержимо:

```
cout<<*pShInt<<endl; // Буде виведено 24842
```

Спрямуємо на це саме місце указник на ціле число:

```
int* pInt=static_cast<int*>(voidPtr);
```

Він вказуватиме на чотири байти:

0A	61	62	63	64	65	66	67	68	69	6A	00

Не дивно, що значення стане іншим:

```
cout<<*pInt<<endl; // Буде виведено 1667391754
```

Тепер створимо указник на дійсне число:

```
float* pf=static_cast<float*>(voidPtr);
```

Хоча він вказуватиме на ті самі чотири байти, число виявиться зовсім не схожим на його цілочислову інтерпретацію: різниця між ними становитиме 10 порядків

```
cout<<*pf<<endl; // Буде виведено 4.17596e+021
```

Якщо ж узяти указник на дійсне число з подвійною точністю

```
double* pdp=static_cast<double*>(voidPtr);
```

то результат виявиться ще цікавішим – `124732e+190`.

Вправа 2.11

Поясніть результати виконання наведеного нижче коду

```
char* str="\nabcdefghij"; void* voidPtr=str;
short int* pShInt=static_cast<short int*>(voidPtr);
cout<<*pShInt<<endl; // Буде виведено 24842
int* pInt=static_cast<int*>(voidPtr);
cout<<*pInt<<endl;   // Буде виведено 1667391754
double* pf=static_cast<double*>(voidPtr);
cout<<*pf<<endl;     // Буде виведено 4.17596e+021
double* pdp=static_cast<double*>(voidPtr);
cout<<*pdp<<endl;    // Буде виведено 124732e+190;
```

Наведені приклади пояснюють, чому заборонено перетворювати типи указників: залежно від типу указника один і той самий код набуває різної інтерпретації, а тому для обчислення значення виразу `rvalue(*px)` потрібна інформація про тип указника `px`.

2.4.5. Операції над указниками

Завершимо підрозділ оглядом *операцій над указниками*. Виходитимемо з того, що кожен тип указників (на цілі, дійсні числа тощо) особливий; над кожним із них визначено три види операцій:

- порівняння;
- присвоєння;
- збільшення чи зменшення.

Візьмемо два однотипних указники:

```
double* p1=new double(1), *p2=new double(2);
```

Їх можна порівнювати, наприклад `p1==p2, p1 > p2` тощо:

Лістинг 2.41

```
if(p2>p1)
   cout<<"p2>p1"<<endl;
else
   cout<<"p1>p2"<<endl;
```

Різнотипні указники порівнювати не можна:

Лістинг 2.42

```
int* pI=new int(1);
cout<<(p1<pI)<<endl;   // Error
```

але будь-який указник можна порівняти з безтиповим.

Над указниками можна виконувати операції присвоєння:

Лістинг 2.43

```
p1=p2; p1=&x;
p1=new double(3);
```

Зрештою, над указниками виконують арифметичні операції. До них належать інкремент і декремент, тобто збільшення (зменшення) на один елемент пам'яті, а також операції збільшення (зменшення) на довільне ціле число елементів пам'яті. Однак до безтипових указників ці операції застосовувати не можна.

Лістинг 2.44

```
p1++; ++p1; p1--; --p1; p1+=10;
cout<<p2<<endl;      // 0x004418F0
cout<<p2+1<<endl;    // 0x004418F4
```

Важливо! *Зверніть увагу, що значення указника збільшується (зменшується) не на один байт, а на стільки байтів, скільки їх потрібно для зберігання елемента даних відповідного типу.*

Робота з указниками часто призводить до виникнення у програмах дуже глибоко прихованих помилок. Ще складніше впоратися з указниками на указники, використання яких – поширена техніка програмування в «чистій» мові C. При цьому ускладнюються проблеми керування пам'яттю, адже доводиться стежити за обома «поверхами» указників. Використовувати їх слід іще рідше й обережніше, ніж звичайні.

Як застосовується «двохповерховий» указник, показано далі.

Лістинг 2.45

```
// Виділяємо пам'ять, потрібну для указника
double** ppd=new double*;
// Виділяємо пам'ять для числа й ініціалізуємо його та указник
*ppd=new double(5);
cout<<ppd<<endl; cout<<*ppd<<endl; cout<<**ppd<<endl;
```

2.5. Відсилки

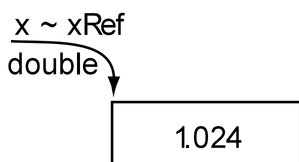


Рис. 2.13.

Альтернативний спосіб доступу до даних – використання конструкції *відсилки*, або *псевдоніма* (reference). Відсилкою називають альтернативне ім'я іменованого об'єкта, яке жорстко пов'язують із цим об'єктом без можливості переозначення, на відміну від указника, значення якого можна перевстановлювати будь-яку кількість разів. Означають псевдонім так (рис. 2.13):

Лістинг 2.46

```
double x=1.024;    // Визначення змінної x
double& xRef=x;    // Визначення її псевдоніма
```

Тепер один і той самий елемент пам'яті стає доступним за двома різними іменами – `x` та `xRef`:

Лістинг 2.47

```
cout<<x<<endl;
x+=20; cout<<xRef<<endl;
xRef+=300; cout<<x<<endl;
```

Яке призначення мають відсилки? Здебільшого їх використовують для передавання параметрів. Однак вичерпну відповідь на це запитання буде надано в наступному розділі. Поки ж що обмежимося зауваженням, що вони дають змогу іменувати одну й ту саму область пам'яті в різних частинах програми, наприклад, підпрограмах, зручними та зрозумілими для

авторів цих частин іменами, що особливо важливо у випадку колективного програмування.

Важливо! *Відсилка являє собою потужний засіб передавання параметрів у функції.*

Порівняємо відсилки з іншими конструкціями. Спочатку розглянемо звичайне копіювання значення змінної:

Лістинг 2.48

```
double x=1.024;      // Визначення змінної x
double xCopy=x;      // Створення її копії
xCopy+=xCopy;        // Подвоєння значення копії; оригінал не змінився
```

Отже, після ініціалізації змінна та її копія стають незалежними; кожна позначає свою область пам'яті, і зміна значення однієї з них не впливає на іншу (рис. 2.14).

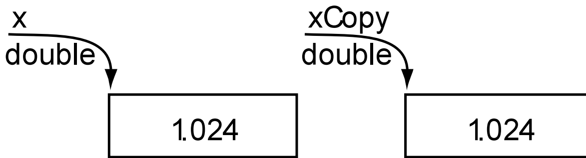


Рис. 2.14. Результат копіювання

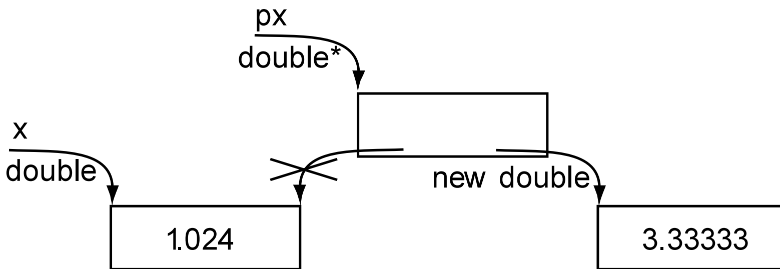


Рис. 2.15. Перевизначений указник.

Розглянемо іншу пару – змінну та указник:

Лістинг 2.49

```
double x=1.024F;     // Визначення змінної x
double* px=&x;       // Визначення указника на неї
```

Змінення значення змінної x є одночасно зміненням значення, на яке вказує указник px і навпаки доти, доки між ними, тобто указником і змін-

ною, не буде розірвано зв'язок – наприклад, шляхом перевстановлення указника (рис. 2.15):

```
px=new double(3.33333); // x==1,024; *px==3.33333
```

Як уже було зазначено, указник можна жорстко пов'язати зі змінною, оголосивши його сталим:

Лістинг 2.50

```
double x=1.024;          // Визначення змінної x
double* const px=&x;     // Сталий указник на неї
```

Тепер зв'язок «указник–змінна» став нерозривним. Так само нерозривним залишається зв'язок змінної з відсилкою:

```
double& rx=x;    // Псевдонім змінної x
```

Сталий указник більше, ніж інші указники, схожий на відсилку, але між ними існують важливі відмінності.

По-перше, це спосіб ініціалізації. Псевдонім ініціалізують безпосередньо значенням самої змінної, а указник – її адресою. Визначення сталого указника неможливе без ініціалізації (перевірте!).

По-друге, указник, навіть сталий, фактично має два значення: перше значення – це адреса в пам'яті, друге – елемент даних. Від одного значення до іншого переходять за допомогою операції розіменування; псевдонім же являє собою просто інше ім'я змінної, а відтак не потребує розіменувань.

Ще більша відмінність існує між відсилкою та звичайним указником. Якщо ініціалізація указника під час визначення залишається лише правилом гарного тону, то ініціалізація відсилки – це норма. Тому відсилка – надійніший інструмент, ніж указник.

Як і для указників, існують псевдоніми сталих (сталі відсилки):

Лістинг 2.51

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

І знову ж таки, як і в разі указників на сталі, стала відсилка може вказувати як на сталу, так і на змінну:

Лістинг 2.52

```
double x=1.23456;
const double& crx=x;
```

Тому зрозуміло, що, подібно до указника, стала відсилка має обмежені права доступу до свого оригіналу. Спробу змінити значення сталої відсилки

```
crx=2.34567;   // Error
```

буде діагностовано як помилку:

```
error C2166: l-value specifies const object
```

Оскільки мова C, на жаль, не має апарату посилань, єдиний засіб альтернативного іменування в C-програмах полягає у застосуванні указників. Переносячи програму з мови C в середовище C++, слід передусім зосередитися на можливості заміни указників відсилками. На відміну від указників, відсилки не бувають завислими, що значно збільшує надійність програм. Щоправда, одночасне використання указників і посилань може призвести до появи завислої відсилки, що демонструє наступний приклад. Але винен у цьому указник, а не відсилка.

Вправа 2.12

Розгляньте лістинг 2.53. Поясніть, що буде виведено в кожному з операторів виведення.

Лістинг 2.53

```
double* smth=new double(0.1234);
double& rsmth=*smth; cout<<rsmth<<endl;
delete smth; smth=0; cout<<rsmth<<endl;
```

2.6. Інструкції

Мові C++ властивий традиційний набір інструкцій (операторів) структурного програмування, успадкованих від C. Інструкції бувають прості та складені.

2.6.1. Прості інструкції

Розглянемо прості інструкції. Найпростіша серед них – *порожня* (null statement). Вона ніяк не позначається та нічого не виконує. Зазвичай інструкція закінчується крапкою з комою, а порожня складається лише з крапки з комою.

Лістинг 2.54

```
int k=0;
while(++k>0)
   /* Порожній оператор */;
cout<<k<<endl;   // Буде виведено 2147483648. Чому?
```

Інші прості інструкції – *операторні вирази* (expression statement). Це звичайні вирази, побудовані із сталих і змінних за допомогою операцій, але вони закінчуються крапкою з комою і тому стають інструкціями. Ви-

конання інструкції полягає в обчисленні значення цього виразу. Обчислення звичайно супроводжується зміною стану пам'яті, хоча це і не обов'язково. Типовими операціями зміни стану пам'яті є присвоєння (зауважимо, що інструкція присвоєння – це також варіант операторного виразу).

Лістинг 2.55

```
double x; double a=1.0;
x; a+x; x=x+a; x+=a;
int i=0, j=0;
i++; i++, j++; cout<<"Hallo, world!"<<endl;
```

І нарешті, будь-яка інструкція чи послідовність інструкцій стає простою, якщо її перетворити на *блок*, помістивши у фігурні дужки. Допустимі також порожні блоки:

Лістинг 2.56

```
void nothing() {
   // Порожній блок
}
void notswap(int x, int y) {
// Ще один блок
   int t=x; x=y; y=t;
}
```

2.6.2. Структури керування

Стандартний набір структур керування складається з послідовних дій, розгалуження (зокрема кратного) та циклів. За браком місця ми не розглядатимемо окремо кожну з них, відсилаючи читача до підручників з C і C++, обмежившись кількома демонстраційними прикладами.

Лістинг 2.57

```
// Приклад розгалуження
if(n%2==0) {n/=2; x*=x;}
else {n--; y*=x;}
```

Зауваження. *За «хакерською» традицією мови C (краще її* не *дотримуватись, але варто хоча б знати, щоб уміти правильно читати «старомодні» програми) умову* `(n%2==0)` *прихильники чистого C записують у вигляді* `(!n%2)`*, а умову* `(n%2!=0)` *– як* `(n%2)`*. Це ідіоми програмування, типові для мови C. Їх корисно знати, але не варто наслідувати, оскільки програми стають складнішими для розуміння. Але в C-програмах часто можна зустріти щось на кшталт*

Лістинг 2.58

```
if(n%2) {n--; y*=x;}
else {n/=2; x*=x;}
```

Лістинг 2.59

```
// Приклад кратного розгалуження:
// лічильник символів a та b
int aCounter=0, bCounter=0; char c=' ';
do {
   cout<<"Press a or b"<<endl; cin>>c;
   switch(c) {
   case 'a':
      ++aCounter;   // Збільшуємо значення лічильника a
      cout<<"you say a"<<endl;
      break;
   case 'b':
      ++bCounter;   // Збільшуємо значення лічильника b
      cout<<"you say b"<<endl;
      break;
   default:
      cout<<"you do not press a or b"<<endl;
   }
   cout<<"Repeat?(y/n)"; cin>>c;
} while(c=='y');
// Скільки літер a та b?
cout<<"Character a has been pressed "<<aCounter<<" times;"<<endl;
cout<<"Character b has been pressed "<<bCounter<<" times."<<endl;
```

На особливу увагу в кратних розгалуженнях заслуговує остання альтернатива перемикача, позначена як **default**. Вона не містить значення для порівняння й буде застосована в разі непридатності жодної з попередніх альтернатив. Власне кажучи, альтернатива **default** необов’язкова, однак правила гарного тону рекомендують *завжди* включати її до кратного розгалуження.

Програмування кратних розгалужень потребує певної дисципліни. Спочатку потрібно проаналізувати всі можливі значення перемикача та розмістити їх в альтернативах. Ситуацію, коли жодна з альтернатив не справдилася, слід розглядати як помилкову, обробка цього випадку записується останньою альтернативою перемикача, яка одержує статус альтернативи за замовчуванням **default**. Альтернативу **default** використовують здебільшого для виявлення помилкових значень у даних. Якщо її немає, то неможливо контролювати випадки (можливо, помилкові), коли жодна альтернатива не виконується. У нашому прикладі це значення, відмінні від `'a'` та `'b'`. Якби альтернативи **default** не було, програма не виявила б некоректного значення.

Головною структурою керування в імперативному програмуванні служить цикл. Як часто повторював Віктор Михайлович Глушков, без циклів комп'ютери були б непотрібні. Адже хто встигав би з такою швидкістю придумувати й записувати команди, мільйони яких комп'ютер виконує за секунду. Цикл – це спосіб компактного запису великого обсягу запланованої до виконання роботи. Про інший спосіб компактного записування обчислень – рекурсію – йтиметься пізніше.

У мовах C та C++ передбачено три цикли: з лічильником, із передумовою (умовою продовження) та з постумовою (умовою закінчення).

Лістинг 2.60

```
// Приклад циклу з лічильником
double power(double x, int n) {
   double y=1;
   for(int i=0; i<n; i++)
      y*=x;
   return y;
}
```

Вправа 2.15

Подумайте, що міг би означати ось такий вироджений варіант порожнього циклу:

```
for(; ;);
```

Виконайте обчислювальний експеримент, спостерігаючи за ходом виконання цього циклу.

Відомо, що цикл із лічильником використовують для програмування обчислень за рекурентними співвідношеннями. Так, попередній приклад піднесення до степеня – це, власне, переписане рекурентне співвідношення

$$y_0 = 1;$$
$$y_n = xy_{n-1},\ n = 1, 2, \ldots .$$

Розглянемо ще одне відоме рекурентне співвідношення для чисел Фібоначчі:

$$F_0 = 0;\ F_1 = 1;$$
$$F_n = F_{n-1} + F_{n-2},\ n = 2, 3, \ldots .$$

Його можна запрограмувати так:

Лістинг 2.61

```
int Fibonacci(int n) {
   if(n==0) return 0;
```

```
   if(n==1) return 1;
   int f=0, a=0, b=1;
   for(int k=2; k<=n; k++) {
      f=a+b; a=b; b=f;
   }
   return f;
}
```

Вправа 2.16

Протестуйте функцію `Fibonacci`. Зверніть увагу на значення чисел Фібоначчі з великими номерами (наприклад, порівняйте F_{46} і F_{47}). Що станеться, якщо для чисел Фібоначчі замість цілого типу задати **`long int`**? А якщо **`long long int`**? У яких межах можна гарантувати точність одержаних результатів?

Лістинг 2.62

```
// Алгоритм Евкліда: цикл з умовою продовження
while(m!=n)
   if(m>n) m-=n;
   else n-=m;
```

А це інший відомий алгоритм – швидкого піднесення до степеня:

Лістинг 2.63

```
double power(double x, int n) {
   double y=1;
   while(n>0)
      if(n%2==0) {n/=2; x*=x;}
      else { n--; y*=x;}
}
```

Для прикладу *циклу з постумовою* розглянемо одну цікаву задачу, пов'язану з іменем німецького математика Карла Гаусса. Мова йде про обчислення так званого арифметико-геометричного середнього $M(a,b)$ двох дійсних чисел $0 < a < b$. Його визначають як спільну границю двох послідовностей, з яких одна монотонно зростає, а інша – монотонно спадає:

$$M(a,b) = \lim_{n\to\infty} a_n = \lim_{n\to\infty} b_n;$$

ці послідовності задано рекурентними співвідношеннями

$$a_0 = a,\ a_n = \sqrt{a_{n-1}b_{n-1}},\ b_0 = b,\ b_n = \frac{a_{n-1} + b_{n-1}}{2},\ n = 1, 2, \dots .$$

```
// Арифметико-геометричне середнє: цикл з постумовою        Лістинг 2.64
double Gauss(const double a, const double b) {
   double sa=a, sb=b; double aa=sqrt(a*b); double bb=(a+b)*0.5;
   do {
      sa=aa; sb=bb; aa=sqrt(sa*sb); bb=(sa+sb)*0.5;
   }
// Проміжок [a, b] не може містити нескінченної послідовності,
// яка зростає, тому умова продовження циклу обов'язково порушиться
   while((sa<aa)&&(aa<bb)&&(bb<sb));
   return aa;
}
```

Зверніть увагу на умову продовження циклу. Зазвичай для наближеного визначення границі задають точність обчислень. Як зазначав професор Фрідріх Бауер [23], під час обчислення арифметико-геометричного середнього немає потреби задавати точність. Монотонність кожної з послідовностей $a_{n-1} < a_n < b_n < b_{n-1}$ гарантує обчислення з точністю, максимально можливою на використовуваній апаратурі (поясніть, чому).

Вправа 2.17 (математичний етюд)

Спробуйте довести існування арифметико-геометричного середнього для будь-яких дійсних чисел a та b таких, що має місце нерівність $0 < a < b$.

На завершення згадаємо ще один часто використовуваний вид циклу – нескінченний. Він не потрапив до стандартних видів циклу в C/C++, однак легко моделюється за допомогою тотожно істинної умови й оператора виходу **break**:

```
int n; char ch;                                              Лістинг 2.65
while(true) {
   cout<<"Уведіть число n:"; cin>>n;
   cout<<"Fibonacci "<<'('<<n<<")="<<Fibonacci(n)<<endl;
   cout<<"Повторити(y/n)?"; cin>>ch;
   if(ch=='n')
      break;
}
```

Вправа 2.18

Перепишіть програму лістингу 2.65, застосувавши арифметичний цикл `for(; ;)`.

2.7. Налагодження програм

Щоб краще зрозуміти, у чому полягає налагодження програм і з якою метою воно виконується, згадайте, скільки помилок ви робили в диктантах рідною мовою, не кажучи вже про твори чи перекази. Іще більше помилок трапляється в разі користування іноземною мовою. Перевірити кілька творів не так важко, бо вони невеликі за обсягом. А тепер уявімо вчителя, який повинен перевірити твори всієї школи, усього району чи цілого міста. Він не лише не помітить певних помилок, а може навіть пропустити чийсь зошит.

Налагодження програм – це пошук і виправлення помилок. На їх наявність програми перевіряють на кількох рівнях. Компілятор аналізує синтаксичну правильність і видає повідомлення про помилки, позбутися яких порівняно легко. Складніше виявити й виправити змістовні семантичні помилки. Головна проблема полягає в необхідності перевірити зазвичай велику кількість різних ланцюжків інструкцій, що виникають у процесі виконання програми залежно від результатів перевірок умов у циклах і розгалуженнях. Виконуючи її десятки й сотні разів, ми, образно кажучи, проходимо по ній однією протоптаною стежкою, не помічаючи нетрів узбіч. Кажуть, що кожна виявлена в програмі помилка є передостанньою: досить зазирнути за поворот – і ви натрапите на наступну.

2.7.1. Макрос суджень

Перший засіб, який ми розглянемо, призначений для виявлення у програмах логічних помилок. Це так званий *макрос суджень* (assertion statement) `assert`. Ідея його використання полягає в перевірці у різних частинах програми особливих умов, які називають *інваріантами*. На думку Девіда Гріса [24], немає жодної можливості простежити за всіма змінами, що відбуваються в програмі. Усе, що може змінюватись, змінюється зі швидкістю мільйонів операцій за секунду. Краще зосередитися на незмінному, виявивши такі властивості програми, які не можуть порушуватися – інваріанти програми. Порушення інваріанта свідчить про виникнення помилки.

Звісно, можна було б перевіряти інваріанти за допомогою звичайних розгалужень. Однак тоді в кожному місці перевірки потрібно було б писати код реагування на виявлену помилку. Це стає зайвим у разі використання макросу суджень (про програму). Виявивши помилку, такий макрос припиняє виконання програми, створюючи відповідне повідомлення.

Визначення макросу суджень міститься в системному файлі `<cassert>`. Як і решту системних файлів, його підключають до програми за допомогою команди

```
#include <cassert>
```

Головна особливість макросу суджень полягає в тому, що його можна легко вмикати й вимикати однією простою командою препроцесора або режимами компіляції. Відтак для перевірки суджень програму запускають у *налагоджувальному режимі* (режим компіляції debug). Умиканням і вимиканням режиму керує константа препроцесора `NDEBUG` (No DEBUGging). Якщо її задати, то режим налагодження буде вимкнено. Для цього рядок

```
#define NDEBUG
```

слід помістити перед командою `#include <cassert>`. Вимкнути режим налагодження можна, обравши режим компіляції робочої версії програми (release).

Режим налагодження вмикають командою

```
#undef NDEBUG
```

Заголовний файл із визначенням `assert` містить такий фрагмент:

Лістинг 2.66

```
#ifdef NDEBUG
   #define assert(exp)((void)0)
#else
........................
```

Його читають так: якщо константу `NDEBUG` визначено, тобто налагодження вимкнено, то макрос `assert` перетворюється на порожній вираз. Схожий прийом можна застосовувати у програмах для того, щоб умикати чи вимикати деякі налагоджувальні команди, наприклад виведення.

Вправа 2.20

Проаналізуйте наведені в наступному прикладі судження та поясніть, як їх використано для забезпечення коректності програми.

Лістинг 2.67

```
int main(void) {
// Ділення націло
   int m,n;
   cout<<"give m="; cin>>m;
   cout<<"give n="; cin>>n;
   int q=0, r=m;
// Макрос перевіряє інваріант, що міститься в ньому (m==n*q+r)&&(r>=0):
// його порушення свідчитиме про помилку в ініціалізації
   assert((m==n*q+r)&&(r>=0));
```

```
    while(r>=n) {
        q++; r-=n;
// Іще раз перевіряємо інваріант: його порушення свідчить про
// некоректність тіла циклу
        assert((m==n*q+r)&&(r>=0));
    }
// До інваріанту додано умову закінчення циклу r<n. Тут він не може
// порушитися, оскільки у такому разі не відбудеться виходу із циклу
    assert((m==n*q+r)&&(r>=0)&&(r<n));
    cout<<"q="<<q<<" r="<<r<<endl;
    return 0;
}
```

Вправа 2.21

Як поводитиметься програма з попередньої вправи, якщо:

а) ініціалізацію замінити на

```
int q=1, r=m;
```

б) у тілі циклу записати

```
q--; r+=n;
```

Макрос суджень реагує на всі аварійні ситуації одним і тим самим способом – завершує виконання програми та виводить повідомлення про порушення судження. Він призначений для виявлення можливих помилок у програмному коді, а тому його слід вимкнути після налагодження, щоб уможливити використання програми в робочому режимі.

2.7.2. Інструкція випробувань і блок захоплення

Інший засіб обробки помилкових ситуацій складає пара – *інструкція випробувань* (try statement), можливо, разом з *інструкцією спричинення* `throw`, і *блок захоплення* (catch block). Їх, на відміну від макросу суджень, не відключають, а вбудовують у програму назавжди. Цей засіб призначено для виявлення некоректного використання програми, зокрема внаслідок уведення неприйнятних даних. Тому його можливості дещо ширші. Це, щонайменше, виведення власного діагностичного повідомлення, передбаченого для кожного типу аварійної ситуації. Можлива навіть нейтралізація помилки, але для її програмування потрібен певний досвід.

Зауважимо, що програмована обробка помилкових ситуацій скасовує їх стандартну обробку системою. Тому в програмі слід чітко окреслити межі програмованої обробки помилок. Їх визначають за допомогою опе-

ратора випробувань **try**. Ним відкривається *блок випробувань* (try block), безпосередньо за яким розміщується блок захоплення **catch**.

```
try {
   Усі аварійні ситуації, що виникнуть під час  виконання цього блоку
чи будуть створені в ньому за допомогою оператора throw, потраплять на
обробку в наступному блоці catch
}
catch(const reason1& parameter1) {
   Обробка помилки типу reason1 за значення параметра parameter1)
}
.....................................................................
catch(const reasonN& parameterN) {
   Обробка помилки типу reasonN за значення параметра parameterN)
}
```

Трапляються як стандартні, так і програмовані аварійні ситуації. Прикладом стандартної ситуації може бути згадана в підрозділі 2.4.4 ситуація `bad_alloc`, відповідальна за переповнення пам’яті. Система породжує її під час виконання оператора **new**. У відповідь на неї надсилається блок захоплення **catch**(bad_alloc&).

Лістинг 2.68

```
try {
  double* px;
  for(int i=0; ;++i)   // Повторюємо, доки вдається
    px=new double [0x7ffffff]; // Розміщуємо 134217727 дійсних значень
}
catch(const bad_alloc&) {
   cout<<"Вільної пам'яті немає"<<endl;
}
```

Програмовані аварійні ситуації можна створювати, зокрема, у прикладних програмах – скажімо, після надходження некоректних вхідних даних. Для цього до об’єкта, що відповідає за спричинену виняткову ситуацію, застосовують оператор спричинення. Тип захоплення залежить від типу об’єкта, який зумовив спричинення. Сам об’єкт може бути параметром захоплення.

Інструкція спричинення **throw** сигналізує про виникнення в певній частині програми виняткової ситуації, реакцію на котру винесено в особливий блок обробки помилкових ситуацій. Перевага застосування інструкцій спричинення полягає у відокремленні обробки помилкової ситуації від місця її виникнення.

Функція, у якій виникла така ситуація, не має ані знати про неї, ані перейматися тим, як її обробити. Завдання інструкції спричинення полягає в тому, щоб повідомити про відповідну подію, а блока випробувань – донести це повідомлення до обробника. Реакція на такий сигнал може бути стандартною, тобто аварійним завершенням програми, чи запрограмованою обробкою виняткової ситуації (якщо вона виникла в зоні спостереження спричинень, заданій за допомогою оператора **try**, до якого додано набір операторів захоплення **catch**).

Передбачені варіанти обробки записують у вигляді окремих блоків захоплення, які в разі виникнення ситуації буде переглянуто послідовно. Якщо жоден із них не підходить, то виконання програми аварійно припиняється.

Розглянемо приклад.

Лістинг 2.69

```
int n;
// Приклад блока випробувань: цикл повторюватиметься,
// доки користувач не задасть правильний параметр
while(true)
   try {
      cout<<"Уведіть значення n:"; cin>>n;
//    Від'ємне значення: обчислення неможливе,
//    викликаємо аварійну ситуацію з параметром цілого типу
      if(n<0)
         throw n;
//    Нормальне продовження обчислень
      …………
      break;
   }
// Обробка аварійної ситуації
   catch(int n) {
      cout<<"Ви задали від'ємне число "<<n<<endl;
   }
// Нормальний перебіг обчислень: сюди доберемося, коли користувач
// задасть правильний параметр
cout<<"Fibonacci="<<Fibonacci(n)<<"\n";
```

2.8. Масиви

Як уже було зазначено, мова C++ багатопарадигменна та багаторівнева. Багатопарадигменність дає змогу в одній програмі застосовувати як окремі функції (процедурна парадигма), так і цілі класи (об’єктна паради-

гма), пов'язані складними ієрархічними залежностями (об'єктно-орієнтована парадигма). Багаторівневість полягає в можливості одночасного використання, з одного боку, засобів машинного рівня (наприклад, побітових операцій), з іншого – абстрактних класів та інтерфейсів.

Чи не найповніше сусідство апаратного й абстрактного аспектів виявляється в масивах. У мові C++ можна водночас використовувати масиви різних рівнів. Передусім це успадковані від C масиви низького рівня, реалізовані апаратними засобами модифікації виконавчої адреси команди за допомогою індекс-регістрів. Саме такі масиви буде розглянуто в цьому підрозділі, а пізніше – масиви у вигляді програмованих класів, асоціативні масиви тощо.

Розглядаючи спадкоємність мови C++ стосовно C, зазначимо, що здатність C++ підтримувати все або майже все програмне забезпечення, раніше створене в C, стала чи не найважливішою причиною популярності цієї мови. Тому в одній C++-програмі архаїчні засоби можуть співіснувати з новітніми. Складаючи власні програми, можна добирати засоби, які вам до вподоби. Тому в підручнику ми розглядатимемо переважно засоби високого рівня абстракції. Утім програмістові доводиться також супроводжувати раніше створені програми, у яких може бути використано засоби досить низького рівня та які завдяки спадкоємності мови C++ залишаються життєздатними. Засоби низького, фактично апаратного, рівня можуть виявитися корисними й для ефективної реалізації нових абстрактних конструкцій. Масиви, вивчення яких ми розпочинаємо, – це приклад архаїчних конструкцій низького рівня.

2.8.1. Масиви та указники

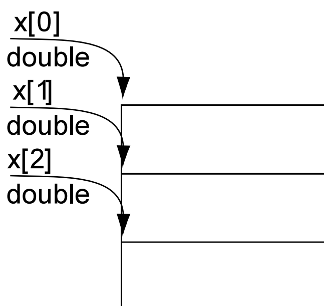


Рис. 2.16. Зображення масиву в пам'яті

Стандартний масив у мові C, а отже, й у C++, завжди одновимірний (хоча при цьому він може бути масивом масивів), його індекс завжди цілий, нижня межа індексу – нуль, а верхня – константа. Так, запис

```
double x[3];
```

означає три елементи пам'яті, розміщені послідовно та пронумеровані як `x[0]`, `x[1]`, `x[2]` (рис. 2.16).

Вихід індексу за межі масиву не контролюється: вирази `x[-1000]`, `x[3]`,

`x[4]`, `x[10000]`, тощо не будуть позначені компілятором як помилкові, хоча явно ні на що не вказують.

Вправа 2.22

Перевірте, якими значеннями буде заповнено масив безпосередньо після його створення.

Визначення та використання масивів

Як ви пересвідчились у вправі 2.22, щойно створений масив заповнений «сміттям», якщо його під час визначення не проініціалізувати – наприклад, так (рис. 2.17):

```
double x[3]={0.1, 0.2, 0.3};
```

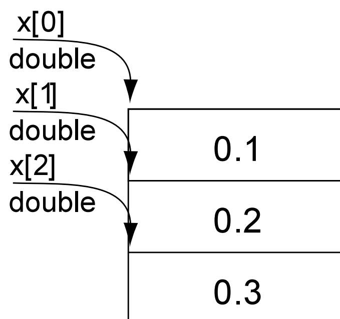


Рис. 2.17. Ініціалізація масиву

На жаль, допустима неповна ініціалізація: її список може бути коротшим, ніж розмір масиву (але не може бути довшим – це діагностуватиметься компілятором як помилка):

```
double y[100]={0.1, 0.2};
```

У цьому разі решта елементів масиву буде проініціалізовано стандартним значенням.

Вправа 2.23

Перевірте, як неповна ініціалізація масивів діє у випадку різних типів даних, зокрема при ініціалізації масиву символів.

Питання про збіг кількості елементів масиву з довжиною списку ініціалізації не виникає, якщо розмір масиву задано неявно за кількістю ініціалізаторів:

```
double z[]={0.1, 0.2, 0.3, 0.4, 0.5};
```

Отже, у масиві `z` п'ять елементів. На жаль, не завжди можна програмно підрахувати фактичну кількість елементів масиву. Це одна з найсерйозніших проблем, пов'язаних з обробкою масивів. У наведеному вище прикладі кількість елементів масиву можна обчислити як

```
sizeof(z)/sizeof(double);
```

Пізніше побачимо, що масив, переданий до функції як параметр, може «забути» свій розмір.

Розглянемо фрагмент програми обробки масивів. Спосіб доступу до елементів масиву очевидний – за індексом.

Лістинг 2.70

```
// Скалярний добуток
const int n=10; double x[n], y[n];
// Далі виконується ініціалізація масивів
.......................................
double s=0;
for(int i=0; i<n; i++)
   s+=x[i]*y[i];
```

Елементи масиву можуть мати будь-які типи; вони можуть бути навіть указниками, але тільки не псевдонімами (поясніть, чому), наприклад:

Лістинг 2.71

```
int i=1, j=1, k=1;
int* ip[]={&i, &j, &k};
int& ir[]={i, j, k};   // Error
```

Масиви призначені для прямого доступу до їх елементів за допомогою модифікації базової адреси (початку масиву) на величину, кратну індексу шуканого елемента (тому, зокрема, індекс першого елемента масиву – нуль). Тоді в разі використання масиву указників

```
int* ip[]={&i, &j, &k};
```

самі указники `ip[0]`, `ip[1]`, `ip[2]` буде розміщено в пам'яті один за одним, і вони відсилатимуть за адресами до числових значень, що можуть міститися де завгодно. Цей прийом стає абсолютно непридатним, якщо спробувати створити масив відсилок

```
int& ir[]={i, j, k};   // Error
```

тому що в цьому разі елементи `ip[0]`, `ip[1]`, `ip[2]` стали б псевдонімами відповідно імен `i`, `j` і `k`, хоча немає жодних підстав припускати, що значення цих змінних розміщено в пам'яті послідовно.

Елементами масивів можуть бути навіть масиви. У цьому разі одержуємо варіант двовимірних масивів:

Лістинг 2.72

```
// Множення матриць
const int n=10, m=12, l=14;
double a[n][m], b[m][l], c[n][l];
// Ініціалізація масивів
```

………………………………

```
for(int i=0; i<n; i++)
for(int j=0; j<l; i++) {
   c[i][j]=0;
   for(int k=0; k<m; k++)
      c[i][j]+=a[i][k]*b[k][j];
}
```

Увага, типова помилка! *Вираз* `a[i,j]`*, яким в деяких інших мовах програмування позначають елемент двовимірного масиву, у мові С++ теж цілком коректний, але його значення зовсім інше. Це пов'язано з операцією «кома». Квадратні дужки – це також операція, аргументом якої стає вираз, побудований за допомогою коми. Розставивши пропущені дужки, одержимо* `a[(i,j)]`*. Значення виразу в круглих дужках – його другий аргумент, тобто значення всього виразу* `a[i,j]` *– елемент масиву* `a[j]` *(перший елемент j-го рядка).*

У цілому, деякі вирази в мові С++ читати складно. Щоб уникнути помилок, варто зважати на пріоритети операцій, за допомогою яких ці вирази побудовано. Наприклад, вираз `int* iar[]` читають як `int* (iar[])` – масив указників на цілі. В указнику на масив цілих потрібно розставити дужки інакше – `int(*px)[]`.

Адресна арифметика

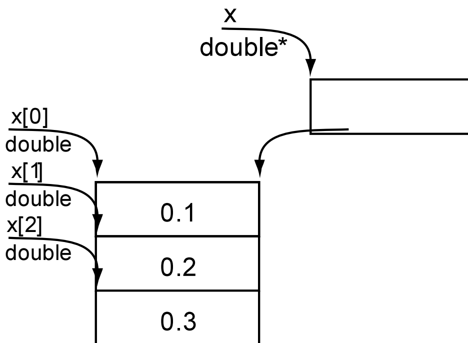


Рис. 2.18. Масиви як указники

Розглянемо зв'язок між масивами та указниками. Певною мірою можна вважати, що в мові С масивів немає – є лише особлива система позначень для використання указників як масивів.

Повернімося до прикладу визначення масиву

```
double x[3]={0.1, 0.2, 0.3};
```

На рис. 2.18 показано, як система програмування інтерпретує цей оператор. Вона створює сталий указник `x`, який з умови `x[0] == *x` визначає перший елемент масиву (з нульовим індексом).

Наступні елементи масиву буде обчислено методами адресної арифметики:

```
x[1] == *(x+1)
x[2] == *(x+2)
```

Лістинг 2.73

і т. д. Тепер стає зрозумілішим принцип адресної арифметики: збільшення указника на одиницю дає адресу, де міститься наступне значення, тип якого задано типом указника. Так, скажімо, збільшення на одиницю указника на дійсні числа подвійної точності збільшує значення адреси на 8, тоді як така сама дія для указника на символи – на 1.

Нехай десь у пам’яті розміщено три указники:

```
double* xd; float* xf;
char* xc;
```

Лістинг 2.74

На рис. 2.19 показано адреси, одержані застосуванням адресної арифметики до указників залежно від їхнього типу за припущення, що всі три указники спрямовані на одне й те саме місце в пам’яті.

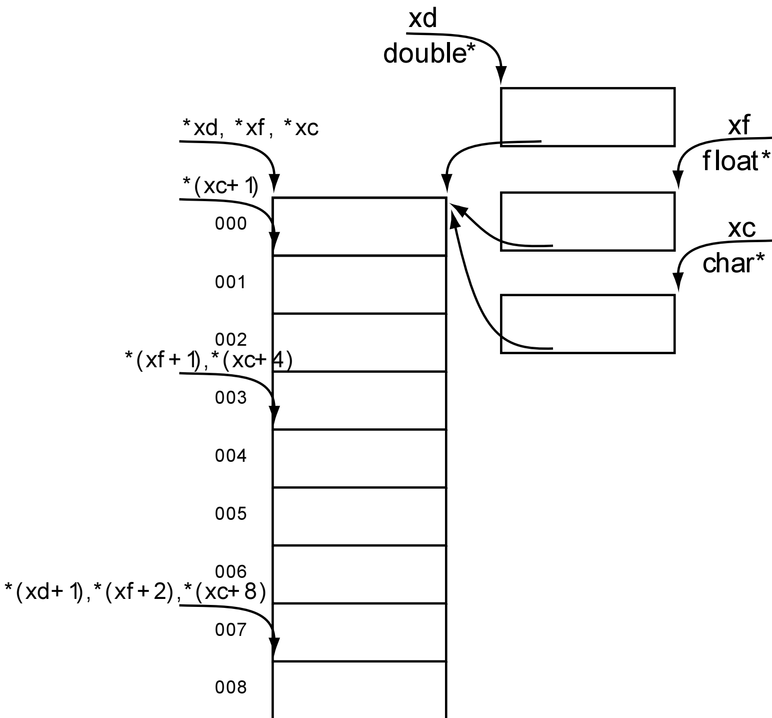


Рис. 2.19. Адресна арифметика

Зазначимо, що дужки в наведених виразах вигляду `*(x+1)` не зайві, оскільки вираз із дужками має зовсім інше значення, ніж вираз без дужок `*x+1`. Справді,

Лістинг 2.75

```
*(x+1)==rvalue(x+1);
*x+1==rvalue(x)+1;
```

Замість визначення масиву

```
double x[3]={0.1, 0.2, 0.3};
```

можна визначити й проініціалізувати указник. Відповідність буде повнішою, якщо використати сталий указник

```
double* const px=new double [2];
```

який у цьому разі надійніший, тому що його не можна переспрямувати на інше місце, залишивши масив напризволяще.

Безпосереднє застосування указників замість масивів має низку переваг, які ми розглянемо, але має й одну хибу: визначення указника не можна сумістити з ініціалізацією.

Динамічні масиви й ітератори

І нарешті, указники дають можливість створювати прості динамічні масиви, розміри яких буде задано не під час компіляції, а на етапі виконання програми. Називатимемо їх простими, оскільки після створення їхні розміри не змінюватимуться. У майбутньому ми ознайомимося з динамічними масивами в повному значенні цього слова.

Попередній масив можна було б задати так:

Лістинг 2.76

```
int n=100;
cout<<"Задайте розмір масиву"; cin>>n;
double* const px=new double [n];
```

Тепер немає потреби перекомпільовувати програму щоразу після зміни значення розмірності.

Реалізація масивів за допомогою указників дає ще один метод вибору елементів масиву, альтернативний звичайному використанню індексів. Це застосування так званих *ітераторів* (iterator). Сутність зазначеного методу полягає у фіксації за допомогою двох указників відповідного типу першого й останнього елементів масиву, точніше місця в пам’яті, наступного за кінцем масиву.

Застосовування ітераторів ілюструє такий фрагмент:

```
const int n=1000;                                                Лістинг 2.77
int ia[n];    // Визначаємо масив
int* pia=ia;  // Поміщаємо указник поточного елемента на початок масиву
int* pend=ia+n;    // Визначаємо указник виходу за межі масиву
while(pia!=pend)   // Виконуємо цикл від початку масиву до його кінця
   cout<<*(pia++)<<endl;
```

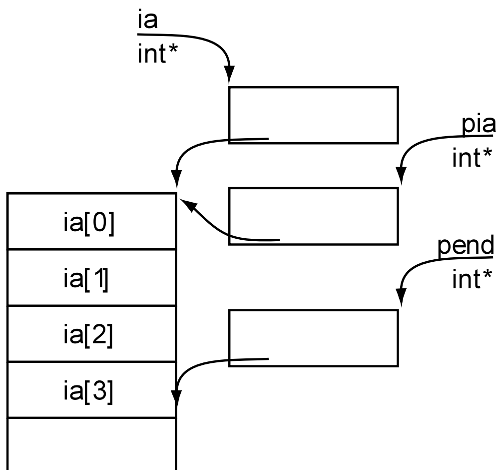


Рис. 2.20. Масив, наділений ітераторами

Тут використано два указники (рис. 2.20). Один, власне ітератор, потрібно встановити на початок масиву та посуватися по ньому, застосовуючи адресну арифметику. Інший указник обмежує вихід за межі масиву. Зараз іще не видно переваг ітераторів над традиційними методами організації циклів із погляду сучасних технологій програмування, зокрема для досягнення мобільності та гнучкості програмного забезпечення. Однак подумайте про критерії їх порівняння в контексті ефективності використання апаратури.

Наведемо схожий приклад, але із застосуванням динамічного масиву.

```
// Запит на розмір масиву                                        Лістинг 2.78
int n;
cout<<"Задайте розмір масиву: "; cin>>n;
int* const dynar=new int [n]; // Виділення області пам'яті
for(int i=0; i<n; i++)        // Традиційний цикл
   dynar[i]=i;
// Визначення ітераторів
int* pcur=dynar; int* pend=dynar+n;
while(pcur!=pend)             // Цикл за ітератором
   cout<<*(pcur++);
delete [] dynar;              // Звільнення пам'яті
dynar=0;
```

Створенням і знищенням динамічних масивів тут керує сама програма. Кожній команді `new` відповідає своя команда `delete`.

Увага, типова помилка! *Унаслідок виконання операції* `new int [n]` *буде створено* `n` *елементів пам'яті, кожний із яких призначено для зберігання одного значення цілого типу. Елементи пам'яті не ініціалізовано, вони містять «сміття», і немає способу сумістити визначення з ініціалізацією. Операція* `new int (n)` *створює один елемент пам'яті, ініціалізований значенням* `n`*. Операція* `delete dynar` *видаляє один елемент пам'яті – той, на який показує указник* `dynar`*. Якщо пам'ять було виділено за допомогою операції* `int* dynar = new int [n]`*, то* `n-1` *елементів пам'яті залишаться «сміттям». Операція* `delete [] dynar` *видаляє* `n` *елементів пам'яті.*

Правило гарного тону. *Якщо пам'ять виділено командою* `new []`*, то видаляти її потрібно за допомогою команди* `delete[]`*. Детальніше проблеми створення та видалення динамічних об'єктів буде розглянуто в підрозділі 7.9.*

Зробимо зауваження стосовно нестачі пам'яті на етапі виконання програми. Тривіальна реакція на таку ситуацію – припинити виконання програми та вивести повідомлення про місце, у якому зазначена проблема виникла. Однак цього може виявитися замало. Скажімо, корисно було б зберегти вже одержані результати й, можливо, виконати ще якісь захисні дії. Проте пам'яті може не вистачити навіть для цих дій. Можна порекомендувати попередньо зарезервувати своєрідний запас пам'яті під буфер, призначений для обслуговування аварійної ситуації.

2.8.2. Масиви символів і указники

Обробка текстової інформації являє собою одне з найважливіших і, на перший погляд, найпростіше застосування обчислювальної техніки. Кожен символ – це байт, послідовність символів – масив байтів, пов'язаний з указником відповідного.

Найнижчий рівень опрацювання рядків символів полягає у використанні символьних указників. Цю техніку успадковано від мови С. Із попереднього підрозділу відомо, що масиви «не знають» своїх меж, проте масиви символів – виняток: *домовлено завжди закінчувати їх нульовим кодом* `'\0'`.

Символьний рядок можна уявляти собі як масив

```
char hello[11];
```

який, згідно з правилами гарного тону, бажано вже під час визначення проініціалізувати послідовністю символів:

```
char hello[11]=                                        Лістинг 2.79
   {'Д', 'о', 'б', 'р', 'и', 'д', 'е', 'н', 'ь', '!', '\0'};
```

Як і раніше, в означенні можна явно не зазначати розміру масиву (таким чином, нагадаємо, усувають конфлікти невідповідності між заданим розміром і довжиною послідовності ініціалізації):

```
char hello[]=                                          Лістинг 2.80
   {'Д', 'о', 'б', 'р', 'и', 'д', 'е', 'н', 'ь', '!', '\0'};
```

Масив символів – єдиний тип масиву, довжину якого можна обчислити програмно за нульовим кодом. Тому, ініціалізуючи рядок описаним вище способом, не слід забувати про нульовий код. Якщо ж він потрапить усередину масиву (скажімо, помилково), то ним рядок і закінчиться:

```
char hello[11]=                                        Лістинг 2.81
   {'Д', 'о', 'б', 'р', '\0', 'д', 'е', 'н', 'ь', '!', '\0'};
cout<<hello<<endl;   // Надрукує «Добр»
```

Тому рядки символів краще ініціалізувати літералом (хоча така ініціалізація в певному розумінні порушує типізацію):

```
char hello[]="Добридень!";
```

Тут масив `hello` знову складається не з 10, а з 11 символів (нульового, першого, другого, …, десятого), але доповнення літералу нульовим кодом бере на себе компілятор. Подивитися на код останнього символу рядка можна, перетворивши його на ціле число, наприклад:

```
cout<<static_cast<int>(hello[10])<<endl;
```

У програмах обробки текстової інформації не без підстав використовується припущення про завершення рядків нульовим кодом. Слід пам'ятати, що довжина рядка завжди на одиницю менша за довжину масиву. Скажімо, масив

```
char lambda[]="";
```

складається з одного елемента, тоді як відповідний йому рядок порожній.

Іще частіше рядок означають не як масив, а як указник:

```
char* hello="Добридень!";
```

Проте варто пам'ятати, що, на відміну від масиву, указник легко спрямувати на інше місце так:

```
hello=new char [5];
```

або так:

```
hello="Something else";
```

зробивши попередню ділянку пам'яті «сміттям», яке не можна видалити (за правилом парності команд **new** та **delete**: не було **new** – немає й **delete**).

Зокрема, через створювані указниками проблеми з обробкою рядків стали використовувати класи рядків символів, про що йтиметься пізніше.

Завершення рядків нульовим кодом розв'язує проблему довжини символьних масивів. Довжину рядка (якщо він правильно закінчується нульовим кодом) можна підрахувати так:

Лістинг 2.82

```
int n;
char* counter=hello;
for(n=0; *counter++; n++)
   ;
cout<<"Length of: "<<hello<<'='<<n<<endl;
```

Наведений цикл записано типовим компактним стилем програмування мови C, що допускає змішування арифметичного циклу та циклу за умовою. Символьний вираз `*counter++` – приклад типової ідіоми мови C, яку тут було вжито як умову продовження циклу. Спочатку беремо символ, на який вказує указник, тобто обчислюємо значення виразу `*counter`, а потім збільшуємо указник на одиницю, переходячи у такий спосіб до наступного символу. Пам'ятаємо, що все відмінне від нуля вважається істинним. (Спробуйте записати щось подібне в Паскалі!) Ось цей самий цикл у звичайному записі для масивів:

Лістинг 2.83

```
for(n=0; hello[n]; n++)
   ;
```

І в першому, і в другому випадках тіло циклу складає порожня інструкція, бо всі дії зосереджено в заголовку. Перепишемо зазначений цикл у традиційній формі циклу за умовою:

Лістинг 2.84

```
int n=0;
while(hello[n++]!=0)
   ;
cout<<"Length: "<<hello<<'='<<n<<endl;
```

або зовсім без застосування стандартної ідіоми порожнього тіла циклу:

```
int n=0;
while(hello[n]!=0) n++;
cout<<"length: "<<hello<<'='<<n<<endl;
```

Лістинг 2.85

Вправа 2.25

Виконайте три наведені цикли на комп'ютері та поясніть розбіжності в одержаних значеннях.

Зауваження. *Зверніть увагу на особливість виведення символьного указника*

```
cout<<hello<<endl;
```

Чим воно відрізняється від виведення числового указника, наприклад `int* p`*?*

Використання указників для іменування рядків породжує ще одну проблему – нульового указника. Домовилися указник, який не має значення, означати так:

```
char* foo=0; // Невизначений указник
```

Слід пам'ятати, що указник, який може виявитися неозначеним, не варто розіменовувати без попередньої перевірки на нуль, бо це може спричинити помилки під час виконання програми. З рядками ж такої проблеми не виникає. Указник, якому ще не надано значення, ініціалізують порожнім рядком:

```
char* foo="";// Порожній рядок
```

що повністю розв'язує проблему невизначеності символьних указників і робить їх безпечними для розіменування без попередньої перевірки.

2.9. Програмовані структури даних

До програмованих структур даних належать, крім власне структур, також бітові поля і об'єднання.

2.9.1. Структури

Структури визначають достатньо зрозумілим способом. Як приклад наведемо визначення структури `Point` – «точки площини»:

```
struct Point {
   double _x; double _y;
};
```

Лістинг 2.86

Точка площини характеризується абсцисою `_x` і ординатою `_y`, але не варто намагатись обчислити місце ординати в пам'яті за місцем абсциси,

користуючись адресною арифметикою, як у разі масивів. Структура Point і масив із двох дійсних чисел – несумірні типи даних.

Звернімо увагу на позначення, яких ми дотримуватимемось у підручнику. Імена членів (полів) структури починаються символом підкреслення «_». Для них немає загальновживаних позначень. Навколо цього питання точаться постійні суперечки, але використання особливого позначення для полів структури виправдане тим, що ці імена не мають самостійного значення; поза структурою вони позбавлені змісту.

Розміщення структур у пам'яті

Наведемо ще один приклад структури:

Лістинг 2.87

```
struct Collection {
   char _chr; double _d; short int _si; double _f;
};
```

Подивимося на адреси її полів у пам'яті. Візьмемо екземпляр цієї структури

```
Collection smthng;
```

і спробуємо вивести адреси його самого та його компонент:

Лістинг 2.88

```
cout<<&smthng<<endl;
cout<<&smthng._chr<<&smthng._d<<&smthng._si<<&smthng._f<<endl;
```

Припустимо, що об'єкт smthng міститься за адресою 1245032. Цю саму адресу має поле _chr, але наступне поле _d внаслідок застосовуваного вирівнювання за подвійним словом потрапить до комірки з адресою 1245040, а не 1245033. Полю _si дістанеться адреса 1245048, а полю _f – адреса 1245052.

Порівняємо розмір структури з розміром її складових:

Лістинг 2.89

```
cout<<sizeof(Collection)<<endl;
cout<<sizeof(char)+sizeof(double)
    +sizeof(short)+sizeof(double)<<endl;
```

Виявляється, що внаслідок вирівнювання в пам'яті з'явилися вільні місця.

Визначення об'єктів-структур

Маючи наведене вище визначення типу Point, можна задати об'єкти цього типу. Визначення структурованих об'єктів можна оформити у два способи, а саме у стилі С – **struct** Point u; – і в стилі С++ – Point u.

Внаслідок використання в мові C++ структур, успадкованих від C, виникають певні непорозуміння. Зокрема, в попередніх двох визначеннях відсутня ініціалізація точки, яку було визначено. Те, що точка не мала ініціалізації, підтвердить також спроба вивести значення її координат

```
cout<<u._x<<','<<u._y<<endl;
```

Для ініціалізації структурованих об'єктів можна користуватися звичайними необов'язковими списками – такими самими, як для масивів:

Лістинг 2.90

```
Point u={1, 1};           // Ініціалізована точка
const Point zero={0, 0}; // Визначення сталої точки
```

Проте вилучення з визначення сталої

```
const Point zero;
```

її ініціалізації не призводить до помилки компіляції, хоча деякі системи можуть генерувати помилку етапу виконання. З'ясуємо, чим визначення неініціалізованої структурованої сталої відрізняється від спроби означити неініціалізовану дійсну сталу, наприклад таку:

```
const double pi;   // ERROR
```

Чому компілятор вимагає ініціалізації `pi`

```
const double pi=3.141592653589793;
```

але байдужий до того, що не ініціалізовано константу структурованого типу?

Річ у тім, що визначення структурованих об'єктів відбувається за допомогою спеціальної функції, яка називається конструктором. Правильно складені конструктори самі відповідають за ініціалізацію, тому компілятор за нею більше не стежить. Детальніше про конструктори йтиметься в підрозділі 4.2.

Point:	u	
double:	_x	1.0
double:	_y	1.0

Рис. 2.21. Діаграма екземпляру структури

Point	
double:	_x
double:	_y

Рис. 2.22. Діаграма структури

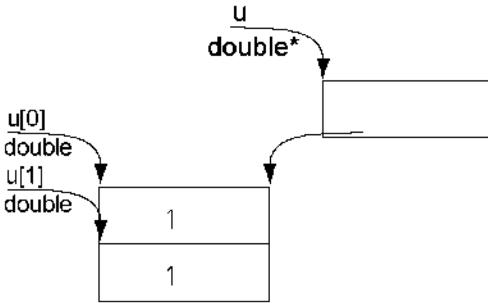


Рис. 2.23. Двоелементний масив

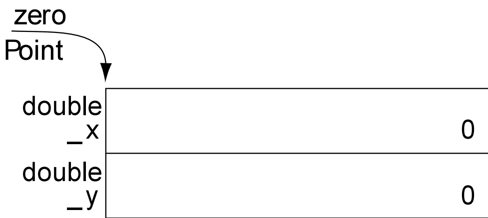


Рис. 2.24. Двоелементна структура

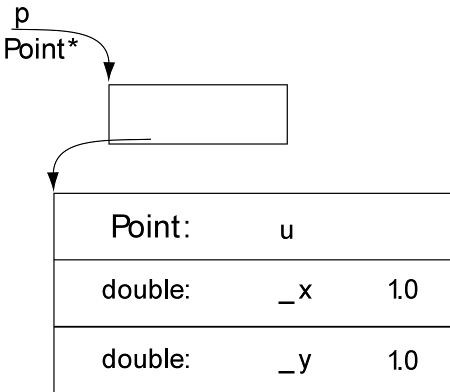


Рис. 2.25. Указник структури

Домовимося про графічні позначення. Зазвичай структуровані об'єкти зображають графічно у вигляді так званої діаграми об'єкта (рис. 2.21), а самі структури – у вигляді діаграмами структур (рис. 2.22).

Ці діаграми відрізняються тільки тим, що діаграма об'єкта, на відміну від діаграми структури, містить значення його полів.

Структури та указники

Структури навіть з однотипними полями не можна плутати з масивами, оскільки завжди є можливість працювати з масивами за допомогою указників (рис. 2.23):

```
double u[2]={1, 1};
```

У реалізації структур непряме адресування не застосовується, а тому указники не потрібні. Якщо взяти згаданий у лістингу 2.113 об'єкт `zero`, то кожен із його членів адресується прямо кваліфікованим іменем `zero._x` або `zero._x` без індексу (а отже, і без указника). У графічних позначеннях попередніх розділів об'єкт `zero` було б зображено приблизно так, як на рис. 2.24.

Звісно, можна означити також указник на структуру, але це вже зовсім інший тип даних (рис. 2.25):

Лістинг 2.91
```
Point u={1, 1};
Point* p=&u;
```

Масиви та структури різняться способом доступу до своїх компонентів. Для доступу до елементів масиву застосовують адресну арифметику; можна обчислювати індекси елементів масиву й організовувати за ними цикли. Поля ж структури одержують за допомогою особливого *оператора доступу до члена структури* (*крапка-оператор*):

Лістинг 2.92
```
Point u={1, 1};
u._x; u._y;             // Члени структури u
double v[2]={1, 1};
v[0]==*v; v[1]==*(v+1); // Елементи масиву
```

Непрямий доступ до членів структури забезпечується за допомогою указників. Найчастіше це указник на структуру. Застосування операції доступу до структури за указником (ця операція позначається як своєрідна стрілка `->`, складена із символів '`-`' і '`>`') і попереднє розіменування указника з наступним застосуванням оператора крапки наведені на лістингу 2.93:

Лістинг 2.93
```
Point* p=&u;
cout<<(p->_x==(*p)._x)<<endl;
// Значення за різними методами доступу збігаються
cout<<(&p->_x==&(*p)._x)<<endl;   // Адреси збігаються теж
```

та безпосередньо за допомогою указників на окремі члени:

Лістинг 2.94
```
double* px=&u._x;
double* py=&u._y;
```

Поверхневе присвоєння

Для однотипних структур визначено операцію поелементного, тобто так званого *поверхневого присвоєння* (simple assignment operator):

Лістинг 2.95
```
Point u={1, 1}; Point v={2, 3};
u=v; cout<<u._x<<' '<<u._y<<endl;                 // Буде виведено 2 3
// Наступна зміна v не впливає на u
v._x=10; v._y=20; cout<<u._x<<' '<<u._y<<endl;   // Буде виведено 2 3
```

Графічно цю операцію можна подати так, як показано на рис. 2.26.

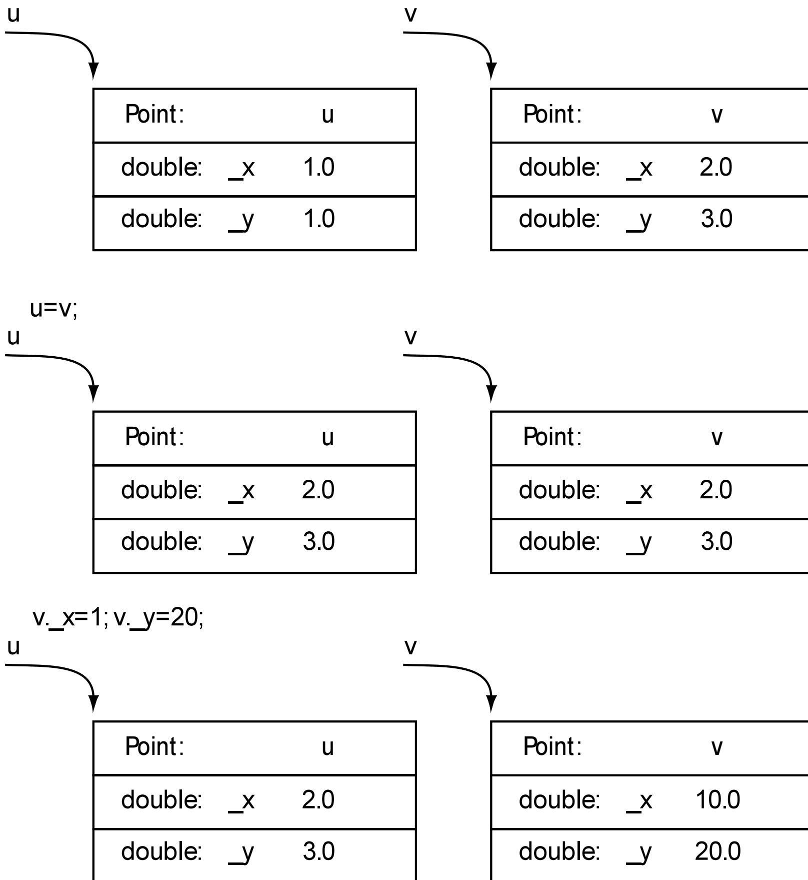


Рис. 2.26. Поверхневе присвоєння

Важлива відмінність між структурами та масивами полягає в тому, що присвоєння масивів, на відміну від присвоєння структур, неможливе:

Лістинг 2.96

```
double x[2]={1, 2}; double y[2]={10, 20};
x=y;      // ERROR!
```

Спробуйте самостійно відповісти на питання, чому це присвоєння масивів не буде відкомпільовано і призведе до такої помилки:

```
error C2106: '=' : left operand must be l-value
```

Однак присвоєння указників можливе:

Лістинг 2.97

```
double* x=new double[2];
x[0]=1; x[1]=2;
double* y=new double[2];
y[0]=10; y[1]=20;
x=y; cout<<x[0]<<' '<<x[1]<<endl; // Буде виведено 10 20
```

хоча й дає дещо несподіваний ефект (поясніть його):

Лістинг 2.98

```
// Чому зміна значення y[1] вплинула на x[1]?
y[1]=30; cout<<x[0]<<' '<<x[1]<<endl; // Буде виведено 10 30
```

Наведемо ілюстрації для пояснення ходу виконання команд з лістингів 2.97 і 2.98.

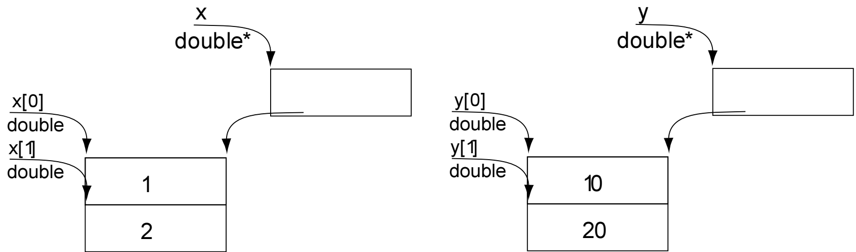


Рис. 2.27. Початкові значення масивів, заданих указниками

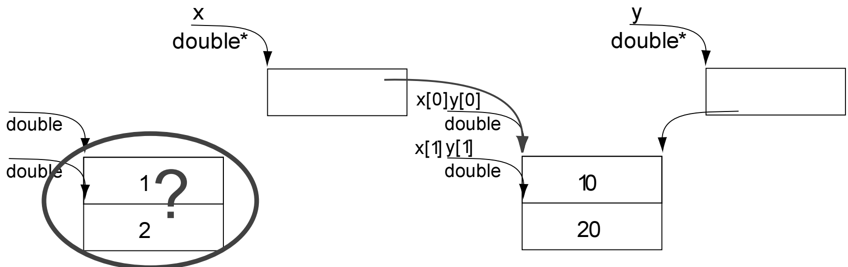


Рис. 2.28. Результат присвоєння x=y;

Ситуацію, що склалася, називають *утратою пам'яті* (утечка памяти, memory leak).

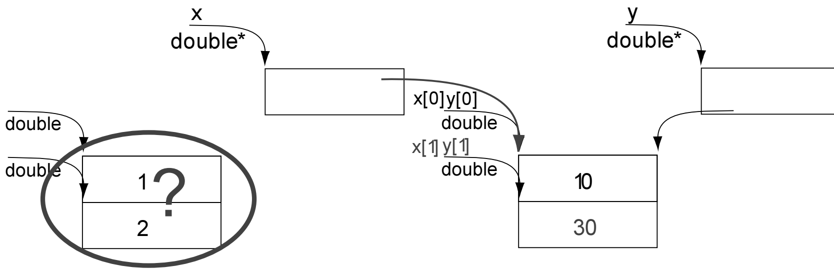


Рис. 2.29. Результат присвоєння y[1]=30;

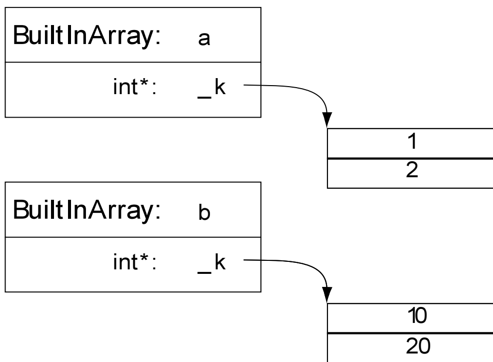


Рис. 2.30. Вбудовані масиви

Як бачимо, поверхневе присвоєння масивів, означених за допомогою указників, призводить до катастрофічних наслідків. Справді, присвоєння

```
y[1]=30;
```

дає інший, ніж у разі структур, результат (рис. 2.30):

```
cout<<x[0]<<' '<<x[1]
    <<endl; // 10 30
```

Утім справжня катастрофа станеться в разі спроби один за одним видалити масиви (поясніть, чому):

Лістинг 2.99

```
delete [] x;
delete [] y;
```

Така увага до присвоєння не випадкова. Оператор присвоєння – основа процедурного програмування, оскільки саме він виконує роботу над пам'яттю, з якою, як далі побачимо, будуть пов'язані досить тонкі проблеми.

Перша з них пов'язана з тим, що поля структури самі можуть бути указниками. Розглянемо приклад такої структури:

Лістинг 2.100

```
struct BuiltInArray {
   int* _k;
};
```

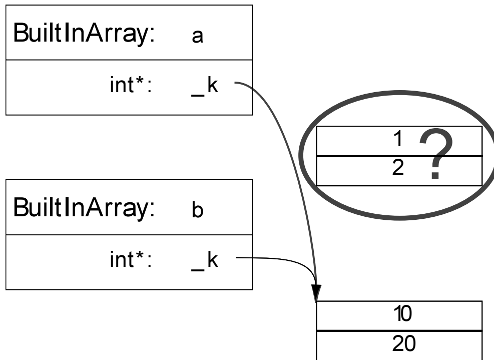


Рис. 2.31. Поверхневе присвоєння вбудованих масивів

Тепер створимо два екземпляри структури `BuiltInArray`. Пам'ятаймо, що з визначення указника неможливо зрозуміти, чи це масив, а якщо так, то зі скількох елементів він складається (рис. 2.31):

Лістинг 2.101

```
int main() {
   BuiltInArray a, b;
   a._k=new int[2];
   a._k[0]=1; a._k[1]=2;
   b._k=new int[2];
   b._k[0]=10; b._k[1]=20;
}
```

А ось що станеться внаслідок присвоєння `a=b` (див. рис. 2.31):

Лістинг 2.102

```
a=b;
cout<<a._k[1]<<endl;   // Буде виведено 20
```

Зрозуміло, що два масиви знову перетворилися на один. І знову-таки сталася втрата пам'яті.

Лістинг 2.103

```
a=b;
cout<<a._k[1]<<endl;   // Буде виведено 20
b._k[1]=100;
cout<<a._k[1]<<endl;   // Буде виведено 100
```

Вправа 2.27

А що станеться внаслідок присвоєння для наведених далі структур?

Лістинг 2.104

```
BuiltInArray a, b;
int na;
cout<<"Give na"; cin>>na;
// Створення na-ементного масиву в структурі a
a._k=new int[na]; a._k[0]=1; a._k[1]=2;
int nb; cout<<"Give nb"; cin>>nb;
// Створення nb-ементного масиву в структурі
b._k=new int[nb]; b._k[0]=10; b._k[1]=20; b._k[2]=30;
a=b;
```

Неважко здогадатися, що просто виправити некоректне присвоєння не вдасться. Кількість елементів у кожному з масивів наперед невідома й, узагалі кажучи, різна. Тому перемістити один масив на місце іншого неможливо. Потрібно застосовувати особливі варіанти програмованого присвоєння, про яке йтиметься далі.

2.9.2. Бітові поля

На структури дуже схожі *бітові поля* (bit field). Це також спадок мови C, який дозволяє програмувати низькорівневі операції з окремими розрядами машинних слів. Члени звичайної структури розміщено в пам'яті побайтово з урахуванням певних норм вирівнювання для кожного типу, а бітові поля – біт за бітом, незважаючи на байтову структуру даних. Під окреме поле можна виділити будь-який біт або групу бітів. Тип кожного з полів – **unsigned char**, а кількість бітів, виділених під поле, задають числом. Наведемо приклад структури, яка ділить байт на 8 бітів, даючи кожному з них особливу назву: `_clear_to_send`, `_data_ready` тощо:

Лістинг 2.105

```
struct StatusByte {
   unsigned char _clear_to_send: 1; unsigned char _data_ready: 1;
   unsigned char _record_end: 1; unsigned char _received_line: 1;
   unsigned char _trans_allowed: 1; unsigned char _line_ready: 1;
   unsigned char _ring_detected: 1; unsigned char _signal_accepted: 1;
};
```

Тип бітового поля може бути довільним. Головне – його ширина, тобто кількість бітів, відведених під це поле; її записують після двокрапки. Доцільно ввести зручне позначення, щоб визначення бітових полів було ще прозорішим:

```
typedef unsigned char Bit;
```

У такому разі структуру можна переписати так:

Лістинг 2.106

```
struct StatusByte {
   Bit _clear_to_send: 1; Bit _data_ready: 1;
   Bit _record_end: 1; Bit _received_line: 1;
   Bit _trans_allowed: 1; Bit _line_ready: 1;
   Bit _ring_detected: 1; Bit _signal_accepted: 1;
};
```

Виділяти біти по одному необов'язково. Розглянемо приклад байта, розділеного навпіл:

```
struct HalfByteColor {
   Bit _fistHalf:4; Bit _secondHalf:4;
}
```

Лістинг 2.107

2.9.3. Об’єднання

Об’єднання (union) відрізняються від структур тим, що в них на всі поля виділено одну й ту саму область пам’яті. Наприклад, об’єднання дає можливість залежно від обставин розглядати одне й те саме слово як ціле без знака чи як дійсне значення:

```
union Word {
   unsigned int _k;
   double _x;
};
```

Лістинг 2.108

Вправа 2.28

Спробуйте пояснити такий фрагмент коду:

```
Word w;
w._x=1;   // Задамо дійсне число 1
cout<<dec<<w._x<<':'<<hex<<w._k<<endl;  // Буде виведено 1:3F800000
w._k=1;   // Задамо ціле число 1
cout<<dec<<w._x<<':'<<hex<<w._k<<endl;  // Буде виведено 1.4013e-045:1
```

Лістинг 2.109

За допомогою об’єднання, суміщеного з бітовими полями, можна побачити байтову структуру слова, не вдаючись, як раніше, до логічних операцій:

```
union Word {
   unsigned int _k;
   double _x;
   struct Word4 {
      Bit _byte1: 8; Bit _byte2: 8; Bit _byte3: 8; Bit _byte4: 8;
   } _word;
};
Word myWord;
myWord._k=0xa1b1c1d1;
cout<<hex<<myWord._word._byte1<<':'<<myWord._word._byte2<<':'
    <<myWord._word._byte3<<':'<<myWord._word._byte4<<endl;
// Буде виведено d1:c1:b1:a1
```

Лістинг 2.110

Завдання до розділу 2

1. В області дії визначення **`int`** `x` спростіть вирази:
   а) `(x=y)?x=y:y=x;`
   б) `(++x=y, ++y=x);`
   в) `~x+1;`.
2. Спростіть оператори:
   а) **`if`** `(x==0) y=false;` **`else`** `y=true;`
   б) **`if`** `(x>y) z=x;` **`else`** `z=y;`
   в) `w=(z>((x>y)?x:y)?z:((x>y)?x:y));`
   г) **`int`** `x=0;` **`while`**`(x>0)++x;`
   д) **`int`** `x=1;` **`while`**`(++x>0);`.
3. Перепишіть цикл **`for`**`(i=1; i<n; ++i)` **`if`**`(a[i]==x)` **`break`**`;` не вживаючи оператора **`break`**.
4. Перепишіть цикл **`while`**`(F) {P;` **`if`**`(G)` **`break`**`; Q}` без використання оператора **`break`**.
5. Виразіть цикл з постумовою **`do`** `P` **`while`**`(F);` де `F` – умова, `P` – оператор, через цикл з передумовою.
6. Виразіть розгалуження **`if`**`(F) P;` **`else`** `Q;` де `F` – умова, `P` – оператор, через цикл.
7. Знайдіть значення заданого а) біта; б) байта в цілому числі без знаку.
8. Не застосовуючи операцію ділення, знайдіть частку і остачу від ділення цілого числа без знаку на а) 2; б) 32; в) 256.
9. Запрограмуйте обчислення $n!!$, де
   $$n!! = \begin{cases} 1\times 3\times \ldots \times n, n = 2k+1; \\ 2\times 4\times \ldots n, n = 2k. \end{cases}$$
10. Запрограмуйте обчислення скінченної суми $S_n = \sum_{k=0}^{n} \frac{(-1)^k x^k}{k!}, n \geq 0$, не використовуючи функцій обчислення факторіалу та піднесення до степеня, як власних, так і стандартних.
11. Запрограмуйте обчислення функцій $y = \sin x$ та $y = \cos x$ $(-\infty < x < +\infty)$ із заданою точністю $\varepsilon > 0$, спираючись на формули $\sin x = \lim_{n\to\infty} S_n$, де $S_n = \sum_{k=0}^{n} \frac{(-1)^k x^{2k+1}}{(2k+1)!}$, $\cos x = \lim_{n\to\infty} C_n$, де $C_n = \sum_{k=0}^{n} \frac{(-1)^k x^{2k}}{(2k)!}$. Встановіть межі застосовності цих формул.

12. Обчисліть золотий перетин *c*, використовуючи границю $c = \lim_{n \to \infty} \frac{F_n}{F_{n-1}}$, де F_n – *n*-е число Фібоначчі.
13. Запрограмуйте обчислення суми
$S_n = x^{1^2} + x^{2^2} + ... + x^{n^2}$. (Вказівка: $x^{i^2} = x^{(i-1)^2} \frac{(x^i)^2}{x}$).
14. Знайдіть рекурентні співвідношення та запрограмуйте обчислення многочлена $y = x^{1^3} + x^{2^3} + ... + x^{n^3}$ за довільними значеннями *x* і *n*.
15. Запрограмуйте обчислення многочлена
$y_n = nx^{n-1} + (n-1)x^{n-2} + ... + 2x + 1$.
16. Запрограмуйте обчислення числа π за формулою Вієта
$$\frac{2}{\pi} = \frac{\sqrt{2}}{2} \cdot \frac{\sqrt{2+\sqrt{2}}}{2} \cdot \frac{\sqrt{2+\sqrt{2+\sqrt{2}}}}{2} \cdot ... .$$
17. Запрограмуйте обчислення многочлена $y = x^{2^n} + x^{2^{n-1}} + ... + x^4 + x^2 + 1$.
18. Запрограмуйте обчислення за довільним непарним *n* ланцюгового дробу $1 + \cfrac{1}{3 + \cfrac{1}{5 + \cfrac{1}{\cfrac{...}{n + \cfrac{1}{n+2}}}}}$.
19. Запрограмуйте обчислення члена послідовності a_n за його номером $a_0 = \delta, a_1 = \lambda, a_{n+2} = \alpha \cdot a_n + \beta \cdot a_{n+1}; n = 0,1,...$ ($\delta, \lambda, \alpha, \beta$ – параметри).
20. Запрограмуйте обчислення довільного члена послідовності a_n, заданої рекурентними співвідношеннями
$a_0 = 1, a_1 = 0, a_2 = 1; a_{n+3} = a_n + 2a_{n+1} + 3a_{n+2}; n = 0,1,..$.
21. Запрограмуйте а) обчислення; б) швидке обчислення; в) рекурсивне обчислення; г) рекурсивне швидке обчислення довільного члена послідовності, заданої рекурентними співвідношеннями
$a_0 = 1, a_1 = 1, a_2 = 1; a_{n+3} = a_n + a_{n+1} + a_{n+2}; n = 0,1,..$.
22. Запрограмуйте швидке обчислення за рекурентними співвідношеннями послідовності $a_0 = 5$, $a_1 = 10$, $a_{n+1} = \frac{a_n}{10} + \frac{a_{n-1}}{5}$. Запропонуйте спосіб перевірки правильності результату.

23. Запрограмуйте цикл для обчислення полінома Чебишева $T_0(x)=1; T_1(x)=x; T_n(x)=2xT_{n-1}(x)-T_{n-2}(x); n=2,3,...$ заданого степеня n в довільній точці x.
24. Для довільних значень початкових членів послідовності, заданої рекурентним співвідношенням $x_n = x_{n-2} + x_{n-4}$, запрограмуйте обчислення суми її перших n членів.
25. Обчисліть арифметико-гармонійне середнє чисел $a=a_0$ і $b=b_0$, що визначається як спільна (довести!) границя послідовностей
$$a_{n+1}=\frac{2a_nb_n}{a_n+b_n}; b_{n+1}=\frac{a_n+b_n}{2}.$$
26. Обчисліть гармонійно-геометричне середнє чисел $a=a_0$ і $b=b_0$, що визначається як спільна (довести!) границя послідовностей
$$a_{n+1}=\frac{2}{\frac{1}{a_n}+\frac{1}{b_n}}; b_{n+1}=\sqrt{a_nb_n}; n=1,2,...$$
27. Складіть програму, що визначатиме максимально можливу кількість а) десяткових; б) вісімкових; в) шістнадцяткових цифр в значенні типу 1) **`float;`** 2) **`double`**.
28. Підрахуйте кількість одиниць у двійковому машинному слові.
29. Підрахуйте кількість нулів у двійковому 32-розрядному машинному слові.
30. Підрахуйте кількість старших нулів у машинному слові.
31. За довільним однозначним десятковим числом `x` підрахуйте кількість входження відповідної йому цифри до десяткового запису заданого натурального числа `n`.
32. Один указник указує на ціле число, інший на дійсне. Обміняйте значення цих указників.
33. Створіть масив дійсних чисел і введіть значення його елементів з клавіатури; одержані значення виведіть на екран.
34. Введіть із клавіатури розмірність масиву дійсних чисел, створіть його та введіть значення його елементів з клавіатури; одержані значення виведіть на екран.
35. Знайдіть в заданому масиві задане число.
36. Впорядкуйте елементи масиву a) за зростанням; b) за спаданням.
37. Знайдіть в заданому впорядкованому за зростанням масиві задане число.
38. Введіть рядок символів із клавіатури. Інвертуйте рядок.

39. Перевірте, чи є рядок паліндромом (однаково читається зліва направо і справа наліво)
40. Знайдіть в заданому рядку номери заданої літери.
41. Знайдіть в заданому рядку номери кількох заданих літер.
42. Замініть всі входження заданої літери в заданий рядок іншою заданою літерою.
43. Знайдіть в заданому рядку задане слово.
44. Закодуйте текст шляхом перестановки кожної пари двох сусідніх символів.
45. Перетворіть натуральне число на рядок символів, що складатиметься з його цифр у шістнадцятковому запису.
46. Загорнутий вектор v складено з пар: ключ, значення. Визначте структури та запрограмуйте пошук запису із заданим значенням ключа a.
47. Запрограмуйте пошук у загорнутому впорядкованому векторі a всіх членів загорнутого впорядкованого вектора b.
48. Виконати циклічний зсув машинного слова на а) один байт; б) два байти вправо.
49. Двома способами запрограмуйте циклічний зсув машинного слова вліво на один байт. Визначте судження для перевірки правильності зсуву.
50. Запропонуйте заміну заданого байта чотирибайтового машинного слова іншим значенням.
51. Запрограмуйте впорядкування байтів чотирибайтового машинного слова за зростанням їхніх значень, використовуючи підхожі структури даних.
52. Знайдіть площу опуклого многокутника на площині, заданого послідовністю вершин, впорядкованих в напрямку їх обходу.
53. Визначте структуру списку дійсних чисел. Запрограмуйте пошук найменшого серед дійсних чисел, що складають список.
54. Запрограмуйте вставлення дійсного числа а) в початок; б) у кінець списку дійсних чисел.
55. Знайдіть координати центру ваги системи з заданої кількості матеріальних точок у просторі. Маса кожної точки відома.

3. Процедурно-орієнтоване програмування

Багатопарадигменість мови C++ дає розробникові змогу добирати якнайприйнятніші архітектури створюваних її засобами програмних систем. Така система узгоджується з об'єктно-орієнтованою парадигмою, максимально спираючись на всі її здобутки та переваги. Однак це не означає, що для кожної програмної одиниці обов'язково потрібно створювати обгортку у вигляді класу, який формально являє собою основну сутність об'єктно-орієнтованого програмування, доступну ззовні. Такий підхід виявився б невиправданим хоча б тому, що стало б недоступним усе програмне забезпечення, раніше розроблене засобами мови C. Отже, перша причина використання в мові C++ процедурно-орієнтованої парадигми полягає в забезпеченні її сумісності з мовою програмування C.

Друга причина, як це не дивно, зумовлена особливостями власне об'єктно-орієнтованого програмування, у якому поряд із функціями, вбудованими до класів як члени, можна використовувати зовнішні позакласові функції, які відіграють роль так званих службових функцій або утиліт класу. Виявляється, що при цьому деякі фундаментальні властивості класів, наприклад інкапсуляція, лише посилюються; про це детально йтиметься у відповідних підрозділах.

Нарешті, функції – члени класів – самі виявляються не чим іншим, як процедурами, хоча й наділеними особливими правами доступу. Тому вміння створювати зрозумілі програмні коди окремих функцій залишається важливим фактором, що й надалі характеризуватиме кваліфікацію розробника імперативних програм.

3.1. Функції

Із матеріалу попереднього розділу випливає, що процедурна програма складається з окремих функцій, які, в свою чергу, можуть містити блоки. Як у мові C, так і в C++ одна функція не може бути вкладена в іншу. У кожної функції має бути *визначення* (function definition), у якому містяться її *заголовок* (function declaration) і *тіло* (function body). Визначення повністю характеризує функцію: описує типи її *формальних параметрів* і *результату*, а також містить реалізацію – набір команд, наприклад такий:

Лістинг 3.1

```
// Приклад визначення функції
// Алгоритм Евкліда пошуку найбільшого спільного дільника
int gcd(int m, int n) {
   while(m !=n)
```

```
      if(m>n) m=m-n;
      else  n=n-m;
   assert(m==n);
   return m;
}
```

Реалізація потрібна для того, щоб компілятор побудував об'єктний код функції.

3.1.1. Визначення й оголошення функцій

Кожну функцію визначають один раз в одному з файлів реалізації, що має розширення **.cpp**, а використовують за допомогою викликів у цьому самому чи інших файлах. Оскільки кожен із файлів, узагалі кажучи, може бути відкомпільований окремо, то компілятору має бути доступна інформація про використані там позначення, зокрема про функції, які викликаються. Якщо визначення функції міститься перед викликом у тому самому файлі, то на момент обробки виклику компілятор має всю інформацію про функцію.

Лістинг 3.2

```
// Приклад визначення функції безпосередньо
// перед її використанням
// Алгоритм Евкліда пошуку найбільшого спільного дільника
int gcd(int m, int n) {
   while(m!=n)
      if(m>n) m=m-n;
      else n=n-m;
   assert(m==n);
   return m;
}
// Використання щойно визначеної функції
int main() {
// З визначення функції gcd видно, що її параметри -
// цілі числа, і результат - також
   cout<<gcd(360, 396)<<endl;
   return 0;
}
```

Спроба розмістити визначення функції після її виклику призводить до виникнення синтаксичної помилки.

Лістинг 3.3

```
int main() {
// Спочатку виклик
   cout<< gcd(360, 396)<<endl;
```

```
   return 0;
}
// Алгоритм Евкліда пошуку найбільшого спільного дільника
int gcd(int m, int n) {
   while(m !=n)
      if(m>n) m=m-n;
      else n=n-m;
   assert(m==n);
   return m;
}
```

```
error C2065: 'gcd' : undeclared identifier
```

У системах програмування C/C++ потрібно попередньо оголошувати сутності до першої спроби їх використання. Помилку усувають, помістивши перед викликом *випереджальне оголошення* функції (forward declaration):

Лістинг 3.4

```
// Випереджальне оголошення
int gcd(int m, int n);
// Використання попередньо оголошеної функції
int main() {
// З оголошення функції gcd видно, що її параметри –
// цілі числа, і результат – також
   cout<< gcd(360, 396)<<endl;
   return 0;
}
// Приклад визначення попередньо оголошеної функції
// Алгоритм Евкліда пошуку найбільшого спільного дільника
int gcd(int m, int n) {
   while(m !=n)
      if(m>n) m=m-n;
      else n=n-m;
   assert(m==n);
   return m;
}
```

Найчастіше для компіляції виклику однієї функції, наприклад `gcd()`, усередині іншої (у нашому прикладі – `main()`) компілятору не потрібен код самої функції. Йому достатньо лише інформації про кількість параметрів і їх типи, а також про тип результату, оскільки він реалізує виклик як послідовність дій, котрі можна поділити на три групи.

До першої групи належить обчислення значень фактичних параметрів і поміщення їх у стек.

Друга група дій – запам'ятовування точки повернення після виклику й організація особливого переходу в точку входу до викликаної функції, який супроводжується зчитуванням значень параметрів зі стека. Це так званий *перехід до підпрограми* (go to subroutine), або *перехід із поверненням*, яке відбувається за допомогою виконання *команди виходу* (return statement) у точку безпосередньо за місцем виклику. Поверненню може передувати занесення результату функції у стек або регістр. Точку входу до викликаної функції компілятор позначає як зовнішню відсилку, тобто відсилку до сутності, що перебуває поза межами створюваного компілятором об'єктного коду. На наступному етапі підготовки програми до виконання *компонувальник* (linker), *розв'язуючи* ці відсилки (linkage), з'ясовує, де містяться всі зовнішні відсилки.

Третя група дій – це вилучення результату зі стека та передавання його за місцем призначення.

Такий підхід не містить нічого нового порівняно зі звичним способом виклику функцій і підпрограм в інших мовах програмування. Щоправда, є виняток із цього правила реалізації викликів функції – так звані відкриті функції, які ми розглянемо в наступному підрозділі.

Згідно з концепцією роздільної компіляції, яка, власне, і є головною особливістю компіляції функцій у мові C, саме визначення може міститися в іншому файлі, оскільки воно не потрібне для компіляції виклику.

Говорять, що оголошення функції задає її *сигнатуру* (prototype), яка визначає типи результату й кожного з параметрів, але не залежить від їхніх імен. Оголошення функції обчислення найбільшого спільного дільника, наведеної на початку розділу, ми записали як

```
int gcd(int m, int n);
```

а її сигнатуру –

```
int gcd(int, int);
```

Її читають так: ціла функція від двох цілих параметрів. Сигнатуру можна використовувати і як оголошення, але в такому разі імена параметрів будуть лише коментарями, що пояснюють їх призначення.

Виконання функції завершується командою *виходу*

```
return expression;
```

Тип виразу `expression` має бути узгоджений (у найпростішій ситуації – збігатися) з типом функції. Його значення – результат виконання функції.

Функція може не повертати значення: тоді тип її результату в сигнатурі позначають ключовим словом **void**, а команда виходу не містить виразу.

```
void showPrice(int cost) {
   cout<<"Hrn "<<cost/100<<','<<cost%100<<endl;
   return;
}
```
Лістинг 3.5

Команда виходу, узагалі кажучи, необов'язкова для функцій без результату: її можна пропускати, але все ж надійніше подавати для того, щоб підкреслити, що саме тут її автор передбачив закінчення.

Складаючи заголовні файли, зазвичай записують оголошення функцій разом з іменами параметрів. Раніше вже йшлося про значущість імен у програмах. Імена параметрів можуть стати найкращим коментарем до свого призначення, що особливо важливо, коли параметри однотипні, але різні за призначенням. Так, у прикладі, що містить функцію обчислення кореня з дійсного числа, її оголошення

```
double root(double x, double eps);
```

зрозуміліше, ніж сигнатура

```
double root(double, double);
```

Із нього видно, що перший параметр задає підкореневий вираз, а другий – точність.

3.1.2. Відкриті та закриті функції

За способом виклику розрізняють функції, реалізовані *відкритим* (вбудованим, *inline*) і *закритим* способом.

Перший термін можна пояснити на такому прикладі. Розглянемо таку функцію:

```
int max(int x, int y) {
   return (x>y)? x :y;
}
```
Лістинг 3.6

У лістингу 3.7 наведено фрагмент програми, у якій її використано.

```
int a=100, b=120;
int c=max(a+b, 300); c=max(max(b-a, 50), c-a);
```
Лістинг 3.7

Перепишемо цей фрагмент, підставивши замість викликів функції блоки її обчислення.

```
int a=100, b=120;
// Перший виклик функції max(a+b, 300)
int tmp1; {int x=a+b, y=300; tmp1=(x>y)? x :y;}
int c=res;
```
Лістинг 3.8

```
// Другий виклик функції max(b-a, 50)
int tmp2; {int x=b-a, y=50; tmp2=(x>y)? x :y;}
// Третій виклик функції max(max(a+b, 300), c-a)
int tmp3; {int x=tmp2, y=c-a; tmp3=(x>y)? x :y;}
c=tmp3;
```

Тепер виникає запитання: якому коду з наведених у лістингах 3.7 і 3.8 надати перевагу? Коли йдеться про розуміння коду, то, ясна річ, що першому. Із нього зрозуміліше, що саме обчислюється. Якщо ж говорити про ефективність виконання, то все не так очевидно. Перший лістинг містить багато прихованих команд, що обслуговують передавання параметрів у стек та вилучення їх із нього, а також забезпечують повернення результату. У другому лістингу таких команд немає, проте сам код тіла функції повторюється стільки разів, скільки є викликів.

Для порівняння розглянемо об'єктний код, що відповідає відкритому та закритому варіантам функції.

Лістинг 3.9

```
int max(int x, int y) {
00402A90 push ebp
00402A91 mov  ebp,esp
00402A93 push ecx
00402A94 mov  dword ptr [ebp-4],0CCCCCCCCh
   return (x>y)? x : y;
00402A9B mov  eax,dword ptr [x]     // (1)
00402A9E cmp  eax,dword ptr [y]     // (2)
00402AA1 jle  max+1Bh (402AABh)     // (3)
00402AA3 mov  ecx,dword ptr [x]     // (4)
00402AA6 mov  dword ptr [ebp-4],ecx // (5)
00402AA9 jmp  max+21h (402AB1h)     // (6)
00402AAB mov  edx,dword ptr [y]     // (7)
00402AAE mov  dword ptr [ebp-4],edx // (8)
00402AB1 mov  eax,dword ptr [ebp-4] // (9)
}
00402AB4 mov  esp,ebp
00402AB6 pop  ebp
00402AB7 ret
```

Це асемблерний код, створений за визначенням функції `max(int x, int y)`. Перші чотири його команди забезпечують розбір параметрів, останні три – повернення результату й вихід. Отже, у лістингу 3.9 сім службових команд. Команди ж (1)–(9) відповідають власне за обчислення імплікації `(x>y)? x : y`.

Лістинг 3.10

```
int max(int x, int y) {
```

```
   return (x>y)? x : y;
}
```

А лістинг 3.11 містить код виклику функції.

Лістинг 3.11

```
int c=max(a, b);
00402AE4 mov  eax,dword ptr [b]
00402AE7 push eax
00402AE8 mov  ecx,dword ptr [a]
00402AEB push ecx
00402AEC call max (40100Ah)
00402AF1 add  esp,8
00402AF4 mov  dword ptr [c],eax
```

Отже, під час кожного виклику функції додатково до коду функції виконується код її виклику, наведений у лістингу 3.11. Маємо ще сім службових команд додатково до наведених у лістингу 3.9. Безпосередньо ж застосувавши імплікацію замість виклику функції, одержимо в лістингу 3.12 ті самі дев'ять команд, що й у лістингу 3.9, а також одну команду ініціалізації змінної `c`.

Лістинг 3.12

```
int c=(a>b)? a : b;
00401093 mov  edx,dword ptr [a]
00401096 cmp  edx,dword ptr [b]
00401099 jle  main+93h (4010A3h)
0040109B mov  eax,dword ptr [a]
0040109E mov  dword ptr [ebp-48h],eax
004010A1 jmp  main+99h (4010A9h)
004010A3 mov  ecx,dword ptr [b]
004010A6 mov  dword ptr [ebp-48h],ecx
004010A9 mov  edx,dword ptr [ebp-48h]
004010AC mov  dword ptr [c],edx
```

Що ж нам дало використання функції обчислення максимуму? На перший погляд, лише зайві 14 команд, з яких дев'ять корисні. Проте зауважимо, що програма, що ґрунтується на викликах функції, значно виразніша, а тому зрозуміліша, ніж та, де обчислення вбудовано безпосередньо в код (порівняйте, наприклад, лістинги 3.16 і 3.17).

Відкриті функції дають змогу поєднати виразність з ефективністю. Розглянемо таку функцію:

Лістинг 3.13

```
inline int max(int x, int y) {
   return (x>y)? x : y;
}
```

Її виклик формально має такий самий вигляд, що й для закритого варіанта, але код принципово відрізняється.

Лістинг 3.14

```
int c=max(a, b);
00402AB4 mov  eax,dword ptr [a]
00402AB7 cmp  eax,dword ptr [b]
00402ABA jle  main+34h (402AC4h)
00402ABC mov  ecx,dword ptr [a]
00402ABF mov  dword ptr [ebp-40h],ecx
00402AC2 jmp  main+3Ah (402ACAh)
00402AC4 mov  edx,dword ptr [b]
00402AC7 mov  dword ptr [ebp-40h],edx
00402ACA mov  eax,dword ptr [ebp-40h]
00402ACD mov  dword ptr [c],eax
```

Як бачимо, тепер це вже десять команд виконання тієї самої імплікації, які в точності повторюють наведені раніше команди прямого виконання імплікації

```
int c=(a>b)? a :b;
```

Відкриті функції називають також *вбудованими* (inline function), тому що їх текст, як видно з асемблерного коду, просто вбудовано в місця виклику. Тому тіло функції повторюється в кожному місці виклику. Не варто замислюватися над тим, чи виправдовують себе вбудовані функції за ефективністю: можна лише приблизно уявляти собі, як кожен конкретний компілятор оптимізує код, з якими накладними витратами пов'язане записування та зчитування параметрів і результатів. Сучасні компілятори самостійно приймають рішення про доцільність або недоцільність відкритої реалізації функції. Слід також зважати на те, що на момент компіляції виклику компілятор має знати не лише сигнатуру функції, але і її тіло. Тому функції, які заплановано реалізувати відкрито, потрібно не декларувати, а визначати прямо в заголовних файлах:

Лістинг 3.15

```
// MyHeader.h
inline int max(int x, int y) {return (x>y)? x : y;}
```

Можна залишити компілятору вказівку про намір реалізувати функцію відкритою підстановкою за допомогою ключового слова `inline`, але він не зобов'язаний виконувати її.

Вправа 3.1

Методом комп'ютерного експерименту спробуйте на вашому компіляторі реалізувати функції відкритим способом. Знайдіть потрібні для

цього значення параметрів компіляції. Для перевірки увімкніть налагоджувач і перегляньте асемблерний код команди виклику.

Увазі до вбудованих функцій перш за все варто протиставити традиційну схильність апологетів мови C до макровизначень. На перший погляд вони схожі на відкриті функції; різниця полягає в способі передавання значень параметрів.

Перепишемо у вигляді макровизначення функцію обчислення максимуму:

```
#define MAX(x,y)(((x)>(y))?(x):(y))
```

За традицією, що склалася, пишемо назву макроса MAX великими літерами. Не варто забувати, що для уникнення проблем із пріоритетами операцій у виразах, що з'являються після підставляння параметрів, слід використовувати додаткові дужки. Тепер повернімося до прикладу з лістингу 3.7, замінивши виклики функції макровикликами.

Лістинг 3.16

```
int a=100, b=120;
int c=MAX(a+b, 300); c=MAX(MAX(b-a, 50), c-a);
```

Препроцесор виконає текстову підстановку параметрів, і на вхід компілятора надійде приблизно такий текст, як у лістингу 3.17 (не точно такий, тому що компілятор не зберігає тексту, одержаного після його обробки препроцесором).

Лістинг 3.17

```
int a=100, b=120;
int c=(a+b>300)? a+b : 300;
c=((b-a>50?b-a:50)>c-a?(b-a>50?b-a:50)c-a);
```

Це, власне, відповідає нашим намірам.

Однак текстова підстановка параметрів виявляється не завжди коректною. Візьмемо, наприклад, інше макровизначення – піднесення числа до квадрата:

```
#define SQUARE(a)((a)*(a))
```

Спробуємо застосувати його таким способом:

Лістинг 3.18

```
int x=1; SQUARE(++x);
```

Результат важко передбачити: наприклад, комп'ютер автора видав 9, тоді як звичайна функція піднесення до квадрата, наведена в лістингу 3.19, за тих самих значень параметра дасть правильний результат 4.

Лістинг 3.19

```
inline int square(int a) {return (a*a);}
```

Річ у тім, що текстове підставляння виявилося непридатним для повторюваного параметра, тому параметр `a` й відповідно вираз `++x` обчислено двічі за кількістю використань у макровизначенні, тоді як виклик функції обчислює його один раз під час передавання параметра.

Вправа 3.2

Проаналізуйте передавання параметра в реалізації піднесення до куба за допомогою макроса

```
#define CUBE(a) ((a)*(a)*(a))
```

Висновок. *Не користуйтеся макровизначеннями, а якщо вам доведеться мати справу з програмами, де вони є, то ретельно аналізуйте їх.*

Щойно розглянуті нами відкриті підстановки застосовують для реалізації невеликих функцій. Справді, такі підстановки дають змогу уникнути зайвих витрат на сам виклик, але ж код функції повторюється в програмі для кожного її виклику. Якщо код великий, то його дублювання в кожному місці виклику виявляється витратним, а в разі використання рекурсії – просто неможливим. Тому основним способом реалізації виклику функції вважають так звану *закриту функцію*. Вона закрита тому, що її тіло може залишатися невідомим (закритим) для компілятора, а об'єктний код – міститись окремо (узагалі кажучи, у невідомому для компілятора місці), на відміну від відкритого, коли тіло функції відкрите й міститься там, де його викликають.

Виклик закритої функції відбувається у визначеній у попередньому розділі послідовності. Уявімо собі, що в програмі використано три змінні `x`, `y` і `z`, перші дві з яких – фактичні параметри виклику, а остання служить для розміщення результату. Перед викликом проходить ініціалізація формальних параметрів `m` і `n` значеннями фактичних. Умовно на рис. 3.1 це позначено присвоєннями `m=x; n=y`. Сам код функції доступний за її іменем, за яким і виконується перехід до підпрограми: на рис. 3.1 це команда `gosub gcd`. Запам'ятовується адреса місця повернення, яку позначено як `sto gcd_return`. На виході забезпечено можливість повернення результату (на рис. 3.1 це умовно позначено присвоєнням `z=m`). Нарешті, виконується сам вихід, який ми позначили як `goto gcd_return`.

Насправді виклик відбувається дещо складніше, оскільки в місці його виконання невідомі імена параметрів (говорять, що їх локалізовано у функції), а функції невідомо, куди треба повертати результат. Тому в передаванні параметрів може брати участь іще одна допоміжна ланка – стек. У лістингу 3.20 наведено асемблерний код виклику функції `gcd` (див. її визначення в лістингу 3.1).

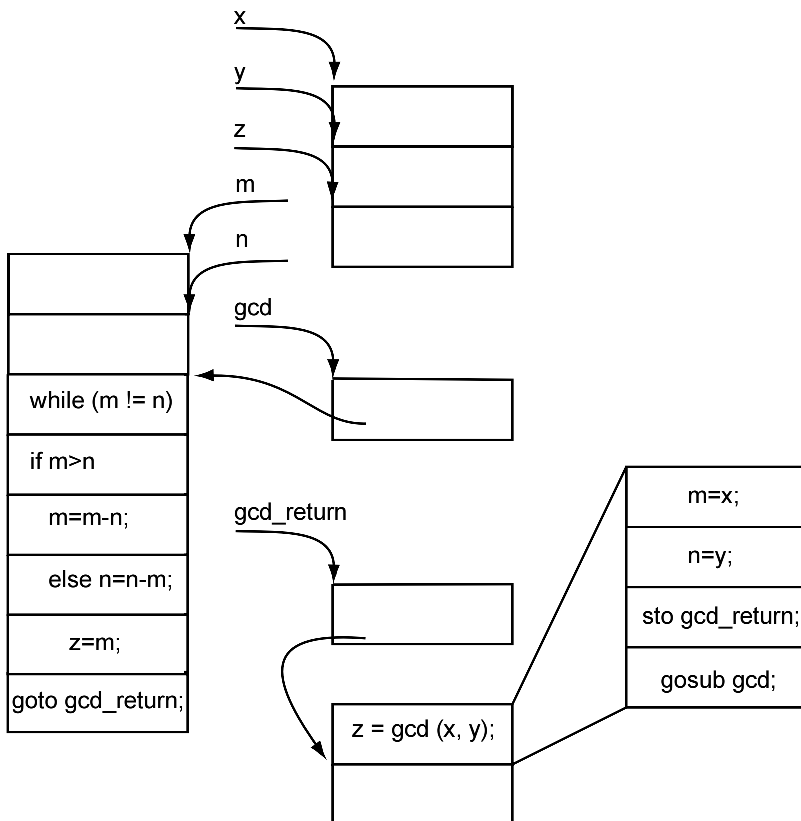


Рис. 3.1. Схема виклику функції

Лістинг 3.20

```
c=gcd(75,25);
004017CF push 19h
004017D1 push 4Bh
004017D3 call @ILT+480(gcd)(004011e5)
004017D8 add  esp,8
004017DB mov  dword ptr [ebp-18h],eax
```

Спочатку буде обчислено та занесено у стек значення фактичних параметрів (цим діям відповідають команди `push` у коді на асемблері, наведеному в лістингу 3.20), далі виконується команда переходу до підпрограми (в асемблерному коді – `call`), дія якої не зводиться лише до переходу, а

полягає також у запам'ятовуванні місця для повернення після завершення виконання підпрограми. Після команди виклику в коді розміщено команди вибору результату і його передачі за місцем призначення.

А ось результат компіляції самої функції (лістинг 3.21). Немає потреби детально вчитуватись у весь текст, але корисно порівняти частину коду, яка відповідає власне командам функції, із тією, що передає управління (у коді її виділено курсивом). Це команди ініціалізації, розміщені між заголовком функції та її першим оператором, і команди виходу зі збереженням результату, що містяться після останнього оператора.

Лістинг 3.21

```
   int gcd(int m, int n) {
004018E0 push ebp
004018E1 mov ebp,esp
004018E3 sub esp,40h
004018E6 push ebx
004018E7 push esi
004018E8 push edi
004018E9 lea edi,[ebp-40h]
004018EC mov ecx,10h
004018F1 mov eax,0CCCCCCCCh
004018F6 rep stos dword ptr [edi]
    while(m != n)
004018F8 mov eax,dword ptr [ebp+8]
004018FB cmp eax,dword ptr [ebp+0Ch]
004018FE je   gcd+3Eh(0040191e)
      if(m>n)
00401900 mov ecx,dword ptr [ebp+8]
00401903 cmp ecx,dword ptr [ebp+0Ch]
00401906 jle gcd+33h(00401913)
        m=m-n;
00401908 mov edx,dword ptr [ebp+8]
0040190B sub edx,dword ptr [ebp+0Ch]
0040190E mov dword ptr [ebp+8],edx
      else
00401911 jmp gcd+3Ch(0040191c)
        n=n-m;
00401913 mov eax,dword ptr [ebp+0Ch]
00401916 sub eax,dword ptr [ebp+8]
00401919 mov dword ptr [ebp+0Ch],eax
0040191C jmp gcd+18h(004018f8)
    return m;
00401A8E mov eax,dword ptr [ebp+8]
   }
```

```
00401A91 pop edi
00401A92 pop esi
00401A93 pop ebx
00401A94 mov esp,ebp
00401A96 pop ebp
00401A97 ret
```

Тому насправді під час виклику функції `z=gcd(x,y)` ініціалізацію формальних параметрів значеннями фактичних (наприклад, `m=x`) розподілено: у місці виклику виконується команда «помістити значення цілого виразу `x` у стек», а в підпрограмі використовуються значення зі стеку (`[ebp+8]` і `[ebp+0Ch]`). У місці повернення значення змінної `m` записується в регістр `eax` і через нього повертається в програму.

Порівнюючи обидва механізми виклику функцій, зазначимо, що закриті функції зручні тим, що їх об'єктний код міститься в програмі в одному екземплярі. Однак додаткові команди керування зумовлюють *активізацію функції*, *перехід* до виконання її коду та подальше *повернення* в місце продовження обчислень після виклику. Це може бути складніше, ніж виконати сам код функції. Можна також розміщувати код функції безпосередньо в місці виклику, тобто використовувати відкриті функції. Однак тоді код функції буде дублюватись у кожному місці виклику. Проте нагадаємо, що в будь-якому разі компілятор сам вирішує, як реалізувати функцію (визначення вбудованих функцій – лише порада).

3.2. Механізми передавання параметрів

Програмна функція, передусім закрита, певною мірою являє собою «річ у собі», бо виконується незалежно, взаємодіючи з викликаючим кодом за допомогою передавання параметрів і результату. Проте, звісно, є й «безрезультатні» функції, які, крім цього, можуть ще й не мати параметрів. У такому разі функції можуть взаємодіяти з місцем виклику завдяки іншим механізмам, наприклад через спільну пам'ять (глобальні величини), хоча обмін даними через спільну пам'ять вважають поганим стилем у програмуванні.

А ось приклад функції, яка не потребує комунікації з місцем виклику, оскільки використовує лише системні ресурси.

Лістинг 3.22

```
#include <iostream>
void hello(void) {
   std::cout<<"Hello everybody!"<<std::endl;
}
```

Її оголошення

```
void hello(void);
```

має передувати місцю виклику. Ключове слово `void` у списку параметрів можна пропускати, записуючи у визначенні

Лістинг 3.23

```
#include <iostream>
void hello() {
   std::cout<<"Hello everybody!"<< std::endl;
}
```

й оголошенні

```
void hello();
```

порожні дужки. Для типу результату позначення `void` є обов'язковим.

Функції без результатів і параметрів – швидше виняток, аніж правило, тому розглянемо способи налаштування зв'язків функцій з їх оточенням під час виклику.

3.2.1. Передавання параметрів значенням

Почнемо з найпростішого випадку – функції з одним параметром довільного *арифметичного типу*. Цей тип виділяємо окремо, тому що механізми передавання параметрів інших типів потребують додаткових роз'яснень. Для простоти розглянемо оголошення функції `f` з одним параметром. Якщо їх більше, то це не впливає на спосіб обробки кожного з них. Тому, розглядаючи параметри різних типів, використовуватимемо умовні позначення відповідних функцій для пояснення механізму передавання параметра. Якщо позначити довільний арифметичний тип параметра як `AT` («А» означає «арифметичний»), то оголошення функції з одним параметром цього типу та порожнім результатом записуватимемо як

```
void f(AT x);
```

а її тип скорочено позначатимемо (`AT→`**`void`**).

Виклик цієї функції має вигляд `f(e)`; тут `e` – вираз відповідного типу (фактичний параметр): як `AT`, так й іншого типу, який можна звести до `AT`.

Скажімо, функцію привітання можна доповнити параметром:

Лістинг 3.24

```
#include <iostream>
void hello(int n) {
   std::cout<<"Hello! Your registration is:"<<n<<std::endl;
}
```

Загалом функції типу `(AT→`**`void`**`)` мають сенс, якщо вони виводять відформатований певним способом результат або записують його до файлу.

Згадаймо, наприклад, функцію showPrice (лістинг 3.5).

Як уже відомо, виконання функції починається з визначення локальної змінної для параметра x із її одночасною ініціалізацією:

```
AT x=e;
```

Ініціалізація відбувається за звичайними правилами: значення виразу e буде обчислено та записано в місці, відведеному для змінної x. Саме тому цей механізм передавання параметрів називають *передаванням параметрів значенням* (call-by-value). Воно має багато переваг, головна з яких – висока надійність, котра полягає в тому, що підпрограма не має виходу за рамки своєї локальної області видимості, а тому сферу її впливу добре локалізовано. Якщо розглядати вдало спроектовані підпрограми, що не користуються глобальними об'єктами, то єдиний спосіб впливу підпрограми з параметрами арифметичного типу на зовнішнє середовище – взаємодія зі стандартними зовнішніми пристроями чи повернення значення. Його способи обговоримо трохи пізніше. Схемою ж, яку розглядаємо зараз, повернення значення поки що не передбачено, оскільки тип результату було позначено як **void**.

Без розуміння механізмів передавання параметрів легко помилитися. Наведемо приклад простої «функції-обманки», яка, власне, нічого корисного не робить.

Лістинг 3.25

```
void doubleMyValue(int valueParam) {
   valueParam*=2; // Значення формального параметра подвоюється
   cout<< valueParam <<'\n'; // Пересвідчимося в цьому
   return;
}
int main() {
   int number=10; cout<<number<<'\n'; // Буде виведено 10
   doubleMyValue(number);
// Подвоєння формального параметра не вплинуло на фактичний
   cout<<number<<'\n'; // Знову буде виведено 10
   return 0;
}
```

У процесі виконання підпрограми значення формального параметра valueParam подвоїться, але це не матиме жодного впливу на фактичний параметр number. Сам же параметр valueParam, локалізований у тілі функції, після завершення її виконання перестане існувати. Складена таким способом програма стає непотрібною. Пізніше ми обговоримо заходи, спрямовані на підвищення міри відповідності намірів їх реалізації.

Дослідимо тепер функції вигляду (AT→AD) з арифметичним парамет-

ром і арифметичним результатом. Уведемо позначення:

```
AD f(AT x);          (AT→AD)
```

Тепер AT й AD обидва позначають арифметичні типи, але можливо, різні, а попередній приклад можна записати так:

Лістинг 3.26

```
int doubleMyValue(int valueParam) {
   valueParam=valueParam*2; // Значення формального параметра подвоюється
   return valueParam; // Подвоєне значення повертається як результат функції
}
int main() {
   int number=10; number=doubleMyValue(number);
   cout<<number<<'\n'; // Буде виведено 20
   int n=10, m=10;
   cout<<n<<','<<m<<'\n'; // Буде виведено 10, 10
   n=doubleMyValue(m);
   cout<<n<<','<<m<<'\n'; // Буде виведено 20, 10
// Значення результату змінилося, значення параметра залишилося незмінним
   return 0;
}
```

Власне, не було навіть потреби змінювати значення формального параметра в тілі функції (його все одно видалять), достатньо передати назовні потрібний вираз:

Лістинг 3.27

```
int doubleMyValue(int valueParam) {
   return valueParam*2;
}
```

А ось приклад, узятий зі знаменитої книги Брайана Кернігана та Денніса Рітчі [2], який демонструє ще одну «функцію-обманку». Він містить функцію з двома параметрами арифметичного типу, що має вигляд (ATxAT→**void**).

Лістинг 3.28

```
// Обмін значень формальних, а не фактичних параметрів
void swap(double x, double y) {
   double z=x; x=y; y=z;
   return;
}
int main() {
   double a=1; double b=2;
   cout<<"Перед swap():\ta: "<<a<<"\tb: "<<b<<endl;
   swap(a, b); cout<<"Після swap():\ta: "<<a<<"\tb: "<<b<<endl;
   return 0;
}
```

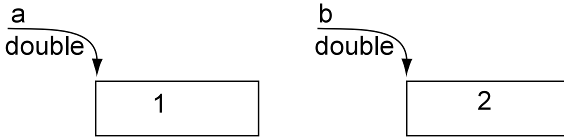


Рис. 3.2. Значення фактичних параметрів до і після виклику `swap`

Результат виконання програми підтверджує, що параметри передаються значеннями, і тому зміна значення формального параметра ніяк не впливає на фактичний. Захист фактичних параметрів виявився надійним, і в цьому сенсі мети передавання параметрів значенням досягнуто, проте не виконано завдання, поставлене перед функцією `swap` (рис. 3.2, 3.3).

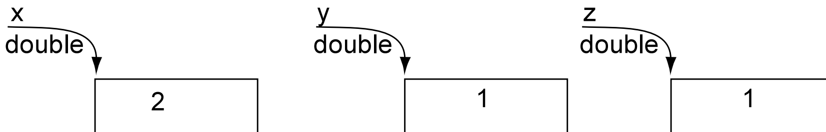


Рис. 3.3. Значення формальних параметрів при виході з функції `swap`

```
Перед swap(): a: 1 b: 2
Після swap(): a: 1 b: 2   // Рис. 3.2
Усередині swap():         // Рис. 3.3
```

Продовжимо дослідження типів параметрів і розглянемо інші можливі варіанти. Повернімося до схеми функції з одним параметром, але тепер уже не накладаючи обмеження на тип формального параметра:

```
void f(T x);          (T→void)
```

Спробуємо розглянути загальний випадок. Тепер тип `T` параметра `x`, переданого до функції `f`, може виявитися:

- арифметичним типом;
- структурою (класом);
- указником;
- масивом;
- відсилкою.

Як і раніше, параметр кожного з типів передається значенням, але обчислення цього значення по-різному впливає на стан пам'яті. Найпростіше обчислити значення параметра арифметичного типу, що, власне, було показано раніше. Повну відповідь щодо значень структур можна одержа-

ти лише в наступному розділі, розглядаючи об'єктне програмування. Зараз обмежимося зауваженням про те, що структури передаються значеннями так само, як і арифметичні типи. Повернімося до структури `Point` із попереднього розділу.

Лістинг 3.29

```
struct Point {
   double _x; double _y;
};
```

Функція обчислення відстані від початку координат до заданої структурою точки виконуватиметься звичайним способом:

Лістинг 3.30

```
double mod(Point u) {
   return sqrt(u._x*u._x+u._y*u._y);
}
```

Деталі ж ініціалізації параметра `u` буде досліджено пізніше.

3.2.2. Параметри-указники

Розглянемо таку схему.

```
void f(T* px);           (T*→void)
```

Для простоти тут `T` – арифметичний тип, тому, як і раніше, параметр `px` передаватиметься значенням, але тепер це буде значення указника, тобто фактично адреса в адресному просторі, доступному з точки виклику. Під час виклику `f(pe)` ініціалізація формального параметра `px` значенням фактичного параметра `pe` відбудеться звичайним способом –

```
T* px=pe;
```

за правилами ініціалізації указників. Потрібно лише не забувати, що `pe` має бути адресним виразом. Справді, з огляду на властивості указників після ініціалізації параметра матимемо

```
rvalue(px) == rvalue(pe);
```

Тому обидва указники – формальний і фактичний параметри – установлено на одне й те саме місце, і поки `px` не змінить свого значення (чого не варто робити в тілі функції), виконуватиметься рівність

```
lvalue(*px)==rvalue(px)==rvalue(pe)==lvalue(*pe);
```

Зі зміненням умісту, адресованого за допомогою указника `px`, водночас змінюється вміст, адресований `pe`:

```
rvalue(*px) == rvalue(*pe);
```

Зауваження. *Передаючи параметр як указник, не варто під час виконання функції змінювати його значення. Насправді призначення указника, що використовується як формальний параметр, – надавати доступ до місця локалізації фактичного параметра, тоді як змінення значення указника порушить цей зв'язок. Детальніше про проблеми, які виникають у цьому разі, йдеться в підрозділі 3.2.4.*

Дуже часто виклик набуває вигляду `f(&a)`, де `a` – вираз, у найпростішій ситуації змінна типу `T`. Тоді ініціалізація має вигляд

```
T* px=&a;
```

З використанням параметра-указника функція подвоєння значення стане тепер коректною. Зверніть увагу на адресування та розіменування.

Лістинг 3.31

```
void doubleMyValue(int* valueParam) {
// Подвоюється значення, адресоване водночас
// як формальним, так і фактичним параметром
   *valueParam*=2;
   return;
}
int main() {
   int number=10;
// Передаємо значення адреси змінної
   doubleMyValue(&number); cout<<number<<'\n'; // Буде виведено 20
   return 0;
}
```

Перепишемо також функцію обміну значень змінних, замінивши самі змінні указниками на них.

Лістинг 3.32

```
// Обмін значеннями змінних за їх указниками
void pswap(double* x, double* y) {
   double z=*x; *x=*y; *y=z;
   return;
}
int main() {
   double a=1; double b=2;
   cout<<"Перед swap():\ta: "<<a<<"\tb: "<<b<<endl;
   pswap(&a, &b); cout<<"Після swap():\ta: "<<a<<"\tb: "<<b<<endl;
   return 0;
}
```

Наведена далі ілюстрація (рис. 3.4) пояснює ситуацію: значеннями фактичних параметрів є відповідно адреси змінних `a` та `b`, формальні ж параметри тепер – указники, що одержали безпосередній доступ до вмісту цих змінних.

Тепер результат виконання програми буде таким:

```
Перед pswap(): a: 1 b: 2
Після pswap(): a: 2 b: 1
```

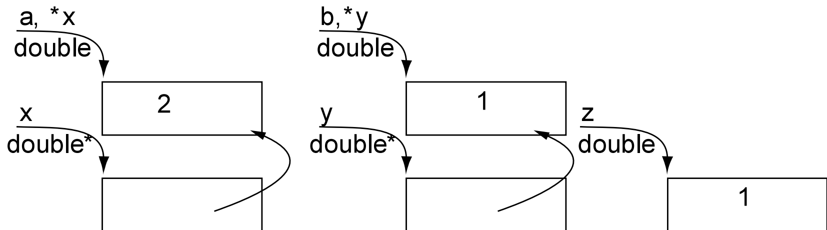


Рис. 3.4. Відповідність між фактичними і формальними параметрами-указниками

Накладні витрати передавання параметра-указника полягають в ускладненні виразу для фактичного параметра. Крім цього, використання указників, як завжди, породжує проблему безпеки.

Вправа 3.3

Поясніть, що станеться, якщо викликати функцію `pswap` так:

```
pswap(0, &b); // ???
```

***Підказка**.* *Згадайте про неозначені указники.*

Розіменовувати указник, не маючи впевненості в його коректності, завжди небезпечно. Можна спробувати захистити себе постійними перевірками під час кожного розіменування, як це зроблено в лістингу 3.33.

Лістинг 3.33

```
// Обмін значеннями змінних за їх указниками
// з перевіркою указників на визначенність
void pswap(double* x, double* y) {
   if((x==0)||(y==0)) return;
   double z=*x; *x=*y; *y=z; cout<<*x<<','<<*y<<endl;
   return;
}
```

Ще складніше боротись із завислими указниками (пам'ятайте про правило гарного тону в разі видалення даних, адресованих за допомогою указника (підрозділ 2.4.2)). Приклад із лістингу 3.34 демонструє, як повноцінний указник `pb` перетворюється на «сміття» через невдалий обмін значеннями.

```
double* pa=new double(100), *pb=new double(200);        Лістинг 3.34
delete pa;
// Використано завислий указник pa
pswap(pa, pb);   // Зіпсовано повноцінний указник pb
cout<<*pb<<endl; // Було виведено-1.46825e+144
```

Виникає запитання: навіщо використовувати указники для передавання параметрів, якщо це так небезпечно? Відповідь дуже проста: використання параметрів-указників – стійка традиція мови C, сумісність із якою було декларовано авторами мови C++.

У наступному підрозділі ми ознайомимося з безпечнішим способом обміну значеннями параметрів – за допомогою відсилок. Оскільки відсилки не належать до засобів мови C, у ній фактично немає іншого способу реалізації функцій, що можуть змінювати значення своїх фактичних параметрів, окрім використання указників (які часто називають параметрами-результатами). Ще одна причина використання указників полягає в їх застосуванні для обробки масивів, коли вони передаються як параметри; про це ітиметься в підрозділі 3.2.4. У практиці програмування мовою C також уживають указники другого рівня (указники на указники), які застосовують для повернення динамічно створених об'єктів (див. підрозділ 3.2.6). Нарешті, за допомогою невизначеного указника можна передати необов'язковий параметр: рівність указника нулю буде сприйнято як те, що немає параметра (однак це не дуже хороша практика).

3.2.3. Параметри-відсилки

Поки що трохи відкладемо дослідження проблем передавання параметрів-масивів, а зараз розглянемо серйозну та надійну альтернативу указникам як параметрам-результатам. Звернімося до схеми з параметром-відсилкою:

```
void f(T& rx);          (T&→void)
```

Знову розглянемо виклик `f(re)`, де `re` – іменований вираз, тобто такий, що може набувати несталого лівобічного значення (у найпростішій ситуації – ім'я типу `T`). Ініціалізація параметра відбувається звичайним для відсилки способом:

```
T& rx=re;
```

Тепер `rx` стає альтернативним іменем для `re` з усією повнотою доступу. Справді, за визначенням відсилки `rx` і `re` йменують одну й ту саму область пам'яті:

```
lvalue(rx) == lvalue(re);
```

а тому їхні значення збігаються:

```
rvalue(rx) == rvalue(re).
```

Знову подивімося, чим завершиться подвоєння параметра.

Лістинг 3.35

```
void doubleMyValue(int& valueParam) {
// Подвоюється значення, іменоване водночас
// як формальним, так і фактичним параметром
   valueParam*=2;
   return;
}
int main() {
   int number=10;
// Передаємо саму змінну для ініціалізації відсилки
   doubleMyValue(number); cout<<number<<'\n'; // Буде виведено 20
   return 0;
}
```

Важливо, що тепер під час виклику зникає потреба в обчисленні адреси, а в самій процедурі – у розіменуванні. Суттєво також, що фактичний параметр не може бути об'єктом, якого не існує, тоді як у разі параметрів-указників завжди є небезпека, що указник набуде нульового значення або буде завислим. Повернімося до функції переставляння параметрів.

Лістинг 3.36

```
// Обмін значеннями змінних за їх відсилками
void rswap(double& x, double& y) {
   double z=x; x=y; y=z;
   return;
}
int main() {
   double a=1; double b=2;
   cout<<"Перед swap():\ta: "<<a<<"\tb: "<<b<<endl;
   rswap(a, b); cout<<"Після swap():\ta: "<<a<<"\tb: "<<b<<endl;
   return 0;
}
```

Ілюстрація цього випадку на диво проста (рис. 3.5). Знову досягнуто бажаного результату.

Дуже корисно виконати комп'ютерний експеримент і поспостерігати за розміщенням у пам'яті фактичних і формальних параметрів у кожному з трьох описаних раніше варіантів функції обміну значеннями параметрів.

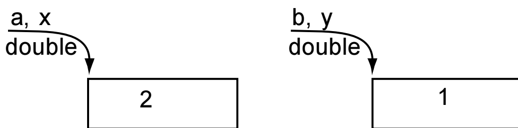


Рис. 3.5. Відповідність між фактичними і формальними параметрами-відсилками

Це можна зробити, наприклад, доповнивши функції обміну значеннями та програму їх тестування виведенням адрес змінних, з якими вони працюють.

Наведемо приклад.

Лістинг 3.37

```
// Обмін змінними за допомогою указників
void pswap(double* x, double* y) {
   double z=*x;*x=*y;*y=z;
// Виведення адрес параметрів
   cout<<"pswap: \t&x: "<<x<<" \t&y: "<<y;
   return;
}
// Обмін змінними за допомогою відсилок
void rswap(double& x, double& y) {
   double z=x; x=y; y=z;
// Виведення адрес параметрів
   cout<<"rswap: \t&x: "<<&x<<" \t&y: "<<&y;
   return;
}
// Безрезультатний обмін змінними
void swap(double x, double y) {
   double z=x; x=y; y=z;
// Виведення адрес параметрів
   cout<<"swap: \t&x: "<<&x<<" \t&y: "<<&y;
   return;
}
int main(){
   double a=10; double b=20;
// Виведення адрес змінних
   cout<<"Main: \t&a: "<<hex<<&a<<"\t&b: "<<&b;
// Виведення початкових значень змінних
   cout<<" Before swap():\ta: "<<a<<"\tb: "<<b<<endl;
   swap(a, b);
// Виведення значень змінних після безрезультатного обміну
   cout<<" After swap():\ta: "<<a<<"\tb: "<<b<<endl;
   pswap(&a, &b);
// Виведення значень змінних після обміну указниками
   cout<<" After pswap():\ta: "<<a<<"\tb: "<<b<<endl;
   rswap(a, b);
```

```
// Виведення значень змінних після обміну відсилками
   cout<<" After rswap():\ta: "<<a<<"\tb: "<<b<<endl;
   return 0;
}
```

Результати виконання цієї програми наведено на рис. 3.6.

Рис. 3.6. Протокол виконання функцій `swap`, `pswap` і `rswap`

Вправа 3.4

Самостійно поекспериментуйте з функцією лістингу 3.38, яка міняє місцями значення двох указників (оголошення **`double`**`*& v` читають справа наліво: `v` – псевдонім указника **`double`**`*` на дійсне число).

Лістинг 3.38

```
void ptrswap(double*& u, double*& v) {
   double* w=v; v=u; u=w;
   return;
}
int main() {
   double a=10; double b=20; double* pa=&i; double* pb=&j;
   cout<<"Перед ptrswap():\tpa: "<<*pa<<"\tpb: "<<*pb<<endl;
   ptrswap(pa, pb);
   cout<<"Після ptrswap():\tpa: "<<*pa<<"\tpb: "<<*pb<<endl;
   return 0;
}
```

Поясніть результат виконання програми:

```
Перед ptrswap: pa: 10 pb: 20
Після ptrswap: pa: 20 pb: 10
```

Зробимо перші висновки.

- **Функції вигляду (`AT→void`).** Якщо параметри належать до базових типів або структур, то у функцію передаються значення фактичних параметрів, і зміна значень формальних параметрів не впливає на фактичні.
- **Функції вигляду (`T*→void`).** Якщо формальний параметр являє собою указник, то відповідна адреса передається значенням, і змінити її неможливо, але можна змінити значення елемента пам’яті, на який цей указник показує. У разі передавання нульового значення указника параметр можна сприйняти як необов’язковий. Викликана функція

завжди має сама відстежувати значення указників і нести відповідальність за проблеми, що можуть виникнути.

- **Функції вигляду (`T&→void`).** Якщо формальний параметр – відсилка, то через нього функція одержує прямий доступ до значення фактичного параметра, до якого відсилає псевдонім. Проблеми необов'язкового параметра чи некоректної адреси не виникає, оскільки програма, що викликає функцію, гарантує правильність фактичного параметра. Деякі автори критично ставляться до передавання параметрів відсилками, уважаючи, що форма виклику, ідентична до передавання параметра значенням, призводить до непорозумінь. Справді, виклики `swap(a, b)` і `rswap(a, b)` не різняться за формою, тоді як виклик `pswap(&a, &b)` явно свідчить про використання адреси. Однак після перегляду оголошення функцій всі питання зникають. Знайти ж відповідь на запитання, чи обов'язковий кожен із параметрів-указників, можна лише з тексту реалізації функції. Це значно складніше завдання.

Зауваження. *Викликаючи функцію, уважно читайте її сигнатуру, адже виклики для передавання значенням вигляду* ***`(AT→void)`*** *і відсилкою вигляду* ***`(T&→void)`*** *не відрізняються один від одного, але зумовлюють принципово різні результати.*

3.2.4. Передавання параметрів-масивів

Масиви – один із найважливіших чинників використання указників для передавання параметрів. Насправді параметр-масив нічим не відрізняється від параметра-указника. Незалежно від того, яким із цих способів його визначено, у функцію буде передано указник.

Повернімося до сигнатури

```
void f(T* px);          // (T*→void)
```

і домовимося, що в усіх випадках, коли указник буде використано для передавання масиву, функція матиме щонайменше два параметри, перший з яких – це указник на масив, а другий – його розмірність:

```
void f(T* px, size_t n);      // (T*,size_t→void)
```

Майже рівнозначна цій сигнатурі така:

```
void f(T[] array, size_t n);       // (T[], size_t→void)
```

Розглянемо кілька сигнатур запропонованих видів. У першій із них за допомогою указника передаватимемо масив. Сигнатура, звісно, не гарантує цього, але наявність другого параметра дає на це прозорий натяк:

```
void init(int* ar, size_t n);
```

У сигнатурі другої функції явно використано позначення масиву (квадратні дужки), що робить зазначений натяк іще прозорішим:

```
void show(int ar[], size_t n);
```

Нарешті, сигнатура третьої функції майже не відрізняється від другої, окрім того, що пропущено імена параметрів:

```
void swapar(int [], size_t);
```

Для виконання комп'ютерного експерименту запропонуємо реалізації для наведених функцій. Нехай, скажімо, перша з них буде функцією ініціалізації масиву. Заповнимо його цілими числами, починаючи з 0.

Лістинг 3.39

```
// Ініціалізація масиву чисел
void init(int* ar, size_t n) {
   for(size_t i=0; i<n; i++)
      ar[i]=i;
}
```

Важко назвати її дуже вдалою, тому що вона спирається на припущення про те, що пам'ять під масив було виділено до виклику. Тому на момент виклику функції масив, який передаватиметься за допомогою фактичного параметра, уже має існувати й містити не менше елементів, ніж вказує другий параметр n. Втім, жодна система програмування не зможе перевірити, чи виконується ця вимога. Окрім того, компілятору байдуже, що немає параметра, який відповідав би за розмір масиву.

Лістинг 3.40

```
// Невдала ініціалізація масиву чисел
void badInit(int* ar) {
   for(size_t i=0; i<1000; i++)
      ar[i]=i;
}
```

Чим закінчиться виклик цієї функції? Важко сказати. Усе залежить від того, скільки пам'яті було виділено під фактичний параметр.

Гіршою може видатися хіба що спроба переспрямувати указник на інше місце чи навіть виділити пам'ять під масив прямо в тілі функції:

Лістинг 3.41

```
// Ініціалізація смітника
void worstInit(int* ar, size_t n) {
// Зв'язок формального параметра з фактичним розірвано
   ar=new int[n];
   for(size_t i=0; i<n; i++)
      ch[i]=i;
}
```

Виклик функції відбудеться успішно:

Лістинг 3.42
```
int* nix;
worstInit(nix, 26); // nix так і залишився невстановленим
```

Однак спроба скористатися виділеним масивом може закінчитися діагностичним повідомленням на кшталт того, що була спроба доступу до недоступної частини пам'яті (`access violation`). Як не згадати про пораду попередньо ініціалізовувати указники!

Лістинг 3.43
```
int* nix=0;
worstInit(nix, 26); // nix так і залишився невстановленим
assert (nix!=0);
```

Це принаймні дало б змогу на етапі надлагодження виконати перевірку судження, яка відразу виявила б помилку в способі передавання параметра.

З огляду на це, безпечніше явно використовувати в сигнатурах масив, а не указник, як це було запропоновано для функцій `show` та `swapar`, тому що система програмування не дозволить застосувати до масиву будь-яку спробу переспрямувати відповідний указник – фактичний параметр.

Запропонуємо реалізації для двох інших функцій. Нехай перша з них забезпечує виведення масиву.

Лістинг 3.44
```
void show(int ar[n], size_t n) {
   cout<<"start:"<<endl;
   for(size_t i=0; i<n; i++)
      cout<<ar[i]<<' ';
   cout<<endl;
}
```

Друга ж функція переставляє елементи масиву в оберненому порядку.

Лістинг 3.45
```
void swapar(int ar[], size_t n) {
   for(size_t i=0; i<n/2; i++)
      pswap(ar+i, ar+n-1-i);
}
```

Наведемо приклад послідовного виклику цих функцій.

Лістинг 3.46
```
const size_t n=10; int ar[n];
// Масив передається як указник
init(ar, n);
// Масив передається як масив
show(ar, n); swapar(ar, n); show(ar, n);
```

і його результати:

```
start:
0 1 2 3 4 5 6 7 8 9
start:
9 8 7 6 5 4 3 2 1 0
```

Зверніть увагу на те, що масив `ar` з однаковим успіхом передано і як указник, і як масив (порівняйте, наприклад, функції `init` і `swapar`).

Вправа 3.5

Знайдіть помилку в реалізації функції `allocate`. Запропонуйте для неї тестову програму.

Лістинг 3.47

```
void allocate(char* t, size_t n, char c) {
   t=new char[n]; t[n-1]= '\0';
   for(size_t i=0; i<n-1; i++)
      t[i]=c;
   cout<<"This is inside: "<<t<<endl;
   return;
}
```

Розглянемо два приклади використання масивів як параметрів і навіть результатів. Перший із них – це копіювання масивів у припущенні, що попередньо для кожного з них було виділено пам’ять.

Лістинг 3.48

```
// Грубе копіювання масиву: немає гарантії
// визначеності указників і однаковості розмірностей обох масивів
void copyArray(double* src, double* dat, size_t sz) {
// Перевірка існування масивів
   assert((src!=0)&&(dat!=0));
// Перевірка масивів на ідентичність
   if(src==dat)
      return;
// Власне копіювання
   for(size_t ix=0; ix<sz;++ix)
      dat[ix]=src[ix];
}
```

Додатковий параметр `sz` відповідає за розмірність. Перша перевірка судження дає змогу з’ясувати, чи були указники попередньо встановлені. Однак гарантії, що у визначенні масивів їм було надано одну й ту саму розмірність, немає. Ось приклад помилки, яку важко виявити.

Лістинг 3.49

```
double m1[10], m2[5];
for(size_t ind=0; ind<10; ind++)
   m1[ind]=ind;
copyArray(m1,m2,10);
```

Переглянувши масив m1 після виконання функції, можна побачити багато несподіваного. Переконайтеся в цьому.

Є спосіб перевірити відповідність розмірностей формального та фактичного параметрів. Він полягає у використанні псевдоніма масиву (відсилки) як параметра. Оголосимо тип масиву Array і тип RefArray як відсилку до нього.

Лістинг 3.50

```
const size_t arraySize=10;
typedef double Array[arraySize]; // Визначення типу масиву
typedef Array& RefArray; // Визначення псевдоніма типу масиву
```

Підправимо функцію копіювання:

Лістинг 3.51

```
// Точніше копіювання масиву:
void copyArray(RefArray src, RefArray dat) {
   if(src==dat) return;
   for(size_t ix=0; ix<arraySize;++ix)
      dat[ix]=src[ix];
}
```

Тепер організуємо виклик:

Лістинг 3.52

```
double ar1[10], ar2[5];
for(size_t ind=0; ind<10; ind++)
   ar1[ind]=ind;
copyArray(ar1,ar2);
```

Наша спроба завершиться виведенням діагностичного повідомлення

```
error C2664: 'copyArray' : cannot convert parameter 2 from 'double [5]'
to 'double(&)[10]'
```

Однак це також не вбезпечує від виходу за межі масиву. Якщо помилково записати всередині функції copyArray вираз dat[11], то його не буде розпізнано як недопустимий.

Ситуація з масивами символів трохи простіша. Певна річ, проблема виділення достатнього місця залишається й для рядків символів, але спрощується проблема виявлення кінця масиву. Тому для копіювання символьних масивів не потрібен додатковий параметр розмірності, позаяк про закінчення масиву свідчить нульовий код. Припущення про попереднє виділення достатньої пам’яті залишається в силі.

Лістинг 3.53

```
// Копіювання рядків
void copy(char* source, char* dest) {
   while(*source!='\0') {
      *dest=*source; dest++; source++;
```

```
   }
   *dest='\0';
}
```

У лістингу 3.54 показано таке саме копіювання компактним С-текстом (його корисно прочитати й проаналізувати особливо уважно).

Лістинг 3.54

```
// Ідіома копіювання рядків
void copyCStyle(char* source, char* dest) {
   while(*dest++=*source++);
}
```

Вправа 3.6

Сформулюйте умови щодо фактичних параметрів, які мають виконуватися для успішного виконання функцій `copy` та `copyCStyle`. Складіть і виконайте тестову програму для них.

Зробимо перші висновки. У мові С++ можна задавати параметри-масиви чи як власне масиви, чи як указники. Перший спосіб безпечніший, тому що з параметром-указником можна припуститися помилки, ненавмисно переспрямувавши його в інше місце та через це втративши зв'язок із фактичним параметром, чого система програмування виявити не в змозі. Трохи пізніше побачимо, що варіантом захисту від такої помилки може бути сталий указник.

Засобом підвищення надійності обробки масивів може бути техніка так званого загортання масивів. Виготовлення «обгорток» для структур даних стало нині поширеною технікою, яка дає змогу адаптувати програмне забезпечення до заданих інтерфейсів. Розглянемо найпростіший приклад.

Нехай потрібно виконувати обчислення над векторами в *n*-вимірному просторі. Тоді варто задати одне значення константи розмірності `n` для всієї програми та «загорнути» вектори в структуру, яка міститиме два члени. Перший член – розмір масиву; він статичний, тому спільний для всіх екземплярів структури. Завдяки цьому всі вектори мають один і той самий розмір. Другий член структури – власне указник на вектор дійсних чисел:

Лістинг 3.55

```
// Структура загорнутих векторів
struct WrappedVector {
  static const size_t _n; // Статична розмірність, одна для всіх об'єктів
  double* _x; // Власне вектор
};
```

Тепер у файлі реалізації потрібно помістити визначення розмірності, наприклад таке:

```
// Визначення статичного поля
const size_t WrappedVector::_n=100;
```

Лістинг 3.55а

Оскільки розмірність – це статичний член структури, її буде ініціалізовано до початку виконання програми без попереднього створення жодного об'єкта – екземпляра структури. Зверніть увагу й на те, що банально названі `_n` та `_x` не конфліктуватимуть з іншими можливими одноіменними об'єктами з області видимості програми, оскільки їх локалізовано в просторі імен структури `WrappedVector`.

Одразу звернімо увагу на слабкі місця цієї структури. По-перше, потрібно домовитися про те, хто відповідає за визначення розміру масиву. Ініціалізацію атрибуту `_n` не вдасться заховати у файлі реалізації структури, бо тоді вона стає невидимою для місця використання, яке зазвичай буває в іншому, ніж реалізація структури, файлі. Найпридатнішим для такої ініціалізації видається файл клієнтської програми. Тоді відповідальність за ініціалізацію розміру буде покладено на клієнта цієї структури.

Інша ще серйозніша слабина виявляється під час створення екземпляра структури. Оскільки розмір масиву попередньо ініціалізовано, не має виникати ускладнень у виділенні пам'яті під сам масив. Однак механізмами реалізації структур передбачено лише виділення пам'яті під кожен атрибут, тобто в нашій ситуації під неініціалізований указник масиву. При цьому автоматичного не виділяється місце під сам масив, тому матимемо справу з некоректно встановленим (власне, засміченим) указником. Тому визначення загорнутого вектора, можливо, надалі ще не раз доведеться переписувати, поки не буде знайдено прийнятне рішення.

Запропонуємо тепер інший варіант функції обчислення скалярного добутку двох векторів.

```
// Обчислення скалярного добутку: дуже погано
double prod(WrappedVector u, WrappedVector v) {
   double s=0;
   for(int i=0; i<u.n; i++)
      s+=u._x[i]*v._x[i];
   return s;
}
```

Лістинг 3.56

Якщо не вжити додаткових заходів, то параметри – загорнуті масиви – передаватимуться значенням, тобто буде скопійовано структури, передані фактичними параметрами. На тому рівні обробки структур даних, на якому ми поки що перебуваємо, буде виконане так зване поверхневе копіювання. Виклик `prod(a, b)` призведе до ініціалізацій

```
WrappedVector u=a; WrappedVector v=b;
```

Лістинг 3.57

виконаних покомпонентно:

Лістинг 3.58

```
u._x=a._x; v._x=b._x;
```

Зауважимо, що це копіювання не масивів, а лише їхніх адрес. Тому на час виконання функції кожен із масивів `a._x` і `b._x` адресуватиметься двома указниками. Як ми побачимо далі, таке спільне використання пам'яті спричиняє проблему відповідальності за видалення спільно використаного масиву.

У розглянутому прикладі масиви були параметрами, які «забезпечували» функцію своїми значеннями. Тепер нехай параметр буде результатом для функції. Візьмемо, наприклад, функцію створення самого вектора виділення пам'яті під масив і наповнення його значеннями. Передавати параметри значенням уже не можна з тих самих причин, що й у разі обміну значеннями змінних (див. лістинг 3.28). Щоб операції з формальним параметром впливали на фактичний параметр, скористаємося параметром-відсилкою. Для ініціалізації створеного масиву заповнимо його значеннями, наданими генератором випадкових чисел.

Лістинг 3.59

```
// Створення масиву та його наповнення випадковими
// значеннями. Параметр-результат передається відсилкою
void construct(WrappedVector& u) {
   u._x=new double[u._n];
   for(int i=0; i<u._n; i++)
      u._x[i]=rand();
   return;
}
```

Згідно з правилом гарного тону створюємо функцію для видалення масиву:

Лістинг 3.60

```
// Видалення масиву
// Параметр-результат передається відсилкою
void destroy(WrappedVector& u) {
   delete [] u._x; u._x=0;
   return;
}
```

Тепер клієнту загорнутого вектора надаватиметься не лише визначення структури, а й додаткові функції її обслуговування – утиліти. Заголовний файл міг би набути такого вигляду:

Лістинг 3.61

```
// WrappedVector.h
// Структура загорнутих векторів
struct WrappedVector {
```

```
// Статична розмірність, одна для всіх
   static const size_t _n;
// Власне вектор
   double* _x;
};
// Створення масиву та його наповнення випадковими значеннями
// Параметр-результат передається відсилкою
void construct(WrappedVector& u);
// Видалення масиву. Параметр-результат передається відсилкою
void destroy(WrappedVector& u);
```

Запропонувавши функції створення та видалення масивів, ми вже зробили важливий крок у напрямку дисциплінованого програмування. Однак наявність у проекті функції прибирання сміття ще не забезпечує її своєчасного виклику. Як же зробити так, щоб згадане правило гарного тону було дотримане завжди? Як гарантувати, що кожен створений масив буде своєчасно видалений? Про такі гарантії йтиметься трохи пізніше, під час вивчення справжніх функцій створення та видалення, які мають назви конструкторів і деструкторів.

Широко використовують також загорнуті рядки символів, але про них ітиметься в наступному розділі.

3.2.5. Особливості передавання сталих параметрів

Знову повернімося до прикладу обчислення найбільшого спільного дільника.

Лістинг 3.62

```
// Найбільший спільний дільник
int gcd(int m, int n) {
   while(m!=n)
      if(m>n) m=m-n;
      else n=n-m;
   assert(m==n);
   return m;
}
```

На етапі налагодження цієї функції не зайве було б перевірити її коректність і переконатися в тому, що знайдене в підсумку число – справді шуканий найбільший спільний дільник. Перевірку варто вбудувати до тіла функції, скориставшись для цього судженням `assert`. Щоб це зробити, спробуємо модифікувати відповідний оператор лістингу 3.62. Однак відразу ж виявимо, що для наведеного раніше коду це зробити неможливо, оскільки значення параметрів, які було надано на вході до функції,

утрачено. Створимо в тілі функції додаткові змінні та скопіюємо в них значення параметрів.

```
// Найбільший спільний дільник із додатковими змінними      Лістинг 3.63
int gcd(int mpar, int npar) {
// Збереження значень параметрів
   int m=mpar, n=npar;
   while(m!=n)
      if(m>n) m=m-n;
      else n=n-m;
// Умова подільності
   assert((m==n)&&(mpar%m == 0)&&(npar%n == 0));
   return m;
}
```

Тепер з'явилися змінні, призначені для зберігання значень параметрів `mpar` і `npar`, і ними можна скористатися, складаючи судження. Тіло функції невелике, і неважко переконатися в тому, що значення параметрів не змінюються в процесі обчислення. А втім, як було вже сказано, існує прийом, який гарантує незмінність значень. Він полягає в присвоєнні об'єктам програми статусу сталих величин.

```
// Найбільший спільний дільник зі сталими параметрами      Лістинг 3.64
int gcd(const int mpar, const int npar) {
// Копіювання значень параметрів
   int m=mpar, n=npar;
   while(m!=n)
      if(m>n) m=m-n;
      else n=n-m;
// Умова подільності. Початкові значення не втрачено
   assert((m==n)&&(mpar%m == 0)&&(npar%n == 0));
   return m;
}
```

Приходимо до простого технологічно виправданого рішення: захистити параметр кваліфікатором константи **`const`**, вираженого сигнатурою

```
int gcd(const int mpar, const int npar);
```

Тепер не тільки не змінюватиметься значення фактичного параметра, що гарантовано способом передавання параметра значенням, але й значення формального параметра стає захищеним від змін. У наших звичайних позначеннях цей вид функцій можна записати так:

```
AD f(const AT cx);          (CAT→AD)
```

У разі виклику `f(a)` визначення формального параметра набуває вигляду

```
const AT cx=a;
```

Згадаймо невдалу функцію обміну значеннями параметрів, яку доречніше було б називати `noswap`.

```
// Функція, яка не обмінює значення параметрів                Лістинг 3.65
void noswap(double x, double y) {
   double z=x; x=y; y=z;
   return;
}
```

Її текст, на перший погляд цілком правильний (якщо не брати до уваги семантики передавання параметрів), одразу перетворюється на синтаксично помилковий, варто лише додати до заголовка специфікацію сталого параметра.

```
// Функція, яку не пропустить компілятор                      Лістинг 3.66
void noswap(const double x, const double y) {
   double z=x;
   x=y;   // Помилка: спроба змінити сталу
   y=z;   // Помилка: спроба змінити сталу
   return;
}
```

Сталі формальні параметри дають змогу повністю розв'язати проблему співвідношення формальних і фактичних параметрів, переданих значенням. Справді, якщо передавання параметра значенням гарантує незмінність фактичного параметра на час виконання функції, то так само логічно було б заборонити змінювати формальний параметр. Завдяки цьому зберігатиметься повна тотожність значень формального та відповідного йому фактичного параметра. Тому параметри, передані значеннями, варто робити сталими, залишаючи право модифікації за параметрами, переданими відсилкою, що знову ж зумовлюватиме повну тотожність відповідних формального та фактичного параметрів. Прагнення змінювати параметр, переданий значенням, можна вважати способом зекономити комірку пам'яті для допоміжної змінної, сумістивши її з формальним параметром. Справді, у лістингу 3.62 маємо прецедент використання обох змінних `m` і `n` водночас за двома призначеннями: спочатку як формальних параметрів, а далі як допоміжних змінних у розрахунках.

Висновок. *Використання однієї змінної за кількома призначеннями може виявитися джерелом серйозних помилок у програмуванні. Варто дотримувати*

принципу «кожній змінній одне призначення», а тому параметри, передані значеннями, визначати як сталі, використовуючи модифікатор **`const`**.

Вправа 3.7

Доповніть функцію обчислення найбільшого спільного дільника викликом функції перевірки правильності обчислень. Для цього запрограмуйте функцію `gcdChecker`

```
bool gcdChecker(const int m, const int n, const int candidate);
```

яка перевірить, чи є число, задане за допомогою параметра `candidate`, найбільшим спільним дільником чисел, заданих параметрами `m` і `n`. Помістіть її виклик у перевірку судження.

Лістинг 3.67

```
// Найбільший спільний дільник із перевіркою результату
int gcd(const int mpar, const int npar) {
   int m=mpar, n=npar;
   while(m!=n)
      if(m>n) m=m-n;
      else n=n-m;
// Повна перевірка
   assert((m==n)&&gcdChecker(mpar, npar, m));
   return m;
}
```

Природно виникає запитання: що зміниться, якщо з визначень параметрів забрати **`const`**? Відповідь на нього така сама проста, як і на запитання з правил дорожнього руху: що станеться, якщо переходити вулицю на червоний сигнал? Навіть за умови дотримання правил іншими учасниками руху зникне гарантія безпечності переходу.

Проаналізуємо ще один приклад – функцію піднесення дійсного числа до натурального степеня.

Лістинг 3.68

```
// Повільне, хоча правильне піднесення до степеня
// Насправді так ніхто не підносить
double slowPower(const double x, const int n) {
   double y=1;
   for(int i=0; i<n; i++)
      y=y*x;
   return y;
}
```

Тут сталі параметри – швидше данина принципу, ніж потреба, оскільки в тілі функції немає спроб змінити їх. Згадайте, скільки разів ви переходили вулицю на червоне світло, переконавшись, що поблизу немає

машин. Але ж ми щойно домовилися програмувати надійно!

Згадаймо тепер алгоритм швидкого піднесення до степеня з попереднього розділу.

Лістинг 3.69

```
double y=1;
while(n>0)
   if(n%2==0) {n/=2; x*=x;}
   else {n--; y*=x;}
```

Він має параметри `x` і `n`, але функція змінює їх. Тому тут візьмемо за параметри інші змінні, скажімо `base` і `exponent`:

Лістинг 3.70

```
// Швидке піднесення до степеня
double power(const double base, const int exponent) {
   double x=base; int n=exponent; double y=1;
   while(n>0) {
      if(n%2==0) {x=x*x; n=n/2;}
      else {y=y*x; n=n-1;}
   }
   return y;
}
```

Завдяки незмінності параметрів тепер для налагодження можна виписати судження про правильність швидкого алгоритму обчислення степеня, використовуючи повільний алгоритм:

Лістинг 3.71

```
// Швидке піднесення до степеня з перевіркою суджень
double power(const double base, const int exponent) {
   double x=base; int n=exponent; double y=1;
// Очевидне судження
   assert((y==1)&&(y*slowPower(x,n)==slowPower(base,exponent));
   while(n>0) {
      if(n%2==0) {x=x*x; n=n/2;}
      else {y=y*x; n=n-1;}
// Індуктивне припущення: інваріант циклу
      assert(y*slowPower(x,n)==slowPower(base,exponent));
   }
// Наслідок умови закінчення й інваріанта циклу
   assert((n==0) && (y==slowPower(base,exponent)));
   return y;
}
```

Запрограмована таким способом функція з лістингу 3.71 стає надто повільною, адже навіть оптимізувавши її, як у вправі 3.8, ми одержимо кільканадцять викликів повільного піднесення до степеня. Однак усі вони

перебувають у межах команди перевірки суджень, яка потрапляє до об'єктного коду лише в результаті компіляції на етапі налагодження. Нагадаємо, що можна вимкнути режим налагодження, а також виклики повільного піднесення до степеня, записавши команду визначення ідентифікатора для препроцесора

```
#define NDEBUG
```

і розмістивши її перед командою

```
#include <cassert>
```

але не після неї. У цьому разі, щоб відмовитися від налагодження, потрібно замінити команду визначення ідентифікатора командою відмови від нього

```
#undef NDEBUG
```

Тепер остаточна версія функції не перевірятиме своєї правильності, а тому не міститиме викликів повільного піднесення до степеня.

Вправа 3.8

Доповніть функцію з лістингу 3.71 командами препроцесора й оптимізуйте її, винісши за межі циклу обчислення, незалежні від його параметрів. Проаналізуйте судження та протестуйте функцію піднесення до степеня.

Так само варто змінити на сталий тип параметра у визначенні функції обчислення відстані до точки:

Лістинг 3.72

```
// Параметр передано сталим значенням:
// надійно, але витратно
double mod(const Point u) {
   return sqrt(u._x*u._x+u._y*u._y);
}
```

Принципова відмінність цього прикладу від попередніх полягає в типі параметра. Якщо там фігурували арифметичні типи, то тепер мова йтиме про структуру даних, яка відрізняється від об'єкта фундаментального типу передусім розмірами: той не більший ніж два слова, а розміри структур довільні та не прогнозовані, оскільки можуть бути визначені на етапі виконання. Друга відмінність полягає в механізмах копіювання. Для фундаментальних типів даних маємо стандартне копіювання байта, півслова, слова чи подвійного слова, а структури даних можуть потребувати складного глибокого копіювання, яке програмують додатково.

Із цієї причини передавання структур даних як параметрів традиційно виконують за допомогою указників, якщо йдеться про мову C, чи відсилок у разі застосування новіших методів мови C++. Так, функцію з попе-

реднього прикладу можна переписати так:

```
// Параметр передано за допомогою відсилки:          Лістинг 3.73
// ефективно, але ненадійно
double mod(Point& u) {
   return sqrt(u._x*u._x+u._y*u._y);
}
```

Ненадійність передавання за допомогою відсилки спричинена наданням надто широкого доступу до фактичного параметра, унаслідок чого може бути змінене його значення. Знову виникає питання про відповідальність і гарантії. Адекватне рішення полягає в застосуванні обмеженого читанням доступу, тобто у використанні сталої відсилки:

```
// Параметр передано за допомогою сталої відсилки:   Лістинг 3.74
// ефективно та надійно
double mod(const Point& u) {
   return sqrt(u._x*u._x+u._y*u._y);
}
```

У наших спрощених позначеннях зафіксуємо цей тип сигнатури так:

```
AD f(const T& crx);          (CT&→AD)
```

У разі виклику `f(u)` визначення формального параметра набуває вигляду

```
const T& crx=u;
```

Формальний параметр стає альтернативним іменем фактичного, але права доступу до нього обмежені лише читанням. Як побачимо пізніше, схема передавання параметрів за допомогою сталих відсилок (`CT&→AD`) стає найважливішою та найпоширенішою в об'єктно-орієнтованому програмуванні.

Принцип незмінності. *У програмуванні варто дотримувати принципу незмінності всього, що може залишатися незмінним. Скрізь, де це можливо, слід визначати об'єкти як сталі.*

Згадаймо ще раз три функції з лістингів 3.39, 3.44, 3.45.

```
void init(int* ar, size_t n);                         Лістинг 3.75
void show(int ar[],size_t n); void swapar(int [],size_t);
```

У функції `init` використано параметр-указник, але мається на увазі масив. Це поширений прийом, зумовлений традиційним для мови C заміщенням масивів указниками. Утім, як було продемонстровано в під-

розділі 3.2.4, це не зовсім безпечно, позаяк указник можна випадково переспрямувати, розірвавши зв'язок фактичного параметра з формальним. Можна поліпшити ситуацію, зробивши указник сталим. Справді, типова змістова помилка, спричинена функцією невдалої ініціалізації `worstInit`, тепер стає синтаксичною, і її виявить компілятор.

Лістинг 3.76

```
// Ініціалізація смітника
void worstInit(int* const ar, size_t n) {
// Компілятор не пропустить наступний рядок
   ar=new int[n];
   for(size_t i=0; i<n; i++)
      ar[i]=i;
}
```

Сталий указник не можна переспрямувати, і зв'язок формального параметра з фактичним зберігатиметься впродовж усього часу виконання функції

```
void init(int* const ar, size_t n);
```

Те саме варто передбачити для функції `swapar` переставляння елементів масиву

```
void swapar(int* const, size_t);
```

Функція виведення `show` має параметром масив

```
void show(const int ar[],size_t n);
```

Тепер спроба зіпсувати масив стає синтаксичною помилкою.

Лістинг 3.77

```
// Синтаксична помилка під час спроби зіпсувати масив
void show(const int ar[n], size_t n) {
   cout<<"start:"<<endl;
   for(size_t i=0; i<n; i++)
// Спроба зіпсувати масив: значення елементів сталого масиву
// змінювати не можна
      cout<<ar[i]++<<' ';
   cout<<endl;
}
```

Якщо ж використовувати не масив, а указник, то повністю виключити переспрямування та гарантувати незмінність наповнення масиву допоможе тип сталого указника на сталу

```
void show(const int* const ar, size_t n);
```

Зауважимо, що третя можливість – несталий указник сталого значення, наприклад, `const int* ar` – майже не має сенсу. Справді, значення, на які він указує, не можна змінювати, але сам указник можна переспрямувати, розірвавши його зв'язок із фактичним за допомогою параметра.

Висновок. *Проаналізувавши надійність зв'язку формальних і фактичних параметрів, доходимо висновку, що передаванню параметра за допомогою указника – найпоширенішому механізму програмування мовою С – теж властиві проблеми. У разі передавання параметра значенням може скластись ілюзія, що зміна значення формального параметра впливає на фактичний. Якщо ж параметр передається за допомогою указника, то зв'язок між формальним і фактичним за допомогою параметра може розірватися, хоча здається, що він незмінний. Тому варто використовувати сталі указники, щоб убезпечитися від переспрямування, і сталі указники сталих (сталі масиви) для захисту вмісту масиву від змін.*

Вправа 3.9

Знайдіть помилку в наведеній версії переставляння значень, на які вказують два указники.

Лістинг 3.78

```
// Невдала спроба переставляння
void pswap(double* x, double* y) {
   double z=*x; x=y; *y=z;
   return;
}
```

Припущена в лістингу 3.78 помилка стає неможливою, якщо застосувати сигнатуру з використанням сталих указників:

Лістинг 3.79

```
// Справжній обмін значеннями pswap засобами
// сталих указників C++
void pswap(double* const x, double* const y) {
   double z=*x; *x=*y; *y=z;
   return;
}
```

Щоправда, розглянуті способи уточнення типів параметрів не усувають проблеми відповідності розмірів формальних і фактичних параметрів, яку певною мірою розв'язувало використання псевдонімів масивів. Застосування псевдонімів сталих розберіть самостійно, розглянувши наступний приклад.

Сталість параметрів дає змогу чіткіше показати їх призначення. Тип цільового масиву `RefArray`, як і раніше, залишиться відсилкою, а тип джерела копіювання `ConstRefArray` визначимо як сталу відсилку.

```
const size_t arraySize=10;                                   Лістинг 3.80
typedef double Array[arraySize];     // Визначення типу масиву
typedef Array& RefArray;      // Визначення псевдоніма для типу масиву
typedef const Array& ConstRefArray;  // Визначення псевдоніма
                                     // для типу сталого масиву
```

Тепер не виникатиме запитання, звідки й куди виконується копіювання: звичайно, зі сталої відсилки до несталої:

```
// Ще точніше копіювання масиву:                             Лістинг 3.81
void copyArray(ConstRefArray src, RefArray dat) {
   if(src==dat) return;
   for(size_t ix=0; ix<arraySize;++ix)
      dat[ix]=src[ix];
}
int main() {
   Array ar1[10], ar2[5];
   for(size_t ind=0; ind< arraySize; ind++)
      ar1[ind]=ind;
   copyArray(ar1,ar2);
   return 0;
}
```

Масив `src` не зможе з'явитися зліва від присвоєння, що захищає від помилок у процесі програмування, а сигнатура функції точно вказує на призначення параметрів. Це захистить від її некоректного вживання:

```
void copyArray(ConstRefArray, RefArray);
```

Сталий масив типу **`ConstRefArray`** копіюється до масиву типу **`RefArray`**.

Вправа 3.10

Видаліть параметр розмірності з функції виведення масиву `show` та запропонуйте її реалізацію без нього.

Зведемо вивчені типи параметрів у табл. 3.1.

Висновки. *Для розробки надійних програм варто використовувати параметри певних типів, на жаль, недоступних у традиційному програмуванні мовою С. Ще раз наголосимо, що з погляду розробника параметр, переданий за допомогою значення як* ***`const`*** *`T`, найнадійніший, бо не створює непорозумінь; параметр* ***`const`*** *`T&` найефективніший, оскільки не потребує витрат на копіювання (співвідношення між ними буде розглянуто в наступному розділі). Параметри-указники варто робити сталими `T*` **`const`**.*

Таблиця 3.1. Типи параметрів

Тип формального параметра	Зв'язок формального та фактичного параметрів	Властивості формального параметра	Зауваження
Параметри-аргументи: немає впливу на фактичний параметр			
`T` (арифметичні типи, структури, класи)	Копіювання	Змінний формальний параметр	Не відрізняються з погляду користувача
`const T`	Копіювання	Незмінний формальний параметр	
`const T&`	Стала відсилка без копіювання	Незмінний формальний параметр	
`const T* par` `const T[] par`	Указник константи без копіювання	Змінний формальний параметр-указник `lvalue(*par)`, але незмінне значення другого рівня `rvalue(*par)`	Невдале рішення (можливе переспрямування)
`const T* const par`	Сталий указник константи без копіювання	Незмінний формальний параметр-указник, незмінне значення другого рівня (елементів масиву) `rvalue(*par)`	Використовують для перегляду масиву без зміни його елементів
Параметри-результати, є доступ до фактичного параметра			
`T&`	Відсилка без копіювання	Повний доступ до фактичного параметра	Найнадійніший параметр-результат
`T* par` `T[] par`	Повний, але непрямий доступ до фактичного параметра; передавання масивів (`T[]`)	Незмінний фактичний параметр-указник, змінні значення формального параметра першого `lvalue(par)` і другого рівня `lvalue(*par)`, `rvalue(*par)`	Невдалий параметр-результат у стилі мови C (можливе переспрямування)
`T* const par`	Сталий указник без копіювання	Незмінний формальний параметр-указник, змінні значення другого рівня `rvalue(*par)`	Захист від переспрямування, повний доступ до елементів масиву

3.2.6. Параметри-указники другого рівня

Як було вже з'ясовано в попередньому розділі, звичайних указників виявляється недостатньо для того, щоб створювати у функціях нові елементи пам'яті й повертати їх через фактичні параметри. Розглянемо тепер функції, параметри яких – указники другого рівня (назвемо їх *подвійними*, *pointer to pointer*). Умовно позначатимемо функцію з параметром такого вигляду як

```
void f(T** ppx);          (T**→void)
```

У разі виклику `f(ppe)` відбудеться ініціалізація

```
T** ppx=ppe;
```

унаслідок якої правобічне значення `ppe` потрапить на місце, задане лівобічним значенням, а тому збігатимуться результати першого

```
rvalue(*ppx) == rvalue(*ppe);
```

та другого

```
rvalue(**ppx) == rvalue(**ppe);
```

розіменування.

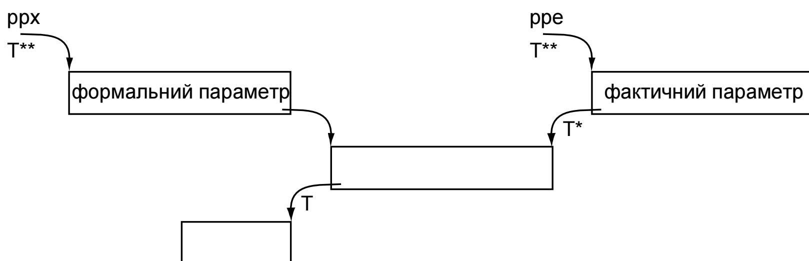


Рис. 3.7. Відповідність між фактичними і формальними параметрами-указниками другого рівня

Пояснимо це на прикладі. Розглянемо функцію створення масиву заданого розміру `n` і заповнення його заданим символом `c`.

Лістинг 3.82

```
// Функція виділення пам'яті й ініціалізації масиву,
// переданого указником на указник
void allocate(char** t, int n, char c) {
   *t=new char[n]; (*t)[n-1]='\0';
   for(int i=0; i<n-1; i++)
      (*t)[i]=c;
```

```
   cout<<*t<<endl;
   return;
}
```

Її виклик матиме, наприклад, такий вигляд:

```
char* a =0; allocate(&a, 10, 'a');
```

Лістинг 3.83

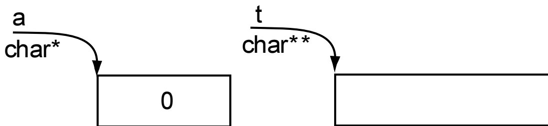


Рис. 3.8. Початковий стан указників

На рисунку покажемо стан указників до початку виконання функції. Її виконання можна зобразити як послідовність таких операцій:

Лістинг 3.84

```
// Ініціалізація параметрів
   char** t=&a;
   int n=10;
   char c='a';
// Виділення пам'яті: rvalue(*t)== rvalue(&a) == lvalue(a) == a:
   *t=new char[n];    // a=new char[n];
// Позначення кінця рядка
   (*t)[n-1]='\0';    // a[n-1] ='\0';
   for(int i=0;i<n-1;i++)
      (*t)[i]=c;      // a[i]=c;
   cout<<*t<<endl;    // cout<<a<<endl;
```

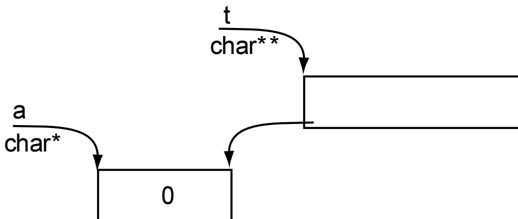


Рис. 3.9. Ініціалізація формального параметра

Наведений спосіб передавання параметра більш властивий програмуванню в стилі мови C. Він небезпечний з точки зору втрат пам'яті, тому що оператор `new` перебуває всередині функції і відповідальність за правило гарного тону розподіляється між функцією та її клієнтом.

Якщо згадати про використання сталих указників, то сигнатура стає точнішою

```
void f(T** const ppx);          (T** const→void)
```

Це захищає указник другого рівня від спроби переспрямування на інше місце, спричиненої, наприклад, помилкою в тексті програми на зразок

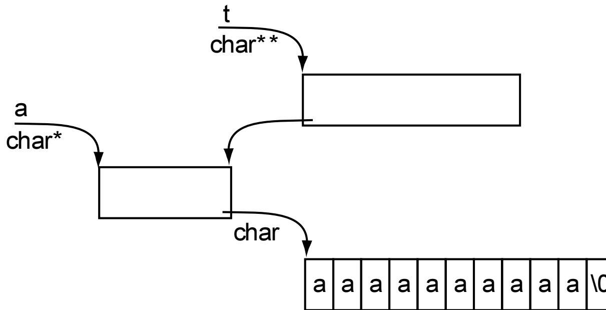


Рис. 3.10. Зміна значення фактичного параметра через указник другого рівня

`t=new char*[n]`

замість

`*t=new char[n];`

і, відповідно, від розриву зв'язку між фактичним і формальним параметром, оскільки для функції вигляду `(T**→void)` переспрямування параметра `t` синтаксично коректне, а для `(T** const→void)` – ні.

Вправа 3.11

Перепишіть та перевірте функцію `allocate` із використанням схеми `(T** const→void)`.

Указники на указники – це данина традиції програмування мовою С, яка полягає у використанні указників для зв'язку з параметрами – результатами функцій. За ідеологією мови С++ адекватнішим буде передавання указника як псевдоніма за схемою

```
void f(T*& rpx);          (T*&→void)
```

Тоді зникають зайві розіменування. Перепишемо визначення функції з попереднього прикладу за новою схемою:

Лістинг 3.85

```
// Функція виділення пам'яті й ініціалізації масиву
// Основний параметр передано за допомогою відсилки на указник
void allocate(char*& t, int n, char c) {
   t=new char[n]; t[n-1]='\0';
   for(int i=0; i<n-1; i++)
      t[i]=c;
   cout<<t<<endl;
   return;
}
```

Виклик стає природнішим:

```
char* a;
allocate(a, 10, 'a');
```

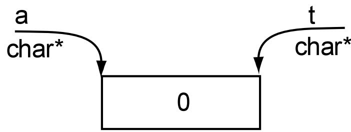


Рис. 3.11. Відповідність між фактичним і формальним параметром-відсилкою до указника

Однак із цим, можливо, не погодяться прихильники мови С, які звикли завжди розуміти фактичний параметр як значення, коли немає операції адресування.

Із погляду розробника другий варіант знову ж таки надійніший, оскільки в першому можна припуститися помилки, пропустивши один рівень розіменування. Але втрати пам'яті все одно можливі.

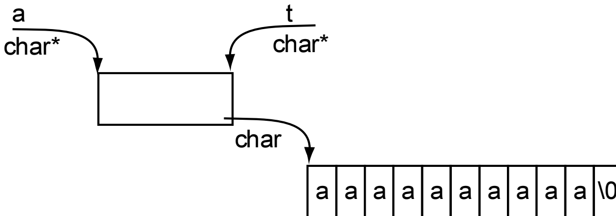


Рис. 3.12. Зміна значення фактичного параметра через відсилку до указника

Повернімося тепер до особливостей нульового параметра-указника. Для прикладу візьмемо бінарне дерево з вузлами такої структури:

Лістинг 3.86

```
// Вузол бінарного дерева
struct Tree  {
   int node; Tree* left; Tree* right;
};
```

Указники `left` і `right` указують на піддерева нетермінального вузла, а термінальні вузли (листки дерева) мають нульові указники `left` і `right`. У лістингу 3.87 наведено функцію створення вершини дерева, де параметр `aTree` відіграє роль параметра-результату.

Лістинг 3.87

```
// Функція створення одного вузла дерева з двох
// піддерев left і right та запису у вузол розмітки node
void createTree(Tree** aTree, int node, Tree* left,  Tree* right) {
   *aTree=new Tree; (*aTree)->node=node;
   (*aTree)->left=left; (*aTree)->right=right;
   return;
}
```

У разі створення термінального вузла дерева, у якого немає власних піддерев, два останні параметри мають бути нульовими. Передавання невизначеного (тобто нульового) указника схоже на необов'язковий параметр, про який уже йшлося раніше (див. висновки до розділу 3.2.3). Ось

приклад використання функції `createTree`, у якому буде послідовно створено дерево з трьох вершин (рис. 3.13).

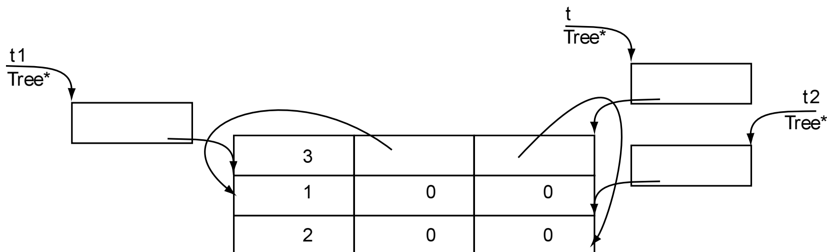


Рис. 3.13. Створення дерева з трьома вузлами

Лістинг 3.88

```
int main() {
   Tree* t=0, *t1=0, *t2=0;   // Указники для створення вершин дерева
   createTree(&t1, 1, 0, 0);  // Створення першої вершини дерева
   createTree(&t2, 2, 0, 0);  // Створення другої вершини дерева
   createTree(&t, 3, t1, t2); // Створення кореня дерева
   return 0;
}
```

Якщо ж скористатися параметрами із заданими за замовчуванням значеннями, то ідіома необов'язкового параметра в разі виклику функції стає ще прозорішою. Ось відповідна сигнатура

```
void createTree(Tree** aTree, int node,
      Tree* left =0, Tree* right=0);
```

і виклики

```
int main() {
   Tree* t=0, *t1=0, *t2=0;   // Указники для створення вершин дерева
   createTree(&t1, 1);        // Створення першої вершини дерева
   createTree(&t2, 2);        // Створення другої вершини дерева
   createTree(&t, 3, t1, t2); // Створення кореня дерева
   return 0;
}
```

У тестовій програмі для створення дерева є серйозна вада. Указник `t` вказує на корінь дерева, а два інші указники `t1` і `t2` – на два його піддерева. На перший погляд проблеми не видно, але поставимо собі запитання: що станеться зі значеннями цих двох указників після видалення дерева за допомогою указника `t`? Функція видалення, як буде видно з наведеного далі прикладу, «не знає» про указники, установлені всередині дерева.

Функцію видалення дерева найпростіше реалізувати рекурсивно:

Лістинг 3.89

```
// Рекурсивна функція видалення дерева
// з параметром-указником другого рівня
void destroyTree(Tree** aTree) {
   Tree* subTree;
   if((*aTree)==0)return;
   subTree=(*aTree)->left;
   destroyTree(&subTree);   // Спочатку видаляємо ліве піддерево
   subTree=(*aTree)->right;
   destroyTree(&subTree);   // Далі видаляємо праве піддерево
   delete* aTree;   // На завершення видаляємо корінь дерева
   *aTree=0;   // Онуляємо указник кореня
   return;
}
```

Звернімо увагу на особливість використання указника другого рівня в сигнатурі функції видалення дерева:

```
// Рекурсивна функція видалення дерева
// з параметром-указником другого рівня
void destroyTree(Tree** aTree);
```

У разі застосування указника першого рівня

```
// Помилкова сигнатура функції видалення дерева
void badDestroyTree(Tree* aTree);
```

видалення дерева залишилося б можливим, але не можна було б повернути назад обнулений параметр. Тоді кожен виклик доводилося б супроводжувати додатковим присвоєнням нуля указнику

```
badDestroyTree (theTree);
theTree = 0;
```

Нейтралізація указника дерева після його видалення – правильне рішення, однак залишається проблема завислих указників піддерев, якщо їх було розставлено в дереві під час чи після його створення; для її розв'язання все ще потрібні додаткові зусилля. У лістингу 3.90 наведено тестову програму для створення та подальшого видалення дерева.

Лістинг 3.90

```
int main() {
   Tree* t=0, *t1=0, *t2=0;   // Указники для створення вершин дерева
   createTree(&t1, 1);        // Створення першої вершини дерева
   createTree(&t2, 2);        // Створення другої вершини дерева
   createTree(&t, 3, t1, t2); // Створення кореня дерева
   destroyTree(&t)
```

```
// t == 0   OK;  t1, t2 зависли
   return 0;
}
```

Удале вирішення цієї проблеми запропонувати важко. Можна покласти відповідальність на клієнта функції створення вузла дерева, вимагаючи від нього нейтралізації зовнішніх указників на піддерева після створення кожного нового вузла, як це було в прикладі невдалого видалення.

Лістинг 3.91

```
// Створення вузла дерева з одночасною нейтралізацією
// указників його піддерев
createTree(&t, 3, t1, t2); t1=t2=0;
```

Відповідальність може взяти на себе розробник. Тоді рівень вкладеності під час адресування параметрів за допомогою функції створення нового вузла дерева має зрости, а клієнт функції має усвідомлювати, що вона руйнує параметри.

Лістинг 3.92

```
// Функція створення одного вузла дерева з
// нейтралізацією указників на піддерева left і right
void createTree(Tree** aTree, int node, Tree** left, Tree** right) {
   *aTree=new Tree;
   (*aTree)->node=node; (*aTree)->left=*left; (*aTree)->right=*right;
   *left=*right=0; // Обнулення указників на піддерева
   return;
}
```

Відповідно зміниться код виклику:

```
createTree(&t, 3, &t1, &t2);
```

Можна обійтися без операції адресування у виклику, якщо замість указників другого рівня використовувати відсилки до указників. Текст залишиться майже без змін, лише буде знижено рівень розіменування:

Лістинг 3.93

```
// Функція створення одного вузла дерева за відсилками
// до указників із нейтралізацією указників на піддерева left і right
void createTree(Tree*& aTree, int node, Tree*& left, Tree*& right) {
   aTree=new Tree;
   (aTree)->node=node; (aTree)->left=left; (aTree)->right=right;
   left=right=0; // Обнулення указників на піддерева
   return;
}
```

Відповідно зміниться виклик:

```
createTree(t, 3, t1, t2);
```

Вправа 3.12

Протестуйте всі три варіанти функції створення вершини дерева.

Так само можна змінити функцію видалення дерева з лістингу 3.89, застосувавши відсилку до указника.

Лістинг 3.94

```
// Рекурсивна функція видалення дерева з параметром
// за допомогою відсилки до указника
void destroyTree(Tree*& aTree) {
   Tree* subTree;
   if((aTree)==0) return;
   subTree=(aTree)->left; destroyTree(subTree);
   subTree=(aTree)->right; destroyTree(subTree);
   delete aTree; aTree=0;
   return;
}
```

Вправа 3.13

Поясніть, чому не можна записати сигнатуру створення вершини дерева так:

```
void createTree(Tree*& aTree, int node, Tree& left, Tree& right);
```

Пам'ятаючи про принцип незмінності, подивімося, де можна визначити параметр як сталий. Як видно з вищенаведеного, сталому формальному параметру «байдуже», чи сталий фактичний параметр. Це випливає зі зробленого в попередньому розділі зауваження про те, що до змінних величин можна застосовувати сталі відсилки та указники. Як пам'ятаємо, ішлося просто про обмеження прав доступу через сталий указник або альтернативне ім'я сталої. Тому щоразу, коли значення формального параметра має залишатися незмінним, варто надавати перевагу сталому параметру, щоб гарантувати незмінність і явно наголосити на відповідності цієї незмінності нашим намірам.

Однак для указників другого рівня ситуація інша. Візьмемо порожню заготовку функції :

Лістинг 3.95

```
// Параметр-указник указника сталої
void g(const char** s) {
   return;
}
```

і спробуємо підставити несталий фактичний параметр:

Лістинг 3.96

```
// Указник указника не може бути фактичним параметром
// для указника на указник сталої
```

```
char** s;
g(s);   // Помилкова команда
```

Одержимо повідомлення про помилку

```
error C2664: 'g' : cannot convert parameter 1 from 'char** ' to 'const char** '
```

У чому річ? Діагностика виявляється доволі несподіваною, якщо згадати попереднє безпроблемне вживання сталих указників. Виявляється, подвійний указник на сталу ще небезпечніший, ніж звичайний несталий указник, бо здається сталим, хоча містить проміжний несталий рівень. Пояснимо це.

Спочатку пригадаємо ситуацію зі звичайним указником. Розглянемо коректну ініціалізацію указника на символ і указника на сталий символ.

Лістинг 3.97

```
// Параметр-указник сталої
void h(const char* s) {
   return;
}
// Указник може бути фактичним параметром для указника сталої
char* s; h(s);
// Так само, як допустимі ініціалізації:
char* pchar=new char('a'); const char* cpchar=pchar;
```

Немає сумнівів щодо результату виведення значень обох указників –

```
cout<<cpchar<<':'<<pchar<<endl;
```

Це буде літера a. Законним буде й присвоєння, яке заміщує символ a символом b:

```
*pchar='b';
```

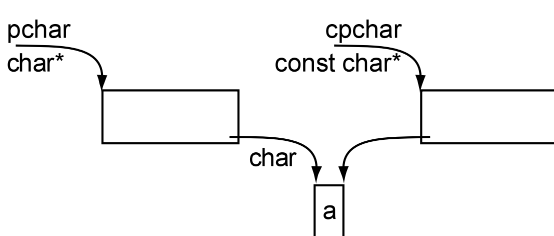


Рис. 3.14. Сталий формальний параметр

При цьому зміниться також значення указника на сталу:

```
cout<<cpchar<<':'
   <<pchar<<endl;
```

Однак заміна символу на сталу за допомогою указника залишається неможливою. Як бачимо, указник на сталу має обмежені права доступу до свого значення. Це гарантує його незмінність, скажімо, у разі передавання до функції за допомогою параметра.

На перший погляд, подвійний указник мав би поводитися так само й дозволяти ініціалізацію:

Лістинг 3.98

```
char** ppc=0; // Недопустимі ініціалізації:
const char** cppc=ppc;   // Помилкова команда
```

Проте знову одержимо повідомлення про помилку:

```
error C2440: 'initializing' : cannot convert from 'char** ' to 'const
char** '
```

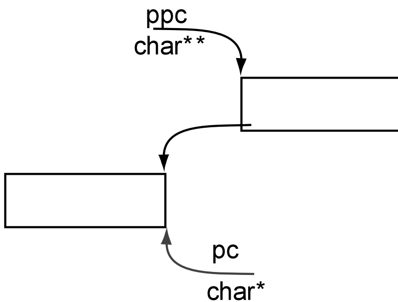


Рис. 3.15. Прихований указник

Щоб зрозуміти причину нелогічної, на перший погляд, поведінки компілятора, виконаємо таке дослідження. Візьмемо звичайний указник

```
char* pc=0;
```

і пов'яжемо його з указником другого рівня, наприклад, за допомогою ініціалізації:

```
char** ppc=&pc;
```

Результат такої ініціалізації зображено на рис. 3.15.

Тепер уявімо собі, що ініціалізація в лістингу 3.98 можлива:

```
// Робимо припущення про можливість некоректної
// ініалізації указника другого рівня
const char** cppc=ppc;
```

Оскільки ініціалізація подвійного указника на константу звичайним подвійним указником неможлива, примусимо компілятор дозволити її, узявши на себе відповідальність за перетворення типів, а тому модифікуємо некоректну ініціалізацію, щоб можна було спостерігати її наслідки:

```
// Хакерська ініціалізація
char** ppc=0; const char** cppc=(const char**)ppc;
```

Зв'язки між указниками стануть такими (рис. 3.16).

Тепер візьмемо сталий символ, наприклад такий:

```
const char chr='c';
```

Розіменування `*cppc` подвійного указника на сталий символ `cppc` має тип «указник на сталий символ», а тому може бути спрямоване на сталу:

```
*cppc=&chr;
```

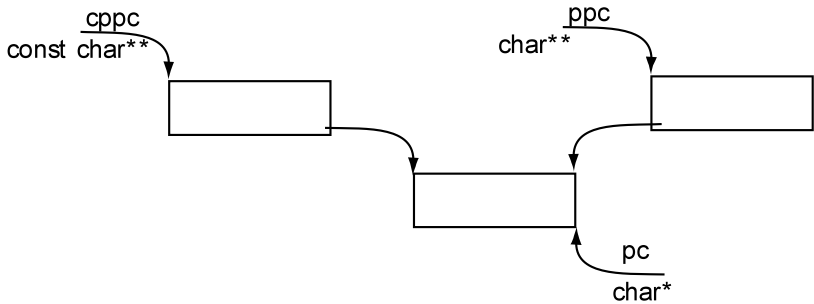


Рис. 3.16. Зв'язування сталого указника з несталим

Одержимо картину, зображену на рис. 3.17. Виникла явна суперечність: у лівій частині рисунка зібрано указники на сталі, у правій – на змінні. Однак ці дві частини взаємопов'язані: за допомогою указника `*cppc` звичайний указник `pc` одержав доступ до сталого символу `chr`. Ніщо не заважатиме тепер змінити значення, на яке спрямовано указник `pc`. Проте це константа, яку ніхто не має права змінювати:

```
*pc='A';
```

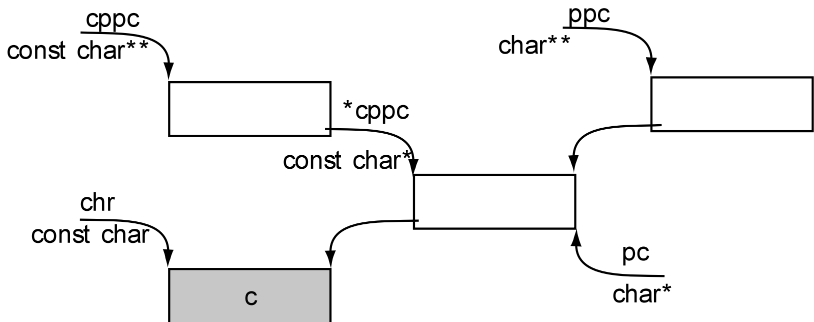


Рис. 3.17. Спрямування сталого указника на сталий символ

Одержимо помилкову ситуацію, зображену на рис. 3.18.

Подвійні указники поводяться зовсім не так, як звичайні. Останні дають змогу спрямовувати на значення звичайного указника указник на сталу, і це не створює суперечностей. Насправді указник на сталу величину має доступ до відповідного значення, але не може змінювати його.

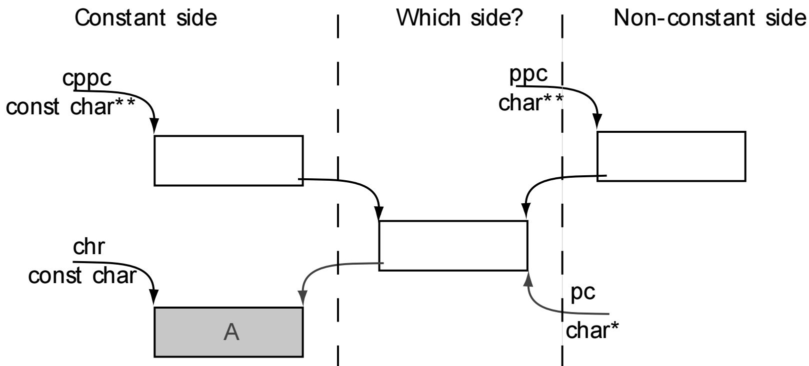


Рис. 3.18. Некоректна модифікація сталої

Подвійні указники потребують детальнішої дворівневої специфікації сталості. Справді, як видно з рис. 3.18, із подвійним указником пов'язано три рівні величин. Нульовий, основний, рівень – це саме значення (у нашій ситуації – стале); перший рівень, проміжний, – це звичайний указник, сталість якого не декларовано, так само як не вимагається сталості другого рівня (власне, самих подвійних указників). Тепер, зв'язавши подвійний указник на сталу величину з указником другого рівня, за допомогою останнього можна модифікувати стале значення через несталий проміжний перший рівень.

Вправа 3.14 (завдання для хакера)

Складіть програму, у якій дійсну константу `pi`,

```
const double pi=3.141592653589793;
```

можна було б вивести як `1` командою

```
cout<<pi<<endl;
```

Тому в разі переходу від подвійних указників до подвійних указників на сталі потрібно гарантувати сталість кожного рівня, тобто визначати указник сталого указника на сталу величину:

Лістинг 3.99

```
// Параметр-указник сталого указника сталої
void f(const char* const* s) {
   return;
}
// Тепер стає можливим виклик із фактичним
// параметром-указником другого рівня
char** s; f(s);
```

Вправа 3.15

Запропонуйте приклад програми, у якій буде використано «потрійну сталість подвійного указника»

```
void f(const char* const* const s);
```

Підіб'ємо підсумки. Непряму адресацію другого рівня, тобто указники указників або відсилки до указників використовують тоді, коли потрібно модифікувати указник, використаний як фактичний параметр. У разі виділення нової пам'яті залишається відкритою питання про відповідальність за її наступне видалення. Особливо важливі випадки модифікації – створення нового об'єкта, адреса якого повертається фактичним параметром, або видалення наявного об'єкта з одночасною нейтралізацією указника на нього (табл. 3.2).

Таблиця 3.2. Типи параметрів другого рівня

Параметри для іменування другого рівня		
Тип формального параметра	**Призначення формального параметра**	**Зауваження**
`T**`	Указник указника служить для повернення параметра-результату (у стилі чистої мови С)	Уживається для модифікації значення указника, зокрема в разі створення чи видалення об'єкта
`T*&`	Відсилка на указник теж служить для повернення параметра-результату (у стилі мови С++)	

3.3. Обчислення значення функції

Функції, що мають параметрами указники чи відсилки, можуть за їх допомогою повертати значення в програму, яка викликає ці функції, однак безпосереднє повернення результату самою функцією – фундаментальна математична традиція, якої дотримуються й у програмуванні. Її фундаментальність може засвідчити хоча б те, що зміну значення параметра зазвичай називають побічним ефектом, тоді як повернене функцією значення – її результатом; це записують, наприклад, як `f(x)`. Інакше замість присвоєння `y=f(x)` уживали б не таку зрозумілу команду виклику `f(x, y)`. Особливо зручним видається використання програмованих операцій, яке дає змогу зберегти звичний запис арифметичних виразів, наприклад `a+b;`, тоді як використання стороннього ефекту змушувало б нас вигадувати додаткові імена для відомих операцій, наприклад `plus` замість знака додавання, а сам арифметичний вираз записувати, наприклад, із застосуванням операції коми: `(plus(a,b,res), res);`.

Доповнимо наші знання про передавання параметрів відомостями про способи повернення результату.

3.3.1. Функції, що повертають значення

Ми постійно, навіть не задумуючись про це, користуємося виразами, побудованими за допомогою суперпозиції функцій або операцій. Кожному ще зі школи відома формула

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a},$$

але уявіть собі, що її довелося б запам'ятовувати як послідовність виразів:

u1=a·c; u2=4·u1; u3=b·b; u4=u3-u2; u5= $\sqrt{u4}$; u6=-b±u5; u7=2·a; u8=u6/u7;

або, ще гірше, програмувати так:

Лістинг 3.100

```
prod(a,c,u1); prod(4,u1,u2); prod(b,b,u3);
subtr(u3,u2,u4); sqrt(u4,u5); subtr(-b,u5,u6);
prod(2,a,u7); divide(u6,u7,u8);
```

Тому в мовах програмування так цінують можливість повернення функцією обчисленого значення. Розглянемо найпростіший випадок.

Лістинг 3.101

```
int min(int x, int y) {
   return x<y ? x : y;
}
```

Проте вже тут виникає запитання: де зберігати і як повертати обчислене значення? Візьмемо приклад складеного виразу, що містить виклик функції:

```
cout<<min(1,2)+10<<endl;
```

Як обчислене функцією значення, що являє собою перший доданок в арифметичному виразі, буде передано на обчислення суми? Відповідь на це запитання залежить від сигнатури функції. Якщо в ній тип значення визначено як арифметичний або як структуру, то функція повертає копію створеного правобічного значення.

Можливі два варіанти результату. Перший – звичайне значення фундаментального типу T. Тоді запишемо сигнатуру як

```
T f(…);                (→T)
```

Цей запис слід читати так: від функції з якими завгодно параметрами одержуємо величину типу T.

Другий варіант – це функції, що повертають стале значення:

```
const T f(…);            (→const T)
```

Для арифметичних типів сигнатури (→T) та (→**const** T) по суті означають одне й те саме. Для структур (а як побачимо пізніше – і для класів) відмінність виявляється суттєвою. Розглянемо приклад: визначимо функцію додавання точок площини Point

Лістинг 3.102

```
// Точка площини
struct Point {
   double _x; double _y;
};
```

із попереднього розділу:

Лістинг 3.103

```
// Невдала функція додавання
Point plus(const Point& u, const Point& v) {
   Point res={u._x+v._x, u._y+v._y};
   return res;
}
```

Одержимо на перший погляд звичайну функцію. Візьмемо дві точки площини.

Лістинг 3.104

```
const Point zero={0, 0};
const Point zero_one={0, 1}; const Point one_zero={1, 0};
```

Застосуємо до них операцію додавання, викликавши функцію plus. Усе виходить як належить, про що свідчать результати:

```
cout<<plus(zero,one)._x<<endl; cout<<plus(zero,one)._y<<endl;
```

Тепер запишемо дещо несподівану, може навіть безглузду команду присвоєння

```
plus(zero,one)=zero;
```

Компілятор пропускає цей вираз як коректний, і можна побачити результат його обчислення, наприклад, вивівши значення абсциси й ординати:

```
cout<<(plus(zero,one)=zero)._x<<endl;
cout<<(plus(zero,one)=zero)._y<<endl;
```

Не буде несподіванкою побачити, що виведені значення нульові.

Причина такої ситуації дуже проста. Виявляється, вираз plus(zero, one) має не тільки правобічне, а й лівобічне значення, а тому може набувати нового значення в результаті присвоєння. Заборонити це нескладно: по-

трібно правильно визначати сигнатуру функції, бо, як побачимо далі, лівобічне значення для деяких функцій може мати дуже важливі застосування.

Отож, у функції додавання точок має бути сигнатура вигляду (→**const** T), а не просто (→T):

Лістинг 3.105

```
// Поліпшена функція додавання
const Point plus(const Point& u, const Point& v) {
   Point res={u._x+v._x, u._y+v._y};
   return res;
}
```

Тепер спроба присвоєння призведе до помилки з повідомленням

```
error C2166: l-value specifies const object,
```

чого й потрібно було досягнути.

Розглянемо ще один приклад. Повернімося до визначення структури дерева:

Лістинг 3.106

```
// Вузол бінарного дерева
struct Tree  {
   int node; Tree* left; Tree* right;
};
```

Перепишемо функцію створення його вузла, зробивши її такою, що повертатиме значення. Почнімо з такого оголошення:

```
// Функція створення одного вузла дерева з двох піддерев left і right
// та запису у вузол розмітки node: несталий результат
const Tree createTree(int node, Tree* left, Tree* right);
```

Реалізацію можна уявити собі так: локальна змінна `Tree aTree` резервує пам'ять для полів структури, а далі відбувається їх заповнення та передавання одержаного значення в місце виклику.

Лістинг 3.107

```
// Функція створення одного вузла дерева з двох піддерев
// left і right та запису у вузол розмітки node: сталий результат
const Tree createTree(int node, Tree* left, Tree* right) {
   Tree aTree; aTree.node=node; aTree.left=left; aTree.right=right;
   return aTree;
}
```

У місці виклику визначимо іншу змінну – `Tree theTree`, яка приймає обчислену функцією вершину:

```
theTree=createTree(1, 0, 0);
```

Якщо порівняти щойно запропонований розв'язок з попереднім –

```
void createTree(Tree** aTree, int node, Tree* left, Tree* right);
```

то результат буде на користь попереднього. Справді, передаючи результат за допомогою параметра, ми оперували безпосередньо місцем пам'яті, у якому будували вершину дерева. Передаючи вершину як результат функції, ми постійно ініціюємо копіювання: скільки вершин, стільки й копій. Візьмемо, наприклад, ті самі три вершини, побудовані раніше:

```
Tree it1=createTree(1,0,0); Tree it2=createTree(2,0,0);
Tree it=createTree(3,&it1,&it2);
```

Копіювання відбудеться тричі.

Висновок. *Передавання результату функції значенням – найпростіший спосіб повернення результату: сигнатура вигляду(→T). Перевагу слід надавати сталому значенню: сигнатура вигляду(→**const** T).*

Вправа 3.16

Складіть функції для виконання арифметичних операцій над комплексними числами, заданими такою структурою:

Лістинг 3.108

```
// Комплексне число
struct Complex {
   double _re; double _im;
};
```

Вправа 3.17

Складіть функції для обчислення модуля комплексного числа, спряженого комплексного числа. Обґрунтуйте вигляд сигнатур.

3.3.2. Функції, що повертають указники та відсилки

Перейдемо до іншого типу сигнатур –

```
T* f(…);            (→T*)
```

і розглянемо функції, які провертають указники. Візьмемо, наприклад, такий варіант функції створення вершини дерева.

Лістинг 3.109

```
// Функція створення одного вузла дерева з двох піддерев
// left і right. Результат повертається за допомогою значення указника
Tree* createTree(int node, Tree* left, Tree* right) {
   Tree* aTree=new Tree;
   aTree->node=node; aTree->left=left; aTree->right=right;
   return aTree;
}
```

Указник `aTree` локалізовано в тілі функції, а тому він зникає після виходу з функції `createTree`. Однак створений ним за допомогою оператора **`new`** вузол дерева являє собою динамічний об'єкт, що існуватиме аж до свого видалення оператором **`delete`**. Зрозуміло, що використовувати створений таким способом об'єкт стає неможливо, якщо його адресу буде втрачено. Тому оператор виходу повертає значення указника на створений вузол дерева в точку виклику функції.

Тепер можна створити кілька вершин за допомогою одного присвоєння, у правій частині якого буде суперпозиція з трьох викликів, наприклад

```
Tree* t = createTree(3,createTree(1,0,0),createTree(2,0,0));
```

Тому допоміжні змінні не потрібні (порівняйте з обчисленням коренів квадратного рівняння на початку цього підрозділу). Так розв'язують розглянуту в попередньому підрозділі проблему завислих указників на піддерева, які можуть залишитися після видалення самого дерева. Для видалення всього дерева можна застосовувати наведену раніше функцію:

```
// void destroyTree(Tree*& aTree)
destroyTree(t);
```

Те саме можна одержати, скориставшись результатом-відсилкою. У цьому разі результатом виконання функції буде об'єкт, до якого має доступ тіло функції. Ідеться про структуру даних, створену функцією за допомогою оператора **`new`**. Схему для сигнатури відповідної функції позначимо так:

```
T& f(…);               (→T&)
```

Дещо зміниться сама функція:

Лістинг 3.110

```
// Функція створення одного вузла дерева з двох піддерев
// left і right. Результат повертається через відсилку
Tree& createTree (int node, Tree* left, Tree* right) {
   Tree* aTree=new Tree;
   aTree->node=node; aTree->left=left; aTree->right=right;
// Відсилка до значення указника
   return *aTree;
}
```

Тепер дуже важливо правильно працювати з пам'яттю. Є два способи правильного виклику наведеної функції створення вершини дерева. Перший із них – це звичайне використання указника:

Лістинг 3.111

```
// Ініціалізація указника деревом із трьома вузлами
Tree* pt=&createTree(3,&createTree(1,0,0),&createTree(2,0,0)));
```

```
// Видалення дерева й обнулення указника
destroyTree(pt);
```

Можливий також інший спосіб виклику без використання указника, але тоді потрібно користуватися псевдонімом, а не звичайною змінною. Справді, копіювання значення вершини дерева до локальної змінної

Лістинг 3.112

```
// Некоректно прийняте дерево
Tree t=createTree(3,&createTree(1,0,0),&createTree(2,0,0)));
```

зробить непридатною функцію видалення.

Вправа 3.18

Покроково виконайте функцію видалення дерева та відшукайте помилкову команду. Поясніть, де втрачається пам'ять.

Тому правильним буде використовувати відсилку на вершину, створену за допомогою функції `createTree()`:

Лістинг 3.113

```
// Дерево, коректно прийняте за допомогою відсилки
Tree& t=createTree(3,&createTree(1,0,0),&createTree(2,0,0)));
// Видалення дерева, обнулення указника не потрібне
destroyTree(&t);
```

Тепер функція видалення виконуватиметься коректно.

Висновок. *Уважно стежте за взаємною відповідністю виділення й вивільнення областей пам'яті глобально в усій програмі. Якщо виділену область пам'яті скопійовано під час повернення результату, то вона може залишитися сміттям, недоступним для вивільнення.*

Увага, типова помилка! *Щойно сконструйовані функції мають одну суттєву ваду (її, щоправда, мав і варіант із параметром-результатом). Вона полягає в тому, що функція звертається за виділенням динамічної пам'яті, але не відстежує її вивільнення. Ми ще повернемося до проблеми виділення-вивільнення пам'яті в наступному розділі, а поки що просто застережемо від спроби повернути адресу локального об'єкта, який перестане існувати після виходу з функції.*

Лістинг 3.114

```
// Помилка: повернення адреси локального об'єкта
Tree* createTree(int node, Tree* left, Tree* right) {
   Tree aTree;          // Визначаємо локальну змінну
   aTree.node=node;     // Заповнюємо локальну змінну значеннями
   aTree.left=left; aTree.right=right;
   return& aTree; // Адреса локальної змінної повертається як результат
// Локальну змінну видалено після виходу з функції
}
```

Вправа 3.19

Удоскональте запропоновану функцію створення вершини дерева так, щоб її результат не міг бути використаний у лівій частині присвоєння.

Ми розглянули повернення указника чи відсилки в разі створення нового об'єкта. Інший важливий випадок – повернення адреси чи відсилки в частину структури даних, якщо саму структуру було передано до функції як параметр. Відсилки дають повний доступ до частин структур , прийнятих функцією, але за умови вибору правильного способу передавання параметра (не варто забувати про проблеми копій і оригіналів даних).

Знову повернімося до точки на площині та розглянемо функцію доступу до її абсциси **double**& x(Point u):

Лістинг 3.115

```
// Передавання параметра за допомогою значення
// непридатне для повернення результату за допомогою відсилки
double& x(Point u) {
   return u._x;
}
```

Оскільки тип результату – відсилка, цю функцію можна записувати як справа, так і зліва від знака присвоєння. У першому випадку вона дає змогу одержати значення абсциси точки u:

Лістинг 3.116

```
Point v={10, 20};
cout<<x(v)<<endl; // Буде виведено 10
```

У другому її можна викликати так:

```
x(v)=125;
```

Залишається тільки розібратися, яке лівобічне значення функція поверне в такому разі. Для цього достатньо подивитися на значення абсциси точки v після виконання присвоєння:

```
Point v={10, 20};
x(v)=125; cout<<x(v)<<endl; // Буде виведено 10, а не 125!!!
```

Як ми пам'ятаємо, передавання фактичного параметра значенням v зумовлює ініціалізацію формального параметра u його копією Point u=v;, тому присвоєння x(v)=125; змінює u, не змінюючи v. Отже, не дивно, що наступний виклик функції x(v) в операторі виведення створить іще одну копію старого значення фактичного параметра та покаже абсцису відповідної точки. Більше того, саме присвоєння x(v)=125; не можна

вважати допустимим, оскільки за межами функції виконується присвоєння за лівобічним значенням, яке було місцем локалізації формального параметра, видаленого після виходу з функції. Захистом у такій ситуації могло б бути використання сталого параметра – **const** Point u, але тепер стає помилковою сама сигнатура:

Лістинг 3.117

```
// Передавання параметра за допомогою сталого значення
// робить передавання результату за допомогою відсилки
// синтаксично неправильним
double& x(const Point u) {
   return u._x;
}
```

Тепер система програмування не допустить спроби некоректного використання параметра. Функція, що може змінити значення сталого параметра через доступ до нього за допомогою результату-відсилки, виявляється синтаксично некоректною. З'явиться діагностичне повідомлення на зразок такого:

```
error C2440: 'return' : cannot convert from 'const double' to 'double& '
```

Як бачимо, дотримуючись дисципліни програмування, можна одержати більший захист з боку системи програмування, яка не пропускає багатьох сумнівних дій. Якщо ж ми все-таки захочемо викликати функцію зліва від знака присвоєння, то маємо подбати, щоб область пам'яті, призначена для лівобічного значення, не очищувалася після виходу з функції. Цього можна досягти кількома способами, зокрема відсилкою до зовнішньої відносно функції області пам'яті. Функції може бути надано доступ до неї за допомогою фактичного параметра, якщо для його передавання не потрібне копіювання. Отже, у правильному варіанті параметр має передаватися за допомогою відсилки:

Лістинг 3.118

```
// Функція повного доступу до абсциси параметра:
// передавання параметра за допомогою відсилки вможливлює
// повноцінне передавання за допомогою відсилки результату
double& x(Point& u) {
   return u._x;
}
```

Тепер адреса абсциси формального параметра u буде водночас адресою абсциси фактичного параметра, визначеного в області виклику, а не всередині функції. Цю функцію можна використовувати як для читання абсциси точки, так і для її заміни. Одержимо очікуваний результат:

```
x(v)=125;
cout<<x(v)<<endl; // Буде виведено 125!!!
```

Лістинг 3.119

Проте трапиться інша несподіванка: спроба вивести значення абсциси сталої точки

```
const Point zero={0, 0};
cout<<x(zero)<<endl;
```

Лістинг 3.120

виявиться помилковою. Виведеться діагностичне повідомлення

```
error C2664: 'x' : cannot convert parameter 1 from 'const struct Point'
to 'struct Point&'
```

У точній специфікації сталих типів і їх захисті не лише від безпосередніх змін, але й від тих, що відбуваються за допомогою непрямого доступу, виявляється сильна типізація мови програмування C++. Справді, результат функції визначено так, що він надає повний доступ до параметра. Тому параметр не може бути сталим, оскільки інакше функція дозволяла б його модифікацію.

Доповнимо функцію повного доступу додатковою функцією для доступу до абсциси сталого параметра. Для сталих параметрів може бути такий варіант функції:

```
// Модифікація функції повного доступу:
// обмежений доступ до абсциси сталого параметра
const double& x(const Point& u) {
   return u._x;
}
```

Лістинг 3.121

Завдяки йому стає неможливим застосування функції зліва від присвоєння. Строга типізація знову ж дає системі програмування змогу розрізняти й спільно використовувати сигнатури повного й обмеженого доступу в одній програмі. Вивченню цієї можливості присвячено наступний підрозділ.

```
// Функція повного доступу до абсциси параметра:
// передавання параметра за допомогою відсилки вможливлює
// повноцінне передавання за допомогою відсилки результату
double& x(Point& u);
// Модифікація функції повного доступу: обмежений доступ
// до абсциси сталого параметра
const double& x(const Point& u);
```

Лістинг 3.122

Зроблені висновки відобразимо в табл. 3.3.

Таблиця 3.3. Типи результатів функцій

Тип сигнатури функції	Коментар
`T f(…)`	Результат арифметичного типу, створений у тілі функції, передаємо значенням; тип параметра довільний
`const T f(…)`	Структуру, створену в тілі функції, передаємо сталим значенням; тип параметра довільний
`T* f(…)`, `T& f(…)`	Створені за допомогою функції структури даних передаємо за допомогою указника (стиль мови C) чи за допомогою відсилки (стиль мови C++); тип параметра довільний
`const T&` `f(const S&)`	Доступ до читання всього агрегату даних, переданого фактичним параметром – сталою відсилкою, забезпечуємо результатом – сталою відсилкою
`T& f(S&)`	Доступ до читання та змінення всього агрегату даних, переданого фактичним параметром, або його частини –відсилки, забезпечуємо результатом-відсилкою

3.4. Статичний поліморфізм

У математиці зазвичай економлять на позначеннях, використовуючи одні й ті самі символи в різних значеннях залежно від контексту. Так, для операції додавання завжди вживають один і той самий символ, хоча в кожній конкретній ситуації вона виконується за іншим правилом. Наприклад, є такі додавання:

- натуральних чисел: 1+2;
- цілих чисел: (–1)+(–2);
- раціональних чисел: 1/3 + 5/8;
- дійсних чисел: $\varphi+2\pi k$;
- комплексних чисел: $(a+bi)+(c+di)$;
- векторів: $u+v$;
- матриць: $A+B$.

Тепер розглянемо засоби, які дають змогу користуватись одними й тими самими позначеннями для схожих операцій над даними різних типів.

3.4.1. Довизначення функцій

Згадаймо функцію обміну значеннями змінних. Для цілих чисел ми мали дві функції:

Лістинг 3.123

```
void pswap(int* x, int* y); // стиль мови C
void rswap(int& x, int& y); // стиль мови C++
```

Як назвати таку саму функцію для дійсних (символьних та інших) значень?

Лістинг 3.124

```
void pswap_char(char* x, char* y);
void pswap_double(double* x, double* y);
```

Звичайно, найкраще було б зберегти одне позначення. Для цього домовилися, що функцію задає її сигнатура, а не тільки ім'я, і кожній сигнатурі можна приписати свою реалізацію.

Лістинг 3.125

```
// Виконати обмін значеннями цілих
void rswap(int& x, int& y) {
   int z=x; x=y; y=z;
   return;
}
// Виконати обмін значеннями символів
void rswap(char& x, char& y) {
   char z=x; x=y; y=z;
   return;
}
// Виконати обмін значеннями дійсних
void rswap(double& x, double& y) {
   double z=x; x=y; y=z;
   return;
}
```

У такому разі за типами фактичних параметрів можна визначити, яку саме функцію викликано.

Лістинг 3.126

```
char a, b; int m, n; double x, y;
rswap(a, b); // Виконано обмін значеннями символів
rswap(m, n); // Виконано обмін значеннями цілих
rswap(x, y); // Виконано обмін значеннями дійсних
```

Щойно розглянутий прийом являє собою частковий випадок *поліморфізму* (polymorphism) – властивості програмного тексту бути придатним для застосування до об'єктів двох або більше типів. У нашій ситуації маємо так званий *статичний поліморфізм*, за якого типи об'єктів, яких він стосується (у нашій ситуації – параметрів), визначають до початку виконання програми, на етапі її складання, а саме способом запису виклику функції, точніше типами фактичних параметрів. Якщо взяти символьні величини `char a, b`, то відбудеться виклик функції `rswap(a, b)` з сигнатурою з параметрами символьного типу:

```
void rswap(char&, char&);
```

Більше того, зникає потреба навіть у різних позначеннях залежно від способу передавання параметра за допомогою указника чи відсилки – `rswap` і `pswap`, оскільки за викликами `swap(a, b)` чи `swap(&a, &b)`, знаючи типи фактичних параметрів, компілятор може однозначно розпізнати, чи йдеться про варіант формального параметра-відсилки чи указника.

Лістинг 3.127

```
// Виконати обмін значеннями за відсилками цілих
void swap(int& x, int& y) {
   int z=x; x=y; y=z;
   return;
}
// Виконати обмін значеннями за указниками на цілі
void swap(int* x, int* y) {
   int z=*x; *x=*y; *y=z;
   return;
}
// Виконати обмін значеннями за відсилками символів
void swap(char& x, char& y) {
   char z=x; x=y; y=z;
   return;
}
// Виконати обмін значеннями за указниками на символи
void swap(char* x, char* y) {
   char z=*x; *x=*y; *y=z;
   return;
}
// Виконати обмін значеннями за відсилками дійсних
void swap(double& x, double& y) {
   double z=x; x=y; y=z;
   return;
}
// Виконати обмін значеннями за указниками на дійсні
void swap(double* x, double* y) {
   double z=*x; *x=*y; *y=z;
   return;
}
```

У наведеному далі лістингу 3.128 за типом фактичних параметрів розпізнано, яку саме функцію слід викликати.

Лістинг 3.128

```
char a, b;
swap(a, b);   // Виконуємо обмін значеннями символів
swap(&a, &b); // Виконуємо обмін значеннями за указниками на символи
int m, n;
```

```
swap(m, n);   // Виконуємо обмін цілими значеннями
swap(&m, &n); // Виконуємо обмін значеннями за указниками на цілі
double x, y;
swap(x, y);   // Виконуємо обмін дійсними значеннями
swap(&x, &y); // Виконуємо обмін значеннями за указниками на дійсні
```

Пригадайте, що розгляд способів передавання параметрів ми почали з традиційної функції-обманки:

Лістинг 3.129

```
// Функція, яка не виконує обміну значеннями параметрів
void swap(double x, double y) {
   double z=x; x=y; y=z;
   return;
}
```

Сигнатура цієї функції відрізняється від кожної з наведених у цьому підрозділі. Але вона не може співіснувати з ними, оскільки в такому разі виникала б неоднозначність під час виклику; про це відразу повідомить компілятор, якщо буде спроба використати її одночасно з функцією, що має параметри-відсилки.

Висновок. Одноймені функції мають різнитися типом параметрів так, щоб за викликом можна було однозначно вибрати потрібну функцію. Таке розпізнавання стає неможливим, якщо функції різняться лише типом результату чи параметри різняться способом передавання за допомогою значення чи відсилки, адже як для значення, так і для відсилки типи фактичних параметрів збігаються. Тоді відповідні сигнатури стають несумісними. Приклади несумісних сигнатур: `int f(double)` і `double f(double)` різняться лише типом результату; `double f(double)` і `double f(double&)` – у першій параметр передається значенням, а в другій – за допомогою відсилки. Однак сигнатура `double f(double*)` сумісна окремо з кожною із сигнатур як із `double f(double)`, так і з `double f(double&)`.

Розглянутий прийом називають *довизначенням* (overloading) чи *поширенням функцій* на інші типи даних. Це один із проявів статичного поліморфізму. Статичним його називають тому, що на етапі компіляції, тобто ще до початку виконання програми, з виклику стає зрозумілим, яку саме функцію викликають.

Одноймені функції з різними сигнатурами можуть як завгодно відрізнятися своєю реалізацією. Так, скажімо, для переставляння цілих чисел можна було б запропонувати реалізацію, що не потребує додаткової змінної:

Лістинг 3.130

```
// «Хитрий» обмін значеннями за допомогою указників
void swap(int* x, int* y) {
   *x=*x+*y; *y=*x-*y; *x=*x-*y;
```

```
   return;
}
// Звичайний обмін значеннями за допомогою відсилок
void swap(int& x, int& y) {
   x=x+y; y=x-y; x=x-y;
}
```

При цьому реалізації для дійсних чисел і символів залишилися б без змін.

Вправа 3.20

Реалізуйте пару функцій `max()` для пошуку максимального значення у векторі:

а) з цілих чисел;

б) дійсних чисел.

Вправа 3.21

Реалізуйте пару функцій для обчислення натурального степеня:

а) дійсного числа;

б) комплексного числа.

Вправа 3.22

Реалізуйте набір функцій сортування за зростанням:

а) рядка символів;

б) рядка цілих чисел;

в) рядка дійсних чисел.

3.4.2. Параметри за замовчуванням

Інший близький за використанням інструмент називають *параметрами за замовчуванням* (*default parameter*). Повернімося до структури `Point` з лістингу 3.103 і визначимо функцію створення точки.

Лістинг 3.131

```
// Функція створення точки на площині
const Point createPoint(double x, double y) {
   Point res={x, y};
   return res;
}
```

Ось приклади її використання.

Лістинг 3.132

```
Point zero=createPoint(0,0);
Point one=createPoint(1,0); Point one_one=createPoint(1,1);
```

Тепер приготуємо таке оголошення функції `createPoint` для заголовного файлу:

```
const Point createPoint(double x=0, double y=0);
```

На підставі такого оголошення стають можливими три види викликів: зовсім без фактичних параметрів, з одним фактичним параметром, з двома фактичними параметрами:

Лістинг 3.133

```
// Значення обох фактичних параметрів
// буде взято за замовчуванням
Point zero=createPoint();
// Значення другого фактичного параметра буде взято за замовчуванням
Point one=createPoint(1);
// Значення обох фактичних параметрів задано викликом
Point one_one=createPoint(1,1);
```

Значення фактичних параметрів, не задані у виклику, буде взято з оголошення, що міститься в заголовному файлі. Правило підстановки значень формальних параметрів дуже просте: спочатку компілятор бере фактичні параметри з виклику, а потім, якщо фактичних параметрів забракне, їх буде доповнено значеннями зі списку заданих за замовчуванням.

Розпізнавання викликаної функції компіляторові ускладнює те, що можна перетворювати типи даних. Уявімо собі, що ми визначили дві функції:

Лістинг 3.134

```
void show(double x) {cout<<x<<endl;}
void show(int x) {cout<<x<<endl;}
```

Одна функція має дійсний параметр, друга –цілий. Тепер розглянемо контекст і виклики:

Лістинг 3.135

```
int m=10; double a=1.25; short sh=25;
char ch='A'; float fl=2.5;
show(m); show(a); show(sh); show(ch); show(fl);
```

Строго кажучи, наявним сигнатурам відповідають лише два з них, але й решта чотири компілятор розпізнає як правильні. Він щоразу діятиме приблизно так:

Лістинг 3.136

```
// Без перетворення типів:
show(m); show(a);
// Підвищення типу фактичного параметра відповідно до
// наявних сигнатур функції.
// Перетворення до цілого
int tmp1=sh; show(tmp1); int tmp2=ch; show(tmp2);
int tmp3=l; show(tmp3);
```

```
// Перетворення до дійсного
double tmp4=fl; show(tmp4);
```

Оскільки перетворення типів не завжди відбуваються так очевидно, як у наведеному прикладі, радимо стежити за відповідними застереженнями компілятора й уникати сумнівних або невиправданих перетворень.

Ускладнимо завдання компілятору, додавши до двох визначених вище в лістингу 3.136 іще одну функцію з першим параметром типу **short**, а другим – символьним зі значенням за замовчуванням:

Лістинг 3.137

```
void show(short x, char currency='$') {
   cout<<x<<' '<<currency<<endl;
}
```

Тепер команда show(sh) виконає виклик цієї функції та виведе текст «25 $», якщо sh визначено як **short**. Водночас спроба замінити параметр функції з лістингу 3.137 на цілий

Лістинг 3.138

```
void show(int x, char currency='$') {
   cout<<x<<' '<<currency<<endl;
}
```

спричинить помилку компіляції

```
error C2668: 'show' : ambiguous call to overloaded function
```

оскільки, наприклад, виклик show(12) став би неоднозначним. Справді, за цим викликом неможливо з'ясувати, ідеться про функцію з одним цілим параметром чи про однойменну функцію з двома параметрами, один з яких задано за замовчуванням.

Нагадаємо, що не все, що правильно для передавання параметрів за допомогою значення, залишається таким і для посилань або указників. Передусім це стосується автоматичних перетворень типів, які в такій ситуації стають неприпустимими. Наприклад, для функції

Лістинг 3.139

```
void show(int& x) {
   cout<<x<<endl;
}
```

не можна використовувати фактичний параметр жодного іншого типу, крім цілого. Справді, виклик show(ch) завершиться повідомленням

```
error C2664: 'show' : cannot convert parameter 1 from 'char' to 'int& '
```

Тепер бачимо, що статичний поліморфізм разом із механізмом заданих за замовчуванням параметрів і перетворенням типів даних ставить перед компілятором складне завдання вибору належної специфікації для

викликаної функції. Вираз `f()` не обов'язково являє собою виклик функції без параметрів. Він може бути викликом функції з багатьма параметрами, усі з яких задано за замовчуванням.

Механізми заданих за замовчуванням параметрів і довизначених функцій інколи можуть доповнювати чи заміняти один одного. Уявімо, наприклад, систему управління навчальним процесом із можливістю підрахунку показників успішності студентів. Відомо, що за сесію не буває більше ніж п'ять іспитів. Треба розробити функції, які для кожного студента визначать його максимальний і середній бал залежно від кількості складених ним іспитів.

Варіант 1. Означаємо п'ять функцій `maxGrade`:

Лістинг 3.140

```
int maxGrade(int);
int maxGrade(int, int);
int maxGrade(int, int, int);
int maxGrade(int, int, int, int);
int maxGrade(int, int, int, int, int);
```

і записуємо п'ять реалізацій:

Лістинг 3.141

```
int maxGrade(int m) {
   return m;
}
int maxGrade(int m1, int m2) {
   return (m1>m2) ? m1 : m2;
}
int maxGrade(int m1, int m2, int m3) {
   return maxGrade(m1, maxGrade(m2, m3));
}
int maxGrade(int m1, int m2, int m3, int m4) {
   return maxGrade(m1, maxGrade(m2, m3, m4));
}
int maxGrade(int m1, int m2, int m3, int m4, int m5) {
   return maxGrade(m1, maxGrade(m2, m3, m4, m5));
}
```

Варіант 2. Оголошуємо замість п'яти попередніх одну функцію з п'ятьма параметрами, чотири з яких мають задані за замовчуванням значення:

```
int maxGrade(int m1, int m2=0, int m3=0, int m4=0, int m5=0);
```

і записуємо одну реалізацію:

Лістинг 3.142

```
int maxGrade(int m1, int m2, int m3, int m4, int m5) {
   int m=(m1>m2) ? m1 : m2; int n=(m3>m4) ? m3 : m4;
```

```
   m=(m>n)? m :n;
   return (m>m5)? m : m5;
}
```

Підрахунок середнього балу не вдасться запрограмувати однією функцією. Доведеться скористатися набором із п'яти однойменних функцій:

Лістинг 3.143

```
float aveGrade(int);
float aveGrade(int, int);
float aveGrade(int, int, int);
float aveGrade(int, int, int, int);
float aveGrade(int, int, int, int, int);
```

з очевидними реалізаціями, які пропонуємо розробити самостійно.

3.5. Програмування операцій

Згадаймо раніше зроблене неоковирне визначення операції додавання точок площини:

Лістинг 3.144

```
// Функція додавання точок
const Point plus(const Point& u, const Point& v) {
   Point res={u._x+v._x, u._y+v._y};
   return res;
}
```

Його неоковирність полягала в тому, що додавання двох точок записували як `plus(a,b)`, а не звичним виразом `a+b`. Після ознайомлення з концепцією довизначення функцій можна сказати, що мова програмування допускає стільки реалізацій операції додавання, скільки в ній типів. Якщо припуститися певної вольності в іменуванні функцій, дозволивши крім ідентифікаторів також імена вигляду **operator**ω, де ω – це символ однієї зі стандартних операцій, то компілятор розумітиме конструкцію

Лістинг 3.145

```
// Операція додавання точок площини
const Point operator+(const Point& u, const Point& v) {
   Point res={u._x+v._x, u._y+v._y};
   return res;
}
```

як визначення функції, виклик якої крім звичайного запису `operator+(a, b)` може мати скорочений вигляд `a+b`.

Вправа 3.23

Визначте тип раціональних чисел як пару, що складається з цілих чисельника та знаменника, і реалізуйте арифметичні операції над ними.

Для точок площини можна визначати багато інших операцій, наприклад, порівняння

```
// Операція порівняння точок площини                    Лістинг 3.146
bool operator==(const Point& u, const Point& v) {
   return (u._x==v._x)&&(u._y==v._y);
}
```

і навіть, як побачимо нижче, уведення та виведення.

За допомогою довизначення операцій можна поширити арифметичну, логічну чи іншу операцію на довільний програмований тип даних, проте не можна змінювати реалізації стандартних операцій на стандартних типах даних. Приймаючи рішення про визначення операції, слід зважати на міркування щодо доцільності та повноти визначень, узгоджуючи реалізацію власних операцій із загальновизнаними властивостями однойменних стандартних операцій. Наприклад, варто подбати про асоціативність додавання чи множення, взаємну оберненість додавання та віднімання тощо. Визначення операції перевірки на рівність потрібно доповнювати визначенням її заперечення – перевірки на нерівність:

```
// Операція порівняння точок площини на нерівність      Лістинг 3.147
bool operator!=(const Point& u, const Point& v) {
   return !(u==v);
}
```

Одна з принципових властивостей арифметичних операцій – закон асоціативності додавання та множення. Якщо попередньо домовитися про порядок виконання операцій (зазвичай зліва направо, тоді як операції присвоєння виконуються справа наліво), то асоціативність робить деякі з дужок у виразах зайвими. Замість ((a+b)+c)+d) досить написати a+b+c+d.

Операції, поширені на програмно створені типи даних, мають зберігати властивості стандартних. Тому для операцій типовою є сигнатура у вигляді функції, що повертає результат певного типу. Крім цього, в операціях використовуються здебільшого параметри сталих типів. Виняток становлять операції інкременту й декременту, а також операція введення, пов’язана як зі змінною, значення якої вводиться, так і зі станом вхідного потоку. Операція ж виведення змінює лише стан вихідного потоку.

3.5.1. Програмування операцій уведення та виведення

Отже, існують операції, для яких поняття результату не таке очевидне, як для арифметичних. До них належать уже згадані операції введення та виведення. Як приклад запишемо операцію виведення точок площини.

Вона має два параметри: вихідний потік виведення та точку площини. Вихідний потік передають за допомогою відсилки (передавання за допомогою значення не використовують, оскільки його копіювання може бути взагалі забороненою операцією). Точку площини можна передавати різними способами, найпростіший з яких – використання сталої відсилки. Операція має результат – оновлений вихідний потік, який знову ж повертають через відсилку:

Лістинг 3.148

```
// Операція виведення точок площини
ostream& operator<<(ostream& os, const Point& u) {
   os<<'('<<u._x<<','<<u._y<<')';
   return os;
}
```

Оскільки вихідний потік є як першим операндом, так і результатом виведення, він не може бути сталим.

```
// Приклад виведення точок площини
Point u(1,1), v(1,2); cout<<u; cout<<v; cout<<endl;
```

Зручно, що замість нього в оператор виведення можна підставити інше виведення, результат якого – знову ж вихідний потік. Цей прийом можна застосовувати багаторазово, виводячи одним оператором виведення довільну кількість значень.

```
// Приклад виведення двох точок площини одним оператором виведення
Point u(1,1), v(1,2); cout<<u<<v<<endl;
```

Останній рядок відповідає такій розстановці дужок:

```
((cout<<u)<<v)<<endl;
```

тобто до оператора `cout<<endl;` підставлено `cout<<v`, а до нього, у свою чергу, – `cout<<u`.

Аналогічно можна визначити операцію введення. Несталими тепер стануть обидва параметри:

Лістинг 3.149

```
// Операція введення точок площини
istream& operator>>(istream& is, Point& u) {
   is>>u._x>>u._y;
   return is;
}
```

Вправа 3.24

Визначте операції введення та виведення раціональних чисел (див. вправу 3.23).

Наведемо приклад застосування наведених операцій.

Лістинг 3.150

```
Point u, v, w;
cout<<"Give the first point";
cin>>u._x>>u._y;              // Стандартні оператори введення
cout<<"Show the first point";
cout<<u._x<<','<<u._y<<endl;   // Стандартні оператори виведення
cout<<"Give the second point";
cin>>v;                        // Програмований оператор введення
cout<<"Show the second point";
cout<<v<<endl;                 // Програмований оператор виведення
cout<<"Check if they are equal(1/0)";
// Програмовану операцію порівняння суміщено зі стандартним виведенням
 cout<<(u==v)<<endl;
// Програмоване додавання
cout<<"Add two points"<<endl;
w=u+v; cout<<"Add and show two points"<<endl;
// Програмоване додавання, суміщене з виведенням
cout<<u+v<<endl;
```

3.5.2. Програмування арифметичних операцій

Арифметичні операції часто виконують над аргументами різних типів. Є два способи їх реалізації: мішані операції та операції з перетворенням типів. Поки що розглянемо перший, а до другого повернемося трохи згодом. Для прикладу зручно буде розглянемо комплексні числа. Одразу зазначимо: у бібліотеці С++ передбачено окремий тип для комплексних чисел, його визначено в заголовному файлі `complex` і названо так само. Ми ж будемо визначати власні комплексні числа, називаючи їх `Complex` (із великої літери), передусім для того, щоб навчитися програмувати типи даних. Цей же прийом застосуємо пізніше до типу `string` і деяких інших типів.

Отож, визначимо тип комплексних чисел як структуру:

Лістинг 3.151

```
// Програмований тип комплексних чисел
struct Complex {
   double _re; double _im;
};
```

Випишемо три сигнатури для операції додавання, кожна для окремої пари аргументів,

Лістинг 3.152

```
// Три варіанти додавання комплексних чисел
const Complex operator+(const Complex&, const Complex&);
const Complex operator+(double, const Complex&);
```

```
const Complex operator+(const Complex&, double);
```

із відповідними реалізаціями:

Лістинг 3.153

```
// Додавання двох комплексних чисел
const Complex operator+(const Complex& u, const Complex& v) {
   Complex res={u._re+v._re, u._im+v._im};
   return res;
}
// Додавання дійсного числа до комплексного
const Complex operator+(const Complex& u, double x) {
   Complex res={u._re+x, u._im};
   return res;
}
// Додавання комплексного числа до дійсного
const Complex operator+(double x, const Complex& v) {
   Complex res={x+v._re, v._im};
   return res;
}
```

Виведення для комплексних чисел визначають так само, як і для точок площини, за сигнатурою

```
ostream& operator<<(ostream& os, const Complex&);
```

Вправа 3.25

Визначте операції введення та виведення комплексних чисел.

Визначимо кілька комплексних і дійсних чисел, застосуємо до них запропоновані операції:

Лістинг 3.154

```
// Приклад операцій із комплексними числами
const Complex i={0, 1}; Complex z={1, 1}; double x=2, y=3;
cout<<(x+5)+(z+i)+y<<endl;
```

Зауважимо, що, зокрема, у виразі

```
cout<<(x+5)+(z+i)+y<<endl;
```

використано чотири різні операції додавання та дві операції виведення. Проаналізуйте їх.

Вправа 3.26

Визначте і реалізуйте повний набір арифметичних і логічних операцій над комплексними числами.

Так само можна визначити операції над векторами:

```
// Структура загорнутих векторів                    Лістинг 3.155
struct WrappedVector {
   static const int n; double* x;
};
```

зокрема операцію скалярного добутку векторів

```
double operator*(const WrappedVector&, const WrappedVector&);
```

чи векторного добутку

```
const WrappedVector operator*(const WrappedVector&, const WrappedVector&);
```

Однак не можна в одному проекті використати водночас обидві операції (поясніть, чому).

Вправа 3.27

Реалізуйте окремо кожну з операцій множення векторів.

Вправа 3.28

Визначте й реалізуйте операції додавання векторів і множення вектора на число.

3.6. Узагальнені функції

Переглянемо ще раз визначення трьох функцій `swap`:

```
// Виконати обмін значеннями цілих                  Лістинг 3.156
void swap(int& x, int& y) {
   int z=x; x=y; y=z;
   return;
}
// Виконати обмін значеннями символів
void swap(char& x, char& y) {
   char z=x; x=y; y=z;
   return;
}
// Виконати обмін значеннями дійсних
void swap(double& x, double& y) {
   double z=x; x=y; y=z;
   return;
}
```

Єдина відмінність між ними – типи їхніх параметрів. Тому створення функції переставляння точок площини чи, скажімо, комплексних чисел стає суто механічною справою: до тексту однієї з функцій слід замість наявного типу підставити `Point` або `Complex`.

3.6.1. Шаблони функцій

Спосіб текстового підставляння конкретного типу замість параметра, що задає ім'я типу, дуже давно відомий у програмування під назвою *узагальненого* чи *параметричного програмування* (generic programming), а самі узагальнені функції дістали назву *шаблонів функцій* (function template). До тексту функції додають особливий заголовок **`template`**, у якому задають параметри шаблону. Цими параметрами можуть бути імена типів, про що свідчить ключове слово **`typename`**. Визначення узагальненої функції переставляння набуде такого вигляду:

Лістинг 3.157

```
// Оголошення параметризованого типу Type
template <typename Type>
// Визначення узагальненої функції
void mySwap(Type& x, Type& y) {
   Type z=x; x=y; y=z;
   return;
}
```

Зауваження. *Так само, як тип `complex`, в бібліотеках C++ є шаблон `swap`, тому довелося використати іншу назву – `mySwap`. Можна було б узяти ім'я `Swap`, однак не будемо іменувати функції з великої літери.*

Визначення узагальнених функцій компілятор обробляє інакше, аніж визначення звичайних, з яких він безпосередньо створює об'єктний код. Побудувати об'єктний код узагальненої функції неможливо, оскільки на цьому етапі ще не відомі типи параметрів, а значить, і спосіб виконання операцій, які потрібно до них застосовувати. Справді, для переставляння символів потрібно переслати один байт, а для переставляння дійсних – вісім. Компілятор може лише зафіксувати наявність шаблону, створюючи його реалізації окремо для кожного виклику залежно від конкретних типів, використаних у ньому замість типових параметрів. Остання дія називається конкретизацією шаблону. Так, у разі визначення символьних змінних **`char`** `a, b` для виклику `mySwap(a, b)` текст шаблону буде конкретизовано у визначення функції переставляння символів, за яким і буде побудовано відповідний об'єктний код, а в контексті **`float`** `a, b` за викликом `mySwap(a, b)` – код для переставляння дійсних чисел. Узагальнені функції можна конкретизувати програмованими типами, навіть такими, яких не існувало на момент створення шаблону: наприклад, у контексті визначень `Point u, v` за викликом `mySwap(u, v)` компілятор генерує об'єктний код переставляння комплексних значень і відповідний виклик цього коду.

Принцип статичного поліморфізму поширюється й на узагальнені фу-

нкції. Водночас із наведеним вище шаблоном у програмі можна використовувати інший шаблон з тим самим ім'ям, але іншими типами параметрів, скажімо такий:

Лістинг 3.158

```
// Визначення однойменної узагальненої функції
template <typename T>
void mySwap(T* x, T* y) {
   T z=*x; *x=*y; *y=z;
   return;
}
```

Конкретизація цього шаблону відбуватиметься за типами параметрів – адресами чи указниками, які використано у виклику:

Лістинг 3.159

```
int m=1, n=2;
mySwap(m, n);     // Параметри - числа: перша функція
mySwap(&m, &n);   // Параметри - адреси: друга функція
double* pa=new double(100), *pb=new double(200);
mySwap(pa, pb);   // Параметри - указники: друга функція
mySwap(*pa, *pb); // Параметри - числа: перша функція
```

Більше того, статичний поліморфізм допускає одночасне використання однойменних звичайних та узагальнених функцій. Разом з обома наведеними шаблонами у тексті програми можуть бути й визначення конкретних функцій, зокрема окремі реалізації:

Лістинг 3.160

```
// «Хитрий» обмін значеннями
void mySwap(int& x, int& y) {
   x=x+y; y=x-y; x=x-y;
}
void mySwap(int* x, int* y) {
   *x=*x+*y; *y=*x-*y; *x=*x-*y;
}
```

Конкретні реалізації функцій мають вищий пріоритет над однойменними шаблонами. У першу чергу компілятор обирає конкретні реалізації, виконуючи конкретизацію шаблону лише в тому разі, коли конкретні реалізації не підходять. Шаблони можуть накладати певні обмеження на типи, застосовані в їх конкретизації. Повернувшись до прикладу з максимумом, одержимо шаблон

```
template <typename Type>
Type myMax(Type x, Type y) {
   return (x>y ? x : y);
}
```

Якщо у визначенні шаблону mySwap використано лише присвоєння, то у визначенні шаблону для максимуму фігурує операція порівняння. Тому його можна застосувати не до будь-якого типу, а лише до того, для якого визначено цю операцію. Компілятор, скажімо, не зможе згенерувати код для спроби обчислити максимум двох точок:

Лістинг 3.161

```
Point u, v, w; w=max(u, v);
```

Річ у тім, що немає відповідної операції порівняння точок.

3.6.2. Особливості конкретизації шаблонів функцій

Розглянемо ще одну проблему. Спробуємо дослівно переписати функцію піднесення до степеня у вигляді шаблону:

Лістинг 3.162

```
// Узагальнена функція швидкого піднесення
// чого завгодно до натурального степеня
template <typename Type>
const Type power(const Type base, const int exponent) {
   Type y=1; int n=exponent; Type x=base;
   while(n>0) {
      if(n % 2==0) {x=x*x; n=n/2;}
      else {y=y*x; n=n-1;}
   }
   return y;
}
```

Застосувати його можна лише до типу, до якого застосовний літерал 1, наприклад **double**:

Лістинг 3.163

```
// Конкретизація піднесення дійсного числа
// до натурального степеня
int n; double x;
cout<<"n,x="; cin>>n>>x; cout<<power(x,n)<<endl;
```

Можемо взяти комплексні числа й визначити для них операцію множення, однак шаблон виявиться непридатним: ми поки що не вміємо розповсюдити літерал 1 для його застосування в комплексному типі. Трохи далі спробуємо відповідно вдосконалити програмований комплексний тип, а поки що запропонуємо рішення, доступне на цьому етапі. Розглянемо фрагмент заголовного файлу для такого випадку.

Лістинг 3.164

```
// Програмований тип комплексних чисел
struct Complex {
   double _re; double _im;
};
```

```
// Множення комплексних чисел
const Complex operator*(const Complex&, const Complex&);
```

А ось реалізація операції множення:

Лістинг 3.165

```
// Множення комплексних чисел
const Complex operator*(const Complex& u, const Complex& v) {
   Complex res={u._re*v._re-u._im*v._im, u._im*v._re+u._re*v._im};
   return res;
}
```

Позбудемося явної одиниці в тексті шаблону:

Лістинг 3.166

```
// Узагальнена функція швидкого піднесення
// чого завгодно до натурального степеня з додатковим
// параметром unit для одиничного елемента
template <typename Type>
const Type power(const Type base, const int exponent, const Type unit) {
   Type y=unit; int n=exponent; Type x=base;
   while(n>0) {
      if(n % 2==0) {x=x*x; n=n/2;}
      else {y=y*x; n=n-1;}
   }
   return y;
}
```

Тепер компілятор зможе згенерувати код для такого обчислення:

Лістинг 3.167

```
// Конкретизація узагальненої фугкції піднесення
// до степеня: випадок комплексних чисел
Complex u={1, 1}; const Complex unit={1, 0};
n=8; cout<<power(u, n, unit)<<endl;
```

Вправа 3.29

Протестуйте функцію піднесення до степеня на дійсних і комплексних числах.

Узагальнені функції, механізм довизначень, задані за замовчуванням параметри та перетворення типів суттєво ускладнюють для компілятора завдання вибору реалізації потрібної функції. Зараз не будемо вдаватися до всіх деталей стандартних механізмів такого вибору. Зауважимо лише, що деякі визначення можуть затіняти інші, тобто робити їх недоступними для компілятора. Наприклад, конкретна функція переставляння цілих чисел затіняє відповідний шаблон. Його можна вивести з тіні, явно конкретизувавши:

```
// Визначення узагальненої функції переставляння          Лістинг 3.168
template <typename Type>
void mySwap(Type& x, Type& y) {
   Type z=x; x=y; y=z;
   return;
}
// Визначення функції переставляння. «Хитрий» обмін значеннями
void mySwap(int& x, int& y) {
   x=x+y; y=x-y; x=x-y;
}
 int main() {
   int m=1, n=2;
// Виклик конкретної функції: вона затінила шаблон
   mySwap(m, n);
// Явна конкретизація шаблону
   mySwap<int>(m, n);
   return 0;
}
```

Інша проблема узагальнених функцій зумовлена складністю перетворення типів. Справді, один і той самий параметр типу, наприклад Type із попереднього прикладу, не може набувати одного значення на місці першого аргументу й іншого – на місці другого. Тому в разі потреби змішування типів під час використання шаблону, що залежить від одного ти́пового параметра, доведеться явно програмувати перетворення типів і визначати додаткову функцію-перетворювач, наприклад:

```
inline double myMax(int x, double y) {
   return myMax<double>(double(x), y);
}
```

Вправа 3.30

Виконайте експеримент і з’ясуйте, як буде оброблено кожен із наведених вище викликів.

```
cout<<myMax(1,2)<<endl; cout<<myMax('a','b')<<endl;      Лістинг 3.169
cout<<myMax(1.F,3.F)<<endl; cout<<myMax(1.,4)<<endl;
cout<<myMax<long>(1,5)<<endl; cout<<myMax<long>(1l,5)<<endl;
cout<<myMax(1,3.14)<<endl; cout<<myMax(1l,4.14)<<endl;
cout<<myMax('a',1)<<endl;
```

Останнє зауваження стосуватиметься місця локалізації визначень узагальнених функцій. Оскільки система програмування не в змозі компілювати їх наперед, до появи конкретного виклику у файлах реалізації, ви-

значення шаблону має бути доступним на момент компіляції кожної його конкретизації. Спосіб, яким це можна зробити, відомий: слід приєднати визначення командою **#include**. Позаяк за попередніми домовленостями приєднують лише заголовні файли, визначення узагальнених функцій розміщують саме там. Розглянемо приклад заголовного файлу Swap.h.

Лістинг 3.170

```
// Swap.h
// Оголошення конкретих функцій
void mySwap(int& x, int& y);
void mySwap(int* x, int* y);
// Оголошення й визначення шаблонів функцій
template <typename T>
void mySwap(T& x, T& y) {
   T z=x; x=y; y=z;
   return;
}
template <typename T>
void mySwap(T* x, T* y) {
   T z=*x; *x=*y; *y=z;
   return;
}
```

Зверніть увагу, що в ньому крім визначень шаблонів є також оголошення звичайних функцій, реалізацію яких, як і належить, розміщено у файлах реалізацій.

3.7. Рекурсія

У процедурному програмуванні для організації повторюваних обчислень використовують дві основні структури керування: ітерацію (цикли) та рекурсію. За нечисленними винятками досі ми розглядали переважно ітеративні програми, повторення дій у яких записано у вигляді циклів. Альтернатива ітеративним програмам – рекурсивні. Крім того, що рекурсія за своєю суттю являє собою потужніший засіб організації обчислень, ніж ітерація, часто вона дає змогу легко переписувати формулювання задачі, подане у вигляді рекурсивних визначень, як рекурсивну програму. Згадаймо рекурсивну функцію обчислення найбільшого спільного дільника за алгоритмом Евкліда:

Лістинг 3.171

```
// Алгоритм Евкліда для обчислення
// найбільшого спільного дільника
int gcd(const int m, const int n) {
   if(m==n) return m;
```

```
   if(m>n) return gcd(m-n, n);
   else return gcd(m, n-m);
}
```

Видно, що це просто його опис

```
gcd(m, m)=m;
gcd(m, n)=gcd(m-n, n), якщо m>n;
gcd(m, n)=gcd(m, n-m), якщо m<n.
```

записаний мовою програмування.

Те саме стосується й функції піднесення до степеня:

```
double slowPower (const double base, const int exponent) { Лістинг 3.172
   if(exponent==0) return 1;
   return base* slowPower(base, exponent-1);
}
```

Це очевидний варіант формули

$$x^n = \begin{cases} 1, \text{ якщо } n = 0, \\ x \cdot x^{n-1}, \text{ якщо } n > 0. \end{cases}$$

Рекурсивна функція може бути узагальненою – наприклад такою:

```
template <typename Type>                                  Лістинг 3.173
Type slowPower(const Type base, const int exponent) {
   if(exponent==0) return 1;
   return base*slowPower(base, exponent-1);
}
```

Вправа 3.31

Доповніть узагальнену функцію `power` з попереднього розділу судженнями про її правильність, побудованими з використанням узагальненої рекурсивної функції `slowPower`.

Як бачимо, у рекурсивних функціях завдяки рекурсивним викликам зникає потреба в багатьох додаткових змінних і водночас спрощується застосування сталих параметрів.

3.7.1. Техніка рекурсивного програмування

Швидке рекурсивне піднесення до степеня трохи складніше. У ньому використано нагромаджувальний параметр `y`, який поступово нагромаджує результати множення. Справді, якщо взяти очевидну залежність

$$base^n = \begin{cases} (base \cdot base)^{\frac{n}{2}}, \text{ якщо } n = 2\text{k}, \\ base \cdot base^{n-1}, \text{ якщо } n = 2\text{k}+1 \end{cases}$$

і помножити її ліву та праву частини на y, одержимо вираз

$$y \cdot base^n = \begin{cases} y \cdot (base \cdot base)^{\frac{n}{2}}, \text{ якщо } n = 2\text{k}, \\ (y \cdot base) \cdot base^{n-1}, \text{ якщо } n = 2\text{k}+1. \end{cases}$$

Його ліва частина має дві цікаві властивості: у разі y=1 вона збігається з $base^n$, а в разі n=0 – зі значенням самої змінної y. Якщо придумати перетворення, що приведуть значення n до нуля, то потрібний результат буде нагромаджено в змінній y. Тепер залишається лише змінювати значення основи *base*, степеня n і нагромаджувального параметра y так, щоб рівність не порушувалася. Вона стає інваріантом функції, яку ми запрограмуємо.

Лістинг 3.174

```
// Узагальнена функція швидкого рекурсивного
// піднесення до степеня
template <typename Type>
const Type& quickPower
(const Type& base, const int exponent, const Type& y=1) {
// Степінь нульовий, відповідь - y
   if(exponent==0) return y;
// Степінь парний: ділимо його навпіл,
// множимо base на себе та робимо рекурсивний виклик
   if(exponent%2==0)
      return quickPower(base*base, exponent/2, y);
// Степінь непарний: зменшуємо його на одиницю,
// відповідно домноживши y на base, і робимо рекурсивний виклик
   return quickPower(base, exponent-1, y*base);
}
```

Утім, із рекурсією пов’язана одна дуже серйозна проблема ефективності. Рекурсію часто легше записати, ніж ітерацію, виходячи з рекурентного співвідношення, що описує поставлену задачу. Однак лише рекурентні співвідношення першого порядку перетворюються на лінійну за часом рекурсію. Якщо взяти рекурентне співвідношення другого порядку, наприклад числа Фібоначчі, то просте переписування його в рекурсію, як у лістингу 3.175, призводить до експоненціальної складності обчислень.

Лістинг 3.175

```
// Дуже поганий спосіб обчислення чисел Фібоначчі
int BadFib(int n) {
```

```
// Так не слід рахувати!!!
   switch(n) {
      case 0: return 0; break;
      case 1: return 1; break;
      default: return BadFib(n-1)+BadFib(n-2);
   }
}
```

Справді, доповнимо функцію BadFib підрахунком кількості рекурсивних викликів. Щоб вивести значення лічильника лише один раз після завершення обчислення, будемо відрізняти зовнішній виклик функції клієнтською програмою від внутрішнього (рекурсивного) виклику нею самою. Для цього використаємо булеву змінну outward, яка набуває значення хиби в усіх внутрішніх викликах:

Лістинг 3.176

```
// Дуже поганий спосіб обчислення чисел Фібоначчі
// з підрахунком рекурсивних викликів
int BadFib(int n) {
   static int counter=1; // Лічильник рекурсивних викликів
// Якщо виклик був зовнішнім, то outward==true
   bool outward=false; if(counter++==1) outward=true; int result;
   switch(n) {
      case 0: result=0; break;
      case 1: result=1; break;
      default: result=BadFib(n-1)+BadFib(n-2);
   }
// У зовнішньому виклику виведемо лічильник
   if(outward) {cout<<counter<<" calls"<<endl; counter=1;};
   return result;
}
```

Викликавши функцію, побачимо, що для обчислення десятого числа Фібоначчі буде зроблено 178 рекурсивних викликів (порівняйте з кількістю повторення циклу в ітеративній версії з лістингу 2.67). Соте число, вочевидь, узагалі не вдасться обчислити таким способом (спробуйте встановити межу розумного очікування результату).

У цьому прикладі використано принципову відмінність між статичною змінною counter й автоматичною змінною outward. Подумайте, для чого на виході з функції змінній counter присвоюється значення 1.

Можна застосувати кращий спосіб рекурсивного обчислення, оскільки неефективність у розглянутому разі – це проблема не рекурсії, а неправильно обраної схеми обчислень. Якщо порівняти два рекурсивні виклики для n-1 і n-2, то стає очевидним, що другий виклик після першого вже зайвий. Справді, n-2-ге число Фібоначчі вже мало бути використане, ко-

ли обчислювалося n-1-ше число. Тоді його варто було б запам'ятати. Обчислення стають тривіальними, якщо функція доставляє значення не одного, а двох сусідніх чисел Фібоначчі. Запропонуємо сигнатуру з трьома параметрами:

```
void fib(int& prev, int& next, const int n);
```

Інваріантом функції fib служить залежність між її параметрами. Якщо перший параметр prev – це якесь число Фібоначчі, то другий параметр next – число, наступне за ним, а значення n – відстань від шуканого числа Фібоначчі до prev. Ознака закінчення обчислень – рівність n одиниці:

Лістинг 3.177

```
// Відшукання чисел Фібоначчі, починаючи з другого;
// n-не число - значення вихідного параметра next.
// У кожному виклику prev і next - два сусідні числа, шукане -
// на відстані n від prev. Якщо n==1, то число next - шукане.
void fib(int& prev, int& next, const int n) {
   int f;
   if(n>=2) {
   // Переходячи до наступного числа Фібоначчі, на одиницю зменшуємо n
      f=next; next+=prev; prev=f; fib(prev, next, n-1);
   }
   return;
}
int Fibonacci(const int n) {  // Інтерфейсна функція до fib
   int prev=0, next =1;
   switch(n) {
      case 0: return prev;
      case 1: return next;
      default:
         fib(prev, next, n); assert (n==1);
         return next;
   }
}
```

Вправа 3.32

Запрограмуйте підрахунок кількості кроків рекурсії в разі відшукання чисел Фібоначчі за допомогою функції fib.

3.7.2. Ефективна рекурсія

Розглянутий щойно спосіб підвищення ефективності обчислень дає мотивацію до уважного аналізу рекурсивних програм із погляду їх ефективності. Однак при цьому не забуватимемо про наявність задач, склад-

ність яких зменшити не вдається. Прикладом може бути задача про знамениті Ханойські вежі, яку не можна розв'язати швидше, ніж за експоненціально зростаючу кількість кроків.

Ханойські вежі. Візьмемо три стержні `a`, `b` та `c`. На першому за спаданням діаметрів нанизано `n` кілець. Потрібно перекласти кільця на інший стержень, використовуючи третій, додатковий, стержень, дотримуючи такого правила: на кожному кроці можна перекладати лише одне кільце так, щоб у жоден момент часу кільце з більшим діаметром не лежало на кільці з меншим.

Цю задачу можна розв'язати на диво просто. Уявімо собі, що нам удалося перекласти `n-1` кільце зі стержня `a` на стержень `c` з використанням як допоміжного стержня `b`. На стержні `a` після цього залишилося одне кільце з найбільшим діаметром. Перекладемо його на стержень `b`. Тепер знову перекладаємо `n-1` кільце, але вже зі стержня `c` на стержень `b`, використовуючи стержень `a` як допоміжний. Відповідну функцію наведено в лістингу 3.178. Корисно проекспериментувати з нею.

Лістинг 3.178

```
void Hanoi(int n, char a, char b, char c) {
   if(n>0) {
      Hanoi(n-1,a,c,b); cout<<"from "<<a<<"to "<<b<<endl;
      Hanoi(n-1,c,b,a);
   }
}
```

Що ж до чисел Фібоначчі, то їх можна обчислювати ще швидше, ніж із лінійною складністю, скориставшись матричним записом

$$\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_{n-1} \\ F_{n-2} \end{pmatrix}, n > 2,$$

звідки випливає, що

$$\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}.$$

Тепер можна використати узагальнену функцію швидкого піднесення до степеня, яка дасть нам можливість обчислювати не всі проміжні значення чисел Фібоначчі. Для цього визначимо структури двовимірних матриць і векторів:

Лістинг 3.179

```
struct Matrix2x2 {
   int _11, _12, _21, _22;
};
```

```
struct Vector2 {
   int _1, _2;
};
```

Означимо операції множення матриць і множення матриці на вектор:

```
// Множення матриць другого порядку                Лістинг 3.180
const Matrix2x2 operator*
   (const Matrix2x2& a, const Matrix2x2& b) {
   // _11 _12
   // _21 _22
   Matrix2x2 res = {a._11*b._11+a._12*b._21,a._11*b._21+a._12*b._22,
                    a._21*b._11+a._22*b._21,a._21*b._21+a._22*b._22};
   return res;
}
// Множення матриці на вектор
const Vector2 operator*
   (const Matrix2x2& a, const Vector2& v) {
   Vector2 res = {a._11*v._1+a._12*v._2, a._21*v._1+a._22*v._2};
   return res;
}
```

Означимо також одиничну матрицю

```
const Matrix2x2 unitMatrix={1,0,0,1};
```

Тепер числа Фібоначчі можна шукати за допомогою функції `quickFib`:

```
// Логарифмічне обчислення чисел Фібоначчі         Лістинг 3.181
int quickFib(int n) {
   Matrix2x2 fibMatrix={1,1,1,0}; Vector2 fibVector={1, 0};
   Vector2 fibRes=quickPower(fibMatrix, n-1, unitMatrix)*fibVector;
   return fibRes._1;
}
```

Вправа 3.33

Порівняйте швидкість обчислення чисел Фібоначчі за ітеративним і рекурсивним способами піднесення до степеня, підрахувавши кількість повторень циклу в ітеративному варіанті та рекурсивних викликів у рекурсивному.

3.8. Функціональна полісемія

Досі можна було помітити певну нерівноправність між іменами змінних і функцій. Змінні в межах, установлених для їхніх типів, набувають довільних значень, тоді як компонувальник іще до початку виконання програми жорстко зв'язує імена функцій з їх тілами.

Виникає питання корисності *функціональної полісемії*, тобто можливості звертатися до різних функцій під одним і тим самим ім'ям. Відповідь на нього позитивна, адже існує багато задач, у яких використовуються не конкретні, а довільні функції з якогось класу. Узяти хоча б задачу обчислення інтеграла функції дійсної змінної на заданому відрізку. Природно, що функція, яка реалізовувала б алгоритм такого обчислення, мала б поряд із числовими параметрами, як-то кінці відрізка інтегрування та точність обчислень, підінтегральну функцію як функціональний параметр.

Певний рух у напрямку функціональної полісемії ми вже вчинили, розглянувши методи поширення функцій і механізм узагальнених функцій. Ідея полісемії полягала в багатозначності імені функції залежно від типів параметрів. Ця полісемія мала статичний характер: навіть у разі узагальнених функції конкретизація шаблону узагальненої функції відбувалася на етапі компіляції програми. За статичної полісемії перехід від одного значення імені функції до іншого пов'язаний із заміною її специфікації. У цьому розумінні статична полісемія – лише зручна економія позначень: програма не зміниться, якщо кожній новій сигнатурі дати інше ім'я функції та користуватися для викликів цими різними іменами.

3.8.1. Указники на функції

Легко уявити собі ситуацію, коли пов'язування імені функції з її тілом відкладається до етапу виконання програми, наприклад тоді, коли сама функція має бути параметром. Тепер, навпаки, фіксується сигнатура, у рамках якої можна вибрати будь-яку належну функцію. Чи можуть самі функції, подібно до елементів даних, бути значеннями змінних? Адже є мови програмування, наприклад LISP, у яких немає відмінності між програмами та даними, й одна функція може приймати як параметр або повертати як результат іншу. Повну відповідь на таке запитання дамо далі, а зараз розглянемо рішення, запропоноване мовою С. Певною мірою можливість розглядати функції як змінні надає інструмент *указників на функції* (pointer to function).

Вивчаючи сигнатури функцій, ми вже звертали увагу на строгу типізацію, яка розрізняє функції за кількістю та типами параметрів і типом результату. Указник на функцію може вказувати лише на функції відповідного для нього типу. Так, визначення

```
// Визначення й ініціалізація указника функції
double(*pf)(double)=0;
```

задає поки що невизначений указник `pf` на довільну дійсну функцію одного дійсного аргументу.

Від імені функції, яке за фіксованої сигнатури жорстко зв'язується зі своїм тілом, указник відрізняється своєю здатністю набувати довільних значень, тоді як кожна функція в межах однієї сигнатури (для конкретного типу параметрів) може мати лише одне визначення. Справді, візьмемо для прикладу сигнатуру **double** quartic(**double**) функції quartic() й одне її визначення:

Лістинг 3.182

```
// Визначення функції
double quartic(double x) {
   return x*x*x*x;
}
```

Спробу дати їй інше визначення

Лістинг 3.183

```
// Альтернативне визначення функції
double sqr(double x) {
   return x*x;
}
double quartic(double x) {
   return sqr(sqr(x));
}
```

буде розпізнано компілятором як помилку, оскільки ім'я функції quartic уже було зв'язане зі своїм значенням – її тілом.

Якщо ж узяти тепер набір різних, але однотипних конкретних функцій, то за допомогою указника можна вибрати будь-яку з них. Указник на функцію може змінювати своє значення за допомогою спрямування на тіло підхожої функції: як визначеної в програмі, так і стандартної:

Лістинг 3.184

```
// Визначення й ініціалізація указника функції
double(*pf)(double)=0;
// Переспрямування указника функції присвоєнням програмованої функції
pf=quartic;
// Переспрямування указника функції присвоєнням стандартної функції
pf=sin;
```

Узагалі кажучи, ці присвоєння потрібно було б записувати як присвоєння адрес:

```
pf=&quartic; pf=&sin;
```

Будь-який компілятор сприймає такий запис як справді коректний, тоді як попередній деякі компілятори визначать як певне спрощення, а інші можуть узагалі заборонити.

Непрямий виклик функції за її указником записують одним із двох рівнозначних способів: pf(a) чи (*pf)(a). Зауважимо, що тепер характер

цього виклику стає динамічним. Щоразу після того, як указник набуває нового значення, зміст виклику змінюється.

Указники на функції використовують здебільшого для передавання функціональних параметрів до інших функцій. Розглянемо, наприклад, функцію розв'язування рівнянь вигляду f(x) = 0 на відрізку [a, b] із заданою точністю ε > 0 методом поділу навпіл. Якщо функція f(x) монотонна та неперервна й набуває на кінцях відрізка значень різних знаків, то рівняння має розв'язок, до того ж один.

Далі наведено текст цієї функції. Зверніть увагу на те, що параметри, зокрема указник на функцію, визначено як сталі. Тому вони можуть набути довільного значення в момент виклику, але це захищає від спроби мимовільно змінити їхні значення всередині функції:

Лістинг 3.185

```
// Розв'язування рівняння f(x)=0
// методом поділу відрізка навпіл
double bisection(const double a, const double b,
      const double eps, double(*const pf)(double)) {
   assert(pf(a)*pf(b)>0);
   double left=a, right=b;
   while((b-left)>eps) {
      if(pf(left)*pf(0.5*(left+right))<0)
//    або: (*pf)(left)*(*pf)(0.5*(left+right)
//    Кореня зліва немає
         right=0.5*(left+right);
      else
         left=0.5*(left+right);
   }
   return left;
}
```

Виклики функції можна записати, наприклад, так:

```
cout<<bisection(-1.0, 2.0, 1.0e-15, sin)<<endl;
cout<<bisection(0, pi, 1.0e-15, cos)<<endl;
```

Визначення типів дають змогу вводити скорочені позначення для типів указників на функції:

Лістинг 3.186

```
// Тип указника дійсної функції дійсного аргументу
typedef double(*PFdouble)(double);
// Тип сталого указника дійсної функції дійсного аргументу
typedef double(*const CPFdouble)(double);
```

Тепер `PFdouble` і `CPFdouble` – повноцінні назви типів. Можливі будь-які з наведених далі варіантів визначення, оголошення чи присвоєння:

```
// Ініціалізація стандартною функцією
PFdouble pf1=sin;
// Ініціалізація невизначеного указника
PFdouble pf2=0;
// Невдале визначення без ініціалізації: не забувайте
// про правила гарного тону
PFdouble pf3;
pf3=cos;   // Присвоєння указнику функції
// Сигнатура функції з функціональним параметром
double bisection (const double a, const double b,
      const double eps, CPFdouble pf);
```

Лістинг 3.187

3.8.2. Використання указників на функції як параметрів

Обчислення визначених інтегралів за методом Сімпсона – дуже повчальний приклад із програмування. Нехай знову $y = f(x)$ – дійсна функція від дійсного аргументу, про яку відомо, що вона інтегровна на відрізку $[a, b]$. Потрібно наближено обчислити інтеграл $S = \int_a^b f(x)dx$ із заданою точністю $\varepsilon > 0$. Сімпсон запропонував робити це за формулою

$$\int_a^b f(x)dx = c_0 \cdot f(a) + c_1 \cdot f\left(\frac{a+b}{2}\right) + c_2 \cdot f(b).$$

Коефіцієнти c_0 , c_1 і c_2 добирають так, щоб обчислене значення було точним для трьох функцій: $y_0 = 1, y_1 = x, y_2 = x^2$. У підсумку для відшукання трьох коефіцієнтів одержимо систему з трьох лінійних рівнянь відносно c_0 , c_1 і c_2 :

$$\begin{cases} c_0 + c_1 + c_2 = b - a, \\ a \cdot c_0 + \dfrac{a+b}{2} \cdot c_1 + b \cdot c_2 = \dfrac{b^2 - a^2}{2}, \\ a^2 \cdot c_0 + \left(\dfrac{a+b}{2}\right)^2 \cdot c_1 + b^2 \cdot c_2 = \dfrac{b^3 - a^3}{3}. \end{cases}$$

Домноживши перше рівняння на $\frac{a+b}{2}$ та віднявши від другого, одержимо $c_0 = c_2$. Знову помноживши друге рівняння на той самий дріб і віднявши від третього, матимемо $c_0 = \frac{1}{3} \cdot \frac{b-a}{3}$. Із першого рівняння те-

пер знайдемо $c_1 = \frac{4}{3} \cdot \frac{b-a}{2}$. Усе разом це дає формулу Сімпсона:

$$s_1 = \frac{1}{3} \cdot \frac{b-a}{2} \left(f(a) + 4\mathrm{f}\left(\frac{a+b}{2}\right) + f(b) \right).$$

Метод Сімпсона полягає в послідовному застосуванні формули Сімпсона до відрізків, одержаних унаслідок поділу відрізка інтегрування на дві рівні частини, і додаванні результатів. Далі кожен із цих відрізків знову ділять навпіл, підсумовують значення, отримані за формулою Сімпсона, і так продовжують доти, доки не досягають потрібної точності. В обчислювальній математиці є оцінка похибки формули Сімпсона. Ми ж будемо керуватися спрощеним практичним критерієм порівняння двох послідовних наближень.

Уведемо такі позначення:

- k – номер кроку послідовного поділу навпіл, тоді кількість відрізків становить $n = 2^k$;
- h_k – довжина кожного відрізка, $h_k = \frac{b-a}{n}$;
- f_i – значення функції в кожному з $n+1$ вузлів інтегрування, $f_i = f(a + ih_k), i = 0, 1, 2, ..., n.$

Остаточно одержимо

$$s_k = \frac{h_k}{3}\left(f_0 + 4f_1 + 2f_2 + 4f_3 + 2f_4 + ... + 2f_{n-2} + 4f_{n-1} + f_n\right).$$

Цю формулу можна запрограмувати напряму:

Лістинг 3.188

```
// Обчислення інтеграла методом Сімпсона
// з повторними обчисленнями значень функції
double slowSimpson(const double a, const double b, const double eps,
                   double(*const f)(double)) {
   int n=2; double h=(b-a)*0.5; double ss=0;
   double s=h*(f(a)+f(a+h)+f(b));
   do {
      ss=s; n*=2; h/=2; s=f(a)+4*f(a+h)+f(b);
      for(int i=2; i<n; i=i+2) {
         s=s+2*f(a+i*h)+4*f(a+(i+1)*h);
      }
      s=s*h;
   } while(fabs(s-ss)>eps);
```

```
   return s/3;
}
```

Однак така програма не дуже вдала, бо методом Сімпсона можна порахувати інтеграл удвічі швидше. Справді, на кожній наступній ітерації обчислюються значення підінтегральної функції в усіх вузлах, зокрема тих, значення яких уже було обчислено на попередніх ітераціях. Ніклаус Вірт [1] запропонував для цього послідовність s_k розділити на суму трьох послідовностей –

$$s_k = s_k^{(1)} + s_k^{(2)} + s_k^{(4)}$$

залежно від коефіцієнта перед f. Послідовності задамо такими співвідношеннями:

$$s_1^{(1)} = \frac{h_1}{3}(f(a) + f(b)),\ s_1^{(2)} = 0,\ s_1^{(4)} = \frac{4\mathrm{h}_1}{3} f\left(\frac{a+b}{2}\right),$$

$$s_k^{(1)} = \frac{1}{2} s_{k-1}^{(1)},\ s_k^{(1)} = \frac{1}{2} s_{k-1}^{(1)},\ s_k^{(2)} = \frac{1}{2} s_{k-1}^{(2)} + \frac{1}{4} s_{k-1}^{(4)},$$

$$s_k^{(4)} = \frac{4\mathrm{h}_k}{3}[f(a + h_k) + f(a + 3\mathrm{h}_k) + \ldots + f(a + (n-1)h_k)],$$

$$k = 2, 3, \ldots .$$

Наведені співвідношення легко переписати в такому вигляді:

Лістинг 3.189

```
// Обчислення інтеграла за методом Сімпсона
// без повторних обчислень значень функції
double Simpson (const double a, const double b, const double eps,
                double(*const f)(double)) {
   int n=2; double h=(b-a)*0.5; double ss=0; double s1=h*(f(a)+f(b));
   double s2=0; double s4=4*h*f(a+h); double s=s1+s2+s4;
   do {
      ss=s; n*=2; h/=2; s1*=0.5; s2=0.5*s2+0.25*s4; s4=0; int i=1;
      do {
         s4=s4+f(a+i*h); i+=2;
      } while(i<=n);
      s4=4*h*s4; s=s1+s2+s4;
   } while(fabs(s-ss)>eps);
   return s/3;
}
```

Вправа 3.34

Підрахуйте кількість обчислених значень підінтегральної функції в обох програмах.

Вправа 3.35

Функцію `Simpson` можна використати для тестування правильності обчислення арифметико-геометричного середнього з попереднього розділу (див. пп. 2.6.2.4. «Цикли»), використовуючи виявлену Карлом Гауссом залежність

$$\frac{1}{M(a,b)} = \frac{2}{\pi}\int_a^{\frac{\pi}{2}} \frac{dx}{\sqrt{a^2\sin^2 x + b^2\cos^2 x}} .$$

Запрограмуйте відповідний тест.

3.8.3. Указники на функції для реалізації меню

Указники на функцію можна збирати в статичні чи динамічні масиви:

Лістинг 3.190

```
double(*pg[n])(double);
PFdouble* pf=new PFdouble [n];
```

Тут `PFdouble`, як і раніше, – тип указника на дійсну функцію від однієї дійсної змінної:

```
typedef double(*PFdouble)(double);
```

Розглянемо простий приклад використання масивів указників для реалізації меню, кожен елемент якого складатиметься з текстового пояснення й імені функції. Щоб користуватися даними типу `string`, слід попередньо командою препроцесора **`#include`** включити відповідний заголовний файл `<string>`.

Елемент меню подамо у вигляді структури:

Лістинг 3.191

```
// Елемент меню
struct MenuItem {
   string _itemName; PFdouble _functionName;
};
```

Елементи меню зберемо в масив, загорнутий у структуру такого вигляду:

Лістинг 3.192

```
// Структура меню
struct Menu {
   static const int _n=4; MenuItem _item[n];
};
```

Зауваження. *Деякі компілятори можуть не розуміти ініціалізації статичного поля структури в її визначенні. У цьому разі потрібно переписати визначення так, як це було зроблено, наприклад, для* `WrappedVector` *(див. лістинг 3.55а).*

Ініціалізація меню матиме такий вигляд:

Лістинг 3.193

```
// Функція ініціалізації меню
void menuInit(Menu& menu) {
// Ініціалізація указників функцій
   menu.item[0]._functionName=cos; menu.item[1]._functionName=sin;
   menu.item[2]._functionName=log; menu.item[3]._functionName=exp;
// Ініціалізація текстів з іменами функцій
   menu.item[0]._itemName="cos"; menu.item[1]._itemName="sin";
   menu.item[2]._itemName="log"; menu.item[3]._itemName="exp";
   return;
}
```

Функція візуалізації меню матиме вигляд

Лістинг 3.194

```
// Функція візуалізації меню
void showNames(const Menu& menu) {
   for(int i=0; i<menu._n; i++)
      cout<<i<<':'<<menu._item[i]._itemName<<endl;
   return;
}
```

Нарешті, наведемо обробник запитів, що обчислюватиме значення однієї з функцій на вибір користувача:

Лістинг 3.195

```
// Обробник меню
void menuHandler(const Menu& menu) {
   showNames(menu); cout<<"Give your choice "; int i; cin>>i;
   double x; cout<<"Give your x "; cin>>x;
   cout<<"The result: ";
   cout<<menu._item[i]._itemName<<'('<<x<<")="<<
         menu._item[i]._functionName(x)<<endl;
   return;
}
```

Вправа 3.36

Запрограмуйте табулювання значень обраної користувачем елементарної функції на вибраному ним відрізку, використовуючи відповідний обробник команд меню.

Вправа 3.37

Запрограмуйте обчислення інтегралів елементарних функцій на вибраних відрізках за методом Сімпсона, використовуючи відповідний обробник команд меню.

Завдання до розділу 3

1. Задана функція

```
double* g(double x){
   return new double(x);
}
```

Які з наведених нижче викликів безпечні? Які небезпечні? Чому?

```
double x=1;
double *p = g(x);
double& r = *g(x);
g(x);
```

2. 32-розрядне машинне слово складається з 4 байтів. Запрограмуйте функції перестановки байтів а) у зворотному порядку; б) в порядку зростання їхніх значень. Коректність функцій перевірте судженням.
3. Байти 32-розрядного машинного слова перепишіть до структури, що складається з чотирьох байтових значень. Коректність функції перевірте судженням.
4. Подвійне 64-розрядне машинне слово складається з 4 півслів. Запрограмуйте функції перестановки півслів а) у зворотному порядку; б) в порядку зростання їхніх значень. Коректність функцій перевірте судженням.
5. Перевірте впорядкованість масиву дійсних чисел.
6. Впорядкуйте масив дійсних чисел за зростанням.
7. Визначте функцію обчислення найбільшого елемента у векторі дійсних чисел. Запропонуйте судження перевірки коректності результату.
8. Запропонуйте три функції для інверсії загорнутого числового вектора, використовуючи як параметр: а) указник; б) відсилку; в) сталу відсилку.
9. Загорнутий вектор `v` складений із машинних слів. Запрограмуйте функції побудови за вектором `v` векторів його лівих та правих півслів. Перевірте коректність функцій судженнями, запрограмувавши обернену функцію з'єднання півслів.
10. Просте стиснення даних за методом RLE полягає у заміні n-кратного повторення символу $\underbrace{aa...a}_{n}$ парою (a,n). Запрограмуйте функцію стиснення довільного рядка символів.
11. Визначте тип `Complex` комплексних чисел. Запропонуйте функції `re` та `im` для обчислення дійсної та уявної частин так, щоб їх можна було б викликати у присвоєннях виду

```
Complex u={1,2}, v={3,4};
re(u) = im(v); im(u) = re(v);
```

12. Якій з наведених нижче сигнатур оператора додавання комплексних чисел `Complex` віддати перевагу? Чому? Реалізуйте найбільш підхожу сигнатуру.

```
Complex operator+(Complex, Complex);
Complex operator+(const Complex, const Complex);
Complex operator+(const Complex&, const Complex&);
const Complex operator+(const Complex&, const Complex&);
const Complex& operator+(const Complex&, const Complex&);
```

13. Визначте для типу `Complex` комплексних чисел операції додавання і множення.
14. Визначте для типу `Complex` комплексних чисел операції присвоєння з додаванням і множенням.
15. Реалізуйте для типу `Complex` комплексних чисел операції додавання та множення через операції присвоєння з додаванням і множенням.
16. Запрограмуйте функцію, яка за вектором комплексних чисел будує вектор їхніх модулів.
17. Визначте операцію додавання `+` двох векторів заданої розмірності.
18. Визначте операцію присвоєння з додаванням `+=` двох векторів заданої розмірності.
19. Запропонуйте сигнатуру і реалізацію операції множення `*` вектора на число.
20. Визначте тип квадратних дійсних матриць розмірності 2×2. Запрограмуйте операції введення, виведення, множення матриці на число, додавання і множення матриць.
21. Визначте тип квадратних дійсних матриць розмірності 2×2. Запрограмуйте операції введення і виведення та функції множення матриці на число, додавання і множення матриць, уникаючи копіювання результату.
22. Визначте операцію множення вектора на матрицю заданої розмірності.
23. Запропонуйте тип даних для многочленів однієї дійсної змінної. Запрограмуйте операцію додавання `+` двох многочленів.
24. Створіть функції `polisum` і `polidif` обчислення суми та різниці многочленів $P_n(x)$ степеню n і $Q_m(x)$ степеню m, уникаючи копіювання одержаних результатів.
25. Запишіть рекурентне співвідношення та складіть функцію для обчислення значення похідної довільного многочлена $P_n(x)$ степеню n в довільній точці x.
26. Складіть функцію для знаходження многочлена, рівного похідній многочлена $P_n(x)$ степеню n.

27. Запрограмуйте обчислення значення раціонального дробу $\dfrac{P_n(x)}{Q_m(x)}$ в довільній точці x, де $P_n(x)$ і $Q_m(x)$ многочлени степенів n і m. Передбачте судження і блок випробувань.
28. Визначте операцію виведення `<<` загорнутого вектора дійсних чисел у вигляді $(x_1, x_2,\ldots, x_n)$.
29. Запропонуйте сигнатуру і реалізацію операції скалярного множення * двох векторів.
30. Порівняти два загорнутих рядка символів на рівність визначивши операції а) `==`; б) `!=`.
31. В довільному рядку символів, складеному із великих і малих літер латинського алфавіту, замініть всі малі літери відповідними їм великими. Запрограмуйте функцію, яка виконує це перетворення, залишаючи формальний параметр незмінним. Запропонуйте декілька можливих сигнатур функції.
32. Впорядкування слів у словнику за алфавітом називають лексикографічним. Порівняйте два загорнутих рядка символів лексикографічно, визначивши операції a) `<`; b) `<=`; c) `>`; d) `>=`.
33. Впорядкуйте лексикографічно словник, заданий масивом символьних рядків.
34. Реалізуйте операції додавання (конкатенації) та суміщеного з присвоєнням додавання загорнутих рядків символів.
35. Запрограмуйте узагальнену операцію`<` лексикографічного порівняння загорнутих рядків символів, застосовну до довільних типів символів: **`char`**, **`w_char`**, розфарбованих символів та інших. (Розфарбований символ – це пара: символ та його колір.)
36. Визначте узагальнену функцію піднесення до натурального степеню, застосовну одночасно як до дійсних, так і до комплексних чисел.
37. Загорнутий вектор `TVector` складений із компонентів довільного типу `T`. Запрограмуйте операції порівняння двох векторів на рівність і нерівність.
38. Визначте операцію виведення вектора довільного числового типу у вигляді $(x_1, x_2,\ldots, x_n)$.
39. Одночленом, залежним від змінної x, називається пара, що складається з коефіцієнта a та степеню n змінної у виразі ax^n. Многочлен задано як список одночленів. Впорядкуйте одночлени за зростанням степенів.
40. Розрідженим многочленом однієї змінної назвемо суму одночленів. Складіть функцію зведення подібних членів розрідженого многочлена.

41. Запрограмуйте функцію обчислення суми двох розріджених многочленів.
42. Список називається циклічним, якщо його перший елемент вважається наступним за останнім. Для циклічного списку цілих чисел запрограмуйте а) функцію додавання елемента до списку; б) функцію впорядкування елементів за зростанням; в) операцію виведення елементів.
43. Нехай $y=f(x)$ – довільна дійсна функція дійсного аргументу. Запрограмуйте табуляцію її значень на довільному відрізку $[a,b]$ з кроком h.
44. Запрограмуйте функцію табуляції результату застосування довільної бінарної арифметичної операції, позначимо її літерою ω, до значень довільних дійсних функцій $f(x)$ і $g(x)$ дійсного аргументу, взятих на відрізку $[a,b]$ із заданим кроком h.
45. Задано масив дійсних чисел. Для довільної дійсної функції дійсного аргументу $y=f(x)$ запрограмуйте функцію побудови масиву пар, що складатимуться зі значень аргументу x та відповідного йому значення функції y.
46. Запрограмуйте функцію, здатну за набором дійсних функцій дійсного аргументу $f_1(x), f_2(x),\ldots, f_n(x)$; набором значень їх аргументів $(x_1, x_2,\ldots,x_n)$ та довільною бінарною арифметичною операцією, позначимо її літерою ω, обчислити результат виконання цієї операції $f_1(x_1)\ \omega\ f_2(x_2)\ \omega\ \ldots\ \omega\ f_n(x_n)$ над значеннями функцій $f_1(x_1), f_2(x_2),\ldots, f_n(x_n)$.
47. Обчисліть значення визначеного інтеграла $\int_a^b x^n e^x dx$ на проміжку $[a,b]$, користуючись рекурентним співвідношенням $\int x^n e^x dx = x^n e^x - n\int x^{n-1} e^x dx$. Функцію обчислення інтегралу перевірте судженням.
48. Запрограмуйте функцію, що виконує довільну бінарну операцію ω над парами, побудованими з елементів загорнутих дійсних векторів $(x_1, x_2,\ldots, x_n)$ і $(y_1, y_2,\ldots, y_n)$. Результат операції подати у вигляді загорнутого вектора.
49. Запрограмуйте функцію, що виконує довільну бінарну операцію ω=, суміщену з присвоєнням, над парами, побудованими з елементів загорнутих дійсних векторів $(x_1, x_2,\ldots, x_n)$ і $(y_1, y_2,\ldots, y_n)$.
50. Знайдіть координати центру ваги системи із заданої кількості матеріальних точок на площині. Маса кожної точки задана як деяка наперед невідома функція від її координат.

4. Об'єктне програмування

У попередньому розділі ми розглянули основні прийоми процедурного програмування, за допомогою яких можна створювати окремі функції. Єдиний спосіб зв'язку між ними – можливість взаємного виклику з відповідним обміном параметрами та результатами (якщо не брати до уваги небажану роботу над спільною пам'яттю). Ще раніше було розглянуто способи структурної організації даних, які дають можливість об'єднувати їх у складні агрегати. Тепер переходимо до агрегатів нового рівня – класів, що об'єднують дані – атрибути разом із методами – функціями для їх обробки.

Звернімо увагу на певну розбіжність термінології, застосовуваної в об'єктно-орієнтованому програмуванні загалом, із технічною термінологією мови C++. Якщо в процедурному програмуванні зазвичай уживають термін *поле* для позначення компонента структури, то в об'єктно-орієнтованому програмуванні заведено говорити про *атрибут класу*, зокрема й структури, як його часткового випадку. Функції, розміщені в класах, в об'єктно-орієнтованому програмуванні називають *методами*. Водночас у мові C++ надають перевагу терміну *члени класу*, які можуть бути як даними, так і функціями. На нашу думку, термін «атрибут» має певну стилістичну перевагу над «даними-членами класу», так само як «метод» – над «функцією-членом класу», і тому ми частіше вживатимемо їх, навіть коли мова йтиме про поняття мови C++.

Домовимося ще про одну пару позначень, надзвичайно важливих для об'єктної парадигми. Її суть полягає в поданні програми як системи взаємодії об'єктів. У рамках послідовного програмування, за які ми не виходитимемо в цьому курсі, у кожний момент часу в програмі, що виконується, виділено один об'єкт, який називають *активним*. Активність об'єкта означає, що на момент розгляду комп'ютер саме виконує один із його методів. Виконання методу може потребувати звернень до інших об'єктів. У момент звернення об'єкт, який *замовив послугу* (у термінах об'єктного програмування – «передав повідомлення», краще сказати – «замовлення», sending a message), стає *клієнтом* (client), а виконавець замовлення – *сервером* (server). Активність клієнта призупиняється, і сервер стає активним об'єктом програми. Зрозуміло, що статус сервера не заважає об'єкту замовити послугу в іншого сервера й перетворитися на його клієнта. На цьому принципі взаємодії ґрунтується так звана *клієнт-серверна архітектура* програмних систем, яку після виходу за рамки послідовного програмування можна застосовувати до паралельних і розподілених програмних систем.

4.1. Принцип інкапсуляції даних і дій

Так само, як літери збирають у слова, а слова – у твори, із команд формують функції, а з функцій – програми. Деякі твори стають шедеврами, але вони не містять нічого, крім речень, складених зі слів, кожне з яких можна знайти в словнику. Слова складаються лише з тридцяти двох літер алфавіту, але осмислені набори літер становлять лише невелику частину всіх можливих комбінацій. І навіть у цій мізерній частині часто трапляються помилки, хоча орфографія вчить нас правил правопису, а граматика – правил побудови речень. Мова стає багатшою, коли в ній ужито особливі мовні конструкції: елегантні звороти, вишукані ідіоми, крилаті вислови, виразні метафори, змістовні гіперболи тощо. У програмуванні такими особливими мовними конструкціями стали ієрархії з класів і об'єктів, які ілюструють найкращі досягнення програмних архітектур, повною мірою втілюючи слова професора Ніклауса Вірта про те, що програми – це взаємозв'язок алгоритмів і даних.

Важко визначити, що складніше – створювати першокласні твори рідною, але неформальною мовою, чи складати програми мовами формальними. У будь-якому разі ми постійно звертатимемо увагу на те, що можна було б назвати «високим стилем» у програмуванні. Він окрім естетичної форми, так само властивої програмуванню, як і високому стилю в літературі, має чисто прагматичні властивості ефективності, надійності, мобільності тощо.

4.1.1. Основні поняття об'єктного програмування

Як і твори, програми складаються з частин: функція – це аналог речення, команда – аналог слова. Збирання команд у функції являє собою найпростіший приклад застосування методу програмування, що має назву інкапсуляції.

Інкапсуляцією (encapsulation) називають способи вбудовування сутностей простішої природи – складових частин або компонентів – в агрегати, що утворюють із них нові, складніші сутності. Механізми вбудовування можуть бути як прозорими (біла скринька), так і прихованими (чорна скринька. Частина агрегату може бути видана своєму клієнтові монолітом, без можливості окремого використання його складових частин, тоді як інша може бути різнорідним набором доступних клієнтові складових. Об'єктний код функції може бути прикладом моноліту, а запис структури мовою програмування – гетерогенного набору її полів.

Продовжуючи аналогію з природною мовою, зауважимо, що буква та слово – різні сутності, але слова можна розглядати як агрегати, утворені

із зібраних разом літер, а слова, своєю чергою, формують речення, які являють собою приклад прозорих агрегатів: усім видно, з яких слів їх складено. Слова, принаймні в сприйнятті їх на слух, не такі зрозумілі: не кожен, хто вимовляє слово, знає, як його правильно записати. Для цього потрібно спочатку принаймні навчитися грамоти.

Відомо, що особливою складністю вирізняється англійський правопис. Сам Бернард Шоу, критикуючи англійську орфографію, писав, що за її правилами слово «*ghoti*» (придумане ним фіктивне слово) можна читати як «*fish*» (риба). Справді, *gh* може звучати як *f* у слові *enough* (досить), *o* – як *i* в слові *women* («жінки» в множині), а *ti* – як *sh* в слові *motion* (рух). Отже, літери можуть певною мірою «ховатись» у слові. У програмуванні приховуванню відведено особливу роль. *Приховування інформації* (information hiding) – іще одне важливе поняття програмування, один із проявів інкапсуляції. Приховування має чітке призначення в програмуванні. Приховують не тому, що люблять секрети чи ребуси, а для того, щоб уберегти компонент від пошкоджень або звільнити клієнта агрегату від опанування деталей, які на його рівні можна вважати неістотними.

Із приховуванням інформації тісно пов'язане поняття *області видимості* (scope), яка задає межі сприймання об'єкта. В інформатиці традиційно розрізняють поняття чорного та прозорого ящиків. Усе, що міститься всередині ящиків, інкапсульовано в них, причому все, що є в чорному, приховано, тому область видимості його компонентів обмежено самим ящиком. Уміст прозорого ящика видимий і за його межами, але не обов'язково доступний: ящик може бути замкненим.

У межах області видимості об'єкта можна говорити про права доступу до нього. Знову вдамося до аналогій. Наприклад, електронні схеми телевізора інкапсульовано та приховано від телеглядача, але не від телемайстра. Щоб увімкнути телевізор і переглянути телепередачу, немає потреби знайомитися з його будовою, а тим паче торкатися прихованих у ньому компонентів. Нічого не змінилося б, якби корпус телевізора було виготовлено з прозорого матеріалу, як у деяких моделях комп'ютерів Macintosh. Його «нутрощі» все одно залишились би недоступними. Телеглядачеві достатньо доступу лише до приладу дистанційного керування, а майстер гарантійного обслуговування має право зняти з телевізора всі пломби, відкрити його корпус і перевірити чи навіть замінити будь-яку з прихованих деталей.

Об'єктне програмування регулює права доступу клієнтів до «нутрощів» сервера: об'єкт інкапсулює всі свої компоненти й приховує частину їх. Важливо, що клас може задавати неоднакові *права доступу* (access

rights) для різних категорій клієнтів своїх об'єктів. Приховане від одних може бути видимим для інших так само, як недоступне для одних може бути доступним для інших. Наприклад, клас календарних дат може показувати всім своїм клієнтам день, місяць і рік поточної дати, але не надавати доступу до них нікому, крім класу системного годинника, який після досягнення 24-ї години міняє день у даті на наступний, а в разі потреби змінює місяць і рік.

Обговоримо ще кілька важливих властивостей об'єктів. Якщо знову взяти, наприклад, телевізор, то можна сказати, що йому властиві *стан* (state) – він може бути вимкненим або увімкненим на певний канал, із використанням або без використання підсилювача низьких частот – і *поведінка* (behavior): він може відтворювати передачі певного телеканалу або записи відеомагнітофона чи DVD-програвача. Нарешті, кожен конкретний телевізор, наприклад той, що стоїть у вас у кімнаті, являє собою *екземпляр* (instance) загального поняття, чи *класу об'єктів* (class, object type) телевізорів. Кожен екземпляр має власну *ідентичність* (identity), яка відрізняє його від усіх інших екземплярів цього класу: хто з нас іще в школі не підписував власного підручника, щоб відрізнити його від такого самого підручника сусіда за партою! Екземпляри класу виникають у певні моменти часу, далі вони функціонують, змінюючи свій стан, але не змінюючи власної ідентичності. Зрештою в якийсь момент часу вони зникають із поля подій.

Розглянуті поняття становлять основу об'єктного програмування, вивчення якого ми зараз розпочинаємо. Об'єктне програмування виникло власне з потреб інженерної практики, тому об'єктні програми складаються із сутностей – об'єктів або екземплярів класів, що являють собою моделі предметів, явищ або процесів реального світу. Поняття стану стає найзагальнішим поняттям, через яке можна спостерігати за результатами функціонування, що залежать від поведінки об'єкта. За наданими правами доступу можна судити про внутрішню будову об'єкта й бачити інкапсульовані в ньому сутності, відкриті для одних клієнтів і приховані від інших, і впливати на них.

4.1.2. Інкапсуляція указників на функції в структурах

Спочатку займемось інкапсуляцією, властивою не лише об'єктному, але й процедурному програмуванню. Справді, у процедурному програмуванні ми вже зустрічалися з інкапсуляцією даних: наприклад, поля були інкапсульованими в структури. Розглянемо вже добре знайомий приклад із точкою площини:

```
// Точка площини                                          Лістинг 4.1
struct Point {
   double _x; double _y;
};
```

У ній інкапсульовано два дійсних числа – абсциса й ордината точки.

Перше запитання, яке варто собі поставити, – чи не можна інкапсулювати в структуру також і дії, наприклад функції? Розглянутий раніше метод функціональної полісемії відразу дає позитивну відповідь на це запитання.

```
// Точка площини з інкапсульованою функцією               Лістинг 4.2
struct QuasiPoint {
   double _x; double _y;
// Інкапсуляція указника на функцію як елемента даних
   double (*pf)(const QuasiPoint&, const QuasiPoint&);
};
```

Тепер, створюючи точку площини, можна разом із наданням значень координатам `_x` та `_y`, скажімо `1` і `0`, надати указнику `pf` як значення певну функцію, наприклад `phiGrad` – обчислення кута між векторами, утвореними кожною з точок і початком координат (у градусах):

```
QuasiPoint u={1, 0, phiGrad};
```

Ніщо не забороняє іншій точці приписати іншу функцію підхожої сигнатури, наприклад функцію `distance` обчислення відстані між двома точками:

```
QuasiPoint v={1, 1, distance}, w={-1, 1, phiGrad};
```

Наведемо для прикладу реалізації згаданих функцій:

```
// Обчислення кута між векторами, утвореними              Лістинг 4.3
// точками u та v й початком координат (у градусах)
const double pi=3.1415926535897932;
double phiGrad(const QuasiPoint& u, const QuasiPoint& v) {
   const double duv=distance(u, v);
   const double du=mod(u); const double dv=mod(v);
   return acos((du*du+dv*dv-duv*duv)/(2*du*dv))*180/pi;
}
// Обчислення відстані між точками u та v
double distance(const QuasiPoint& u, const QuasiPoint& v) {
   return sqrt((u._x-v._x)*(u._x-v._x)+(u._y-v._y)*(u._y-v._y));
}
```

Тут mod – допоміжна функція:

Лістинг 4.4

```
// Обчислення відстані від точки u до початку координат
double mod(const QuasiPoint& u) {
   return sqrt(u._x*u._x+u._y*u._y);
}
```

Ось приклади викликів інкапсульованих функцій:

Лістинг 4.5

```
QuasiPoint v={1, 1, distance},
w={-1, 1, phiGrad}, u={1, 0, phiGrad};
cout<<u.pf(u, v)<<endl; // Кут
cout<<v.pf(u, v)<<endl; // Відстань
cout<<u.pf(u, w)<<endl; // Кут
cout<<v.pf(u, w)<<endl; // Відстань
cout<<w.pf(v, w)<<endl; // Кут
cout<<v.pf(v, w)<<endl; // Відстань
```

Цей програмний код не дуже вдалий, оскільки програми, у яких виклики, здавалося б, однієї й тієї самої функції pf, скажімо u.pf(u, v) та v.pf(u, v), позначають цілком різні речі, важко читати, а їхню логіку неможливо зрозуміти. До того ж у будь-який момент значення указника pf можна легко переспрямувати на іншу функцію:

Лістинг 4.6

```
u.pf=distance; cout<<u.pf(u, v)<<endl;
```

Щоб зрозуміти, добре це чи зле, удамося до міркувань прагматичного характеру. Візьмемо, наприклад, комплексні числа

Лістинг 4.7

```
// Комплексне число
struct Complex {
   double _re; double _im;
};
```

і поміркуємо, які функції можна було б інкапсулювати до цієї структури. Нехай це будуть функції визначення модуля й аргументу комплексного числа, задані такими сигнатурами:

```
double mod(const Complex& u); double arg(const Complex& u);
```

Виникає запитання, чи є сенс вбудовувати в комплексні числа указник, який залежно від обставин скеровуватиме обчислення до одержання чи модуля, або аргументу? Швидше всього, що ні. Варто вбудувати тоді два указники: один на функцію обчислення модуля, інший – аргументу, і

жорстко пов'язати ці функції зі своїми указниками, назвавши їх наприклад, pmod і parg.

Вправа 4.1.

Проаналізуйте, чи можна визначати член структури pf як сталу.

4.1.3. Статичні указники на функції в структурах

Спробуємо зробити навіть трохи більше, ніж запропоновано у вправі 4.1: оголосимо указники на функції як спільні для всіх екземплярів структури.

Отож, наступний крок у напрямку вдосконалення механізмів інкапсуляції функцій до структур полягатиме в зміненні статусу указника на функцію в структурі. Оголосивши його спільним для всіх екземплярів структури за допомогою ключового слова **static**, до того ж іще незмінним, можна усунути помічену раніше ваду:

Лістинг 4.8

```
// Точка площини з незмінною
// спільною інкапсульованою функцією
struct StaticQuasiPoint {
   double _x; double _y;
   static double(*const distance)
      (const StaticQuasiPoint&, const StaticQuasiPoint&);
};
```

Тепер усі екземпляри структури StaticQuasiPoint упродовж усього виконання програми використовуватимуть одну й ту саму функцію. Підправимо відповідно сигнатуру функції обчислення відстані

```
double staticDistance(const StaticQuasiPoint&, const StaticQuasiPoint&);
```

Її реалізація з точністю до типів параметрів і назви самої функції залишається без змін.

Зміниться ініціалізація точок площини: з неї зникне ініціалізація указника:

Лістинг 4.9

```
StaticQuasiPoint v={1, 1}; StaticQuasiPoint w={-1, 1};
```

Указник distance ініціалізується один раз для всіх структур:

Лістинг 4.10

```
double(*const StaticQuasiPoint::distance)
   (const StaticQuasiPoint&, const StaticQuasiPoint&)=staticDistance;
```

Сталий (***const**) указник (*) distance на функцію сигнатури **double**(**const** StaticQuasiPoint&, **const** StaticQuasiPoint&), визначений у структурі StaticQuasiPoint, ініціалізується функцією staticDistance. Таку ініціалізацію досить важко читати. Для її спро-

щення запропонуємо окреме позначення для типу, до якого належить функція `staticDistance`, і назвемо його `CPDistance` (скорочення від «Constant Pointer to Distance»). Зверніть увагу на одну особливість наведеного далі тексту з перехресними посиланнями між структурою `StaticQuasiPoint` і типом `CPDistance`: щоб визначити тип `CPDistance`, спочатку оголошують ужите в ньому ім'я `StaticQuasiPoint`, а вже потім визначають саму структуру:

Лістинг 4.11

```
// Випереджальне оголошення структури
struct StaticQuasiPoint;
// Визначення типу указника на функцію
typedef double(*const CPDistance)
   (const StaticQuasiPoint&, const StaticQuasiPoint&);
// Визначення структури точки площини з незмінною
// спільною інкапсульованою функцією
struct StaticQuasiPoint {
   double _x; double _y; static CPDistance distance;
};
```

Назва типу спростить запис ініціалізації статичного поля:

```
CPDistance StaticQuasiPoint::distance=staticDistance;
```

Його читають так: надати значення `staticDistance` указнику `distance`, який визначений в області імен `StaticQuasiPoint` і має тип `CPDistance`.

Тепер звернення до указника функції з будь-якого об'єкта ініціюватиме виклик однієї й тієї самої функції:

```
cout<<u.distance(u, v)<<endl;
```

Зауваження. *Оскільки указник визначено як статичний, тобто спільний для всіх екземплярів структури, то можливою стає інша форма виклику –*

```
cout<<StaticQuasiPoint::distance(u, v)<<endl;
```

без посилання на об'єкт, але з обов'язковим посиланням на простір імен.

Вправа 4.2

Доповніть структуру комплексних чисел указниками на функції обчислення модуля й аргументу.

4.2. Конструктори та деструктори

Ми ще не вийшли за рамки власне процедурного програмування, залишаючись на рівні чистої мови C, тому, коли треба використовувати C як мову реалізації, розглянутий варіант дає змогу програмувати майже в

об'єктній парадигмі. Правда, незадовільним залишається механізм створення та видалення об'єктів. Нагадаємо його, для чого повернемося до загорнутих векторів і ще раз наведемо визначення структури й оголошення функцій створення та видалення вектора:

Лістинг 4.12

```
// Вектор, доповнений функціями створення та видалення
struct PreWrappedVector {
   static const int _n; double* _v;
};
void construct(PreWrappedVector&);
void destroy(PreWrappedVector&);
```

4.2.1. Функції створення та видалення екземплярів структур

Типовий сценарій використання вектора складається з таких пунктів:

- визначення екземпляра структури – `PreWrappedVector vec`;
- створення масиву для зберігання вектора у пам'яті до початку роботи з ним – `construct(vec)`;
- вивільнення пам'яті, зайнятої вектором, після закінчення роботи з ним – `destroy(vec)`.

Цей сценарій має дві суттєві вади. По-перше, він ненадійний, оскільки не захищає об'єкт від спроби почати роботу з ним у проміжок між визначенням структури та створенням масиву:

Лістинг 4.13

```
PreWrappedVector vec;
cout<<vec._v[100]; // Несанкціоноване звернення до вектора,
                   // який іще не створено
construct(vec);
```

Результат роботи такої команди непередбачуваний.

По-друге, не гарантовано, що масив буде видалено після завершення роботи з ним. Тут можливі різні неприємності. Масив залишиться в пам'яті після ліквідації структури, що забезпечувала доступ до нього, тобто перетвориться на сміття:

Лістинг 4.14

```
// Зовнішній блок: структуру vec ще не визначено
{
// Внутрішній блок: визначення структури vec
   PreWrappedVector vec; construct(vec);
.......................
// Вихід з блоку: автоматичне видалення структури vec
```

```
// Про видалення вектора vec._v забули: він залишився сміттям
// destroy(vec);
}
```

Уявімо собі тепер, що блок з лістингу 4.14 знаходиться у циклі, після входу в кожну ітерацію якого щоразу виділяється новий масив, а його звільнення при закінченні ітерації нема. Така неуважність призведе до створення нової порції сміття після кожного повторення циклу, який, швидше за все, закінчиться аварійно через нестачу пам'яті.

В той же час вихід напрошується сам собою. Він полягає у суміщенні виділення пам'яті під масив зі створенням самої структури та звільненні її під час видалення структури. Виходячи з міркувань щодо своєчасного та правильного створення й видалення об'єктів, зробимо перший важливий крок від мови C до C++ у напрямі об'єктної парадигми. Він полягає в інкапсуляції до структур спеціальних функцій – *конструктора* (constructor) і *деструктора* (destructor), – призначених відповідно для створення та видалення екземплярів структур.

Лістинг 4.15

```
struct WrappedVector {
   static const int _n; double* _v;
   WrappedVector();   // Конструктор створює об'єкт
   ~WrappedVector();  // Деструктор видаляє його
};
```

Головна особливість цих функцій полягає у способі виклику: кожне визначення об'єкта призводить до автоматичного виклику компілятором його конструктора, а вихід з області дії об'єкта – до автоматичного виклику його деструктора:

Лістинг 4.16

```
int main() {
// Визначення екзеипляду структури - це виклик
// конструкторів об'єктів w1 і w2
   WrappedVector w1, w2;
// Тепер робіть із цими об'єктами що завгодно
.................................
// При виході із функції компілятор викличе їхні деструктори
   return 0;
}
```

Друга особливість зазначених функцій – стандартний спосіб іменування: конструктор завжди називають тим самим іменем, що й структуру, а деструктор має таке саме ім'я з тильдою ~ на початку.

Переглянемо об'єктний код до наведеного прикладу:

```
002: WrappedVector w1, w2;                                    Лістинг 4.17
00401EAD lea  ecx,[ebp-10h]
00401EB0 call @ILT+670(WrappedVector::WrappedVector) (004012a3)
00401EB5 mov  dword ptr [ebp-4],0
00401EBC lea  ecx,[ebp-14h]
00401EBF call @ILT+670(WrappedVector::WrappedVector) (004012a3)
003: return 0;
00401EC4 mov  dword ptr [ebp-18h],0
00401ECB lea  ecx,[ebp-14h]
00401ECE call @ILT+225(WrappedVector::~WrappedVector) (004010e6)
00401ED3 mov  dword ptr [ebp-4],0FFFFFFFFh
00401EDA lea  ecx,[ebp-10h]
00401EDD call @ILT+225(WrappedVector::~WrappedVector) (004010e6)
00401EE2 mov  eax,dword ptr [ebp-18h]
```

Код визначення екземплярів структур міститиме два виклики конструктора (рядки `00401EB0` і `00401EBF`), по одному на кожен об'єкт, а код виходу з функції – виклики двох деструкторів (рядки `00401ECE` і `00401EDD`). Згадані виклики на лістингу виділено напівжирним. Зрозуміло, що для виконання такої програми спочатку потрібно реалізувати конструктор і деструктор.

4.2.2. Виклики конструктора та деструктора

Щоб закінчити визначення вектора, доповнимо його визначеннями конструктора та деструктора, які наведено далі в демонстраційному варіанті разом з операторами виведення, які сигналізують про їх виклик.

```
// Загорнутий вектор із конструктором і деструктором       Лістинг 4.18
struct WrappedVector {
   static const int _n; double* _v;
// Конструктор створює об'єкт
   WrappedVector():
// Ініціалізація указника
    _v(new double[_n]) {
// Повідомлення про виклик конструктора
       cout<<"Constructor WrappedVector"<<endl;
// Ініціалізація вектора: краще така, ніж ніякої
       for(int i=0; i<_n; i++)
          _v[i]=1;
       return;
    }
```

```
// Деструктор видаляє об’єкт, у нього параметрів не буває
   ~WrappedVector() {
// Повідомлення про виклик деструктора
      cout<<"Destructor WrappedVector"<<endl;
      delete [] _v; _v=0; // Вивільнення пам'яті
      return;
   }
};
```

Запропонований тут конструктор не має параметрів. За дужками з порожнім списком параметрів у його визначенні йде список ініціалізації членів структури. Оскільки один із цих членів статичний, а тому ініціалізується один раз перед виконанням програми, список ініціалізації в конструкторі складається лише з ініціалізації указника на вектор, розмірність якого гарантовано визначено наперед:

```
_v (new double[_n])
```

Зауваження. *Позаяк у конструкторах дуже часто вживають операцію виділення пам’яті, не варто забувати про обробку можливих аварійних ситуацій, про яку сигналізуватиме подія* `bad_alloc` *(див. підрозділ 2.7.2). Для цього виклики таких конструкторів розміщують в області, захищеній блоком випробувань.*

Лістинг 4.19

```
int main() {
   try {
      WrappedVector w1, w2; // Це виклик конструкторів об’єктів w1 і w2
// Тепер робіть із цими об'єктами що завгодно,
// але в разі виходу з функції компілятор викличе їхні деструктори
   }
   catch (bad_alloc) {
// Тут розміщують обробку аварійної ситуації
      cerr<<"Out of memory"<<endl;
   }
   return 0;
}
```

У лістингу 4.18 визначення конструктора та деструктора розміщено безпосередньо у визначенні структури. До проблеми відокремлення реалізації від визначення типів даних ми ще повернемося пізніше, перейшовши від структур до класів, але вже тепер зазначимо, що визначення конструктора та деструктора можна перемістити до файлу реалізації. Тоді перед їхніми іменами потрібно зазначати структуру, до якої вони належать, за допомогою оператора розв’язування області дії:

```
// Реалізація конструктора загорнутих векторів          Лістинг 4.20
// Запис у такому форматі: Простір імен::Конструктор()
// параметрів поки що немає
WrappedVector::WrappedVector(): _v(new double[_n]) {
// Ініціалізація: краще така, ніж ніякої
   for(int i=0; i<_n; i++) _v[i]=0;
   return;
}
```

Запишемо деструктор для цієї структури:

```
// Реалізація деструктора загорнутих векторів           Лістинг 4.21
// Простір імен::Деструктор(), параметрів не буває
WrappedVector::~WrappedVector() {
   delete [] _v; // Вивільнення пам'яті
   _v=0;
   return;
}
```

Важлива особливість запису конструктора та деструктора полягає в тому, що вони не повертають результату в командах виходу з них, але специфікатор **void**, як це заведено для звичайних функцій, у їхніх заголовках не пишуть.

Крім наведених раніше випадків визначення об'єктів, конструктор і деструктор викликаються автоматично в разі виконання відповідно команд **new** та **delete**, застосованих до указників об'єктів програмованих типів:

```
// Створення і видалення об'єкта за допомогою указника   Лістинг 4.22
int main() {
   WrappedVector* pw;
   pw=new WrappedVector; // Це неявний виклик конструктора об'єкта *pw
   delete pw;            // Це неявний виклик деструктора об'єкта *pw
   return 0;
}
```

У попередньому розділі ми вже обговорювали відмінності між мовами C та C++. Одна з них виявляється в особливостях роботи з пам'яттю. Команди `malloc/calloc` і `free` лише резервують і звільняють пам'ять, тоді як **new** та **delete** синхронізовані з конструкторами та деструкторами. Ось чому заборонено змішувати операції розподілу пам'яті мов C та C++.

Якщо виділити пам'ять для `WrappedVector` командами `malloc/calloc`, а звільнити – командою **delete**, то невідомо, хто має виділити пам'ять для

вектора _v, оскільки це функція конструктора, а він викликається командою **new**, а не malloc/calloc. Окрім того, у разі застосування команди malloc/calloc незрозумілою стає функція деструктора ~WrappedVector(), який робитиме спробу (швидше за все, невдалу) видалити вектор _v. Навпаки, виділення пам'яті командою **new** та її звільнення за допомогою команди **free** видалить об'єкт без виконання його деструктора, який, зі свого боку, мав би вивільнити від вектора _v виділену в конструкторі пам'ять.

Виникає запитання: звідки структура PreWrappedVector, яку ми розглянули раніше, візьме власні конструктор і деструктор? У тому, є вони чи їх немає, і полягає головна відмінність між реалізацією структур у мовах С та С++. Якщо йдеться про мову С, то в ній справжніх конструкторів і деструкторів не буває, хіба що програміст сам використовує відповідні ерзац-функції, запрограмовані за зразком функцій construct() та destroy(), як це було зроблено у підрозділі 4.2.1, і сам несе відповідальність за їх своєчасний виклик. У мові ж С++ структура має конструктор і деструктор, навіть якщо їх не записано у її визначенні. В останньому випадку компілятор сам доповнює визначення потрібною інформацією. Наприклад, можна вважати, що структуру PreWrappedVector він сприймає так, ніби в ній записано визначення невидимих конструктора та деструктора з порожніми тілами:

Лістинг 4.23

```
// Згенеровані за умовчанням конструктор і деструктор
struct PreWrappedVector {
   static const int _n; double* _v;
   PreWrappedVector() {}   // Конструктор за умовчанням
   ~PreWrappedVector() {}  // Деструктор за умовчанням
}
```

Такі додані компілятором функції називають відповідно *конструктором* і *деструктором за умовчанням* (default constructor/destructor). Завдяки такому рішенню тексти мовами С та С++ сумісні (структури мови С буде коректно відкомпільовано компілятором С++). Однак це породжує багато інших проблем, які ми обговоримо далі. Поки що зауважимо, що власний порожній конструктор або деструктор кращий від компіляторового хоча б тим, що розробник бачить його, може зайти в нього за допомогою налагоджувача, а також помістити в ньому команди виведення, захистивши їх за потреби за допомогою директиви NDEBUG (порівняйте з лістингом 2.66):

Лістинг 4.24

```
// Власні конструктор і деструктор
// Порожній неправильний конструктор – аналог конструктора за умовчанням
```

```
PreWrappedVector::PreWrappedVector() {
   #ifndef NDEBUG cout<<"PreWrappedVector constructor is called"<<endl;
   #endif
}
// Порожній неправильний деструктор – аналог деструктора за умовчанням
PreWrappedVector::~PreWrappedVector() {
   #ifndef NDEBUG cout<<"PreWrappedVector destructor is called"<<endl;
   #endif
}
```

Розглянувши тепер інший приклад, побачимо, що конструктори можуть мати – і зазвичай мають – параметри:

Лістинг 4.25

```
// Точка площини з конструктором і деструктором
struct Point {
   double _x; double _y;
   Point(const double x=0, const double y=0): _x(x), _y(y) {}
   ~Point();
};
```

Цим параметрам зручно надати значення за умовчанням – тоді буде збережено використовуваний у мові C спосіб визначення екземплярів структур. Це знову ж данина сумісності мов C та C++. Повний запис визначення об'єкта за допомогою конструктора має вигляд

```
Point a=Point(1, 2); // Повний виклик конструктора
```

Замість цього зазвичай пишуть скорочено

```
Point a(1, 2); // Скорочений виклик конструктора
```

Якщо другий параметр узято за умовчанням, то замість

```
Point b=Point(1, 0); // Повний виклик конструктора
// Або повний виклик конструктора, другий параметр узято за умовчанням
Point b=Point(1);
```

пишуть просто

```
// Скорочений виклик конструктора, другий параметр узято за умовчанням
Point b(1);
```

Відповідно замість одного з таких варіантів

```
Point c=Point(0, 0); // Повний виклик конструктора
// Повний виклик конструктора, обидва параметри взято за умовчанням
Point c=Point();
```

записують

```
// Скорочений виклик конструктора, обидва параметри взято за умовчанням
Point c;
```

Саме так ми раніше записували визначення екземплярів структур, наприклад QuasiPoint, які, як тепер знаємо, ініціалізуються конструктором за умовчанням. До речі, стає зрозуміло, чому компілятор мови С++ дозволяв визначати комплексні константи без ініціалізації:

```
const Point zero; // Якою буде константа zero? – нульовою
```

Це була скорочена ініціалізація визначеного власного конструктора

```
// Власному конструктору буде передано нульові значення параметрів
const Point zero = Point(0,0);
```

Якщо ж власного конструктора не було, то компілятор викликав конструктор за умовчанням, тобто фактично працював код

```
QuasiPoint :: QuasiPoint() {}  // Конструктор за умовчанням
const QuasiPoint foo;          // буде викликано без параметрів
cout<<foo._x<<','<<foo._y<<endl;
```

виклику конструктора без параметрів із порожнім тілом. У другому випадку константа може бути заповнена сміттям, але це вже проблема розробника програми, а не компілятора. *Не пишіть таких програм!*

Проте, віддаючи належне компілятору, зазначимо, що в таких випадках він генерує попередження

```
warning C4269: 'foo' : 'const' automatic data initialized with compiler
generated default constructor produces unreliable results
```

про непередбачуваний характер результату.

4.2.3. Реалізація конструкторів

Розглянемо способи реалізації конструктора. Типова помилка в конструкторах пов'язана з нерозумінням відмінностей між ініціалізацією та присвоєнням, про що вже йшлось у підрозділі 2.3.2. Часто, особливо в програмах початківців, трапляються конструктори, у реалізації яких поля набувають значень унаслідок присвоєння, котре за своєю суттю мало б бути ініціалізацією (пам'ятаймо, що визначення навіть звичайної змінної без ініціалізації рівнозначне її ініціалізації сміттям). Таку програму вважатимемо невдалою хоча б тому, що конструктор починає виконуватися, оперуючи засміченими атрибутами, що видно з лістингу 4.26:

```
// Невдалий конструктор: атрибути
// набувають значень унаслідок присвоєння
Point::Point(const double x, const double y) {
// Ця частина конструктора виконується до ініціалізації атрибутів:
   cout<< _x<<’,’<< _y<<endl; _x=x; _y=y;
// Ця частина конструктора виконується після ініціалізації атрибутів:
   cout<< _x<<’,’<< _y<<endl;
   return;
}
```

Лістинг 4.26

Водночас у разі належної ініціалізації атрибутів у заголовку конструктора вони гарантовано набудуть наданих їм значень ще до входу в нього, а тому звертання до них будуть коректними упродовж виконання всього тіла конструктора.

З ініціалізацією атрибутів у структурах пов’язане ще одне непорозуміння. Початківцям в об’єктному програмуванні важко утриматися від бажання записати визначення структури з одночасною ініціалізацією її членів – щось на зразок такого:

```
struct Point {
   double _x=0; double _y=0;
};
```

Лістинг 4.27

Однак це не спрацьовує, оскільки визначити структуру – не те саме, що створити об’єкт. Можна було б спробувати проінтерпретувати такий текст як попередній запит на ініціалізацію, але ж під час створення екземпляра структури діють інші правила неявної ініціалізації – з використанням значень заданих за умовчанням параметрів. Тому в конструкторах використовують *списки ініціалізації* (initialization list) членів структур фактичними значеннями параметрів. Важливо, що ініціалізація відбувається в момент виділення пам’яті під екземпляр структури, тобто перед виконанням першої команди конструктора поля вже ініціалізовано. Правильна реалізація конструктора має приблизно такий вигляд:

```
// Список ініціалізації має вигляд «: атрибут(значення),…»
Point::Point(const double x, const double y): _x(x), _y(y) {
// Важливо! Починаючи з першої команди, можна використовувати
// значення членів структури. У демонстраційній версії конструктора
// вже можна показати члени структури
   cout<<_x<<', '<<_y;
   return;
}
```

Лістинг 4.28

Якщо конструктор такий простий, як для точки площини, то його ще краще розмістити безпосередньо у визначенні структури, що дасть компілятору можливість реалізувати його вбудованою функцією `inline`. Позаяк динамічна пам'ять не виділяється, то деструктор залишається порожнім:

Лістинг 4.29

```
// Конструктор і деструктор inline
struct Point {
   double _x; double _y;
   Point(const double x=0, const double y=0): _x(x), _y(y) {}
   ~Point() {}
};
```

Виникає питання стосовно виділення динамічної пам'яті. Чи варто це краще робити в тілі конструктора, як на лістингу 4.30?

Лістинг 4.30

```
// Конструктор без ініціалізації
WrappedVector::WrappedVector() {
   _v=new double[_n];
// Ініціалізація: краще така, ніж ніякої
   for(int i=0; i<_n; i++) _v[i]=0;
   return;
}
```

Чи краще надати указнику значення в списку ініціалізації, як на лістингу 4.31? Очевидно друге рішення краще, тим більше що атрибут **`_n`** статичний і його значення на момент виклику конструктора вже наявне. Випадок залежності ініціалізації одного атрибуту від значення іншого нестатичного атрибуту буде розглянуто окремо.

Лістинг 4.31

```
// Конструктор з ініціалізацією указника
WrappedVector::WrappedVector(): _v(new double[_n]) {
// Ініціалізація: краще така, ніж ніякої
   for(int i=0; i<_n; i++)
      _v[i]=0;
   return;
}
```

Вправа 4.3

Розмістіть виклик конструктора структури `WrappedVector` до блоку випробувань **`try`** і промоделюйте виникнення аварійної ситуації нестачі пам'яті.

У навчальних програмах, щоб можна було трасувати процеси створення та видалення об'єктів, конструктори та деструктори зручно допов-

нювати керованим препроцесором виведення. Із цією метою для кожного нового типу об'єктів зазвичай визначають операцію виведення:

Лістинг 4.32

```
// Оператор виведення точок площини
ostream& operator<<(ostream& os, const Point& u) {
   os<<'('<<u._x<<', '<<u._y<<')';
   return os;
}
```

Щоб викликати цей оператор у конструкторі точки площини, слід передати йому саму точку як параметр. Домовилися використовувати ключове слово **this**, яке розуміють як указник саме на той екземпляр структури, у якому його вжито (детальніше про ключове слово **this** йтиметься у підрозділі 4.5). Нагадаємо також, що NDEBUG – це стандартна константа препроцесора, яка вимикає режим налагодження:

Лістинг 4.33

```
// Конструктор із керованим виведенням
Point::Point(const double x, const double y): _x(x), _y(y) {
#ifndef NDEBUG
   cout<<*this<<endl;
#endif
   return;
}
// Деструктор із керованим виведенням
Point::~Point() {
#ifndef NDEBUG
   cout<<"Object "<<*this<<" deleted"<<endl;
#endif
   return;
}
```

Зауваження про масиви. *Визначення масивів об'єктів зумовлює кратний виклик конструктора під час створення масиву, а в разі його видалення – і деструктора. Це ще одна особливість керування пам'яттю в мові С++. Так, визначення*

```
WrappedVector wrevec[10]; // Створення одного масиву з десяти векторів
```

10 разів викликає конструктор загорнутих векторів, завдяки чому всі вони будуть коректно створені. На жаль, цим конструкторам не можна передавати параметри. Якщо в класі, масив екземплярів якого будують, немає конструктора без параметрів або зі значеннями всіх заданих за умовчанням параметрів, то спроба визначити масив призведе до синтаксичної помилки. Вихід з блоку, у якому визначено масив `wrevec[10]`*, завершиться десятиразовим викликом деструктора, унаслідок чого буде вивільнено замовлену конструктором динамічну пам'ять для кожного з*

*десяти векторів. Те саме стосується операцій виділення та вивільнення пам'яті для динамічних масивів за допомогою операцій **new** [] та **delete** []:*

Лістинг 4.34

```
// Виклик десяти конструкторів
WrappedVector * pwrevec =new WrappedVector [10];
delete [] pwrevec; // Виклик десяти деструкторів
```

В узгодженості апарату створення та видалення об'єктів за допомогою конструкторів і деструкторів з операціями керування пам'яттю й полягає головна перевага керування масивами в мові С++ порівняно із С.

Списки ініціалізації в конструкторах – єдиний спосіб визначення сталих членів структури. Для прикладу розглянемо структуру, що містить особисту інформацію про певну особу. Там мають бути ім'я та прізвище, може бути інформація про освіту, рід занять тощо. Узагалі-то всі названі відомості, навіть ім'я та прізвище, можуть бути змінені, але не може змінитися дата народження чи, скажімо, ідентифікаційний код. Ці поля мають бути сталими, і їх інакше, ніж списком ініціалізації, не задаси.

Лістинг 4.35

```
struct Person {
   int _len; char* _name;
   const int _dayOfBirth; const int _monthOfBirth;
   const int _yearOfBirth; const int _personalID;
   Person(const int len, const char[] n, const int d,
      const int m, const int y, const int id);
   ~Person();
};
```

Природно, що конструктор має відповідати не лише за ініціалізацію членів, але й за виділення пам'яті під масиви.

Лістинг 4.36

```
// Конструктор і деструктор осіб
Person::Person(const int len, const char[] n, const int d, const int m,
   const int y, const int id): _len(len), _name(new char[_len +1]),
// Ініціалізація сталих членів структури
   _dayOfBirth(d), _monthOfBirth(m), _yearOfBirth(y), _personalID(id) {
   for(int i=0; i<_len; i++)
      _name[i]=name[i];
   _name[_len]='\0'; cout<<"A person "<<_name<<" was created"<<endl;
}
Person::~Person() {
   cout<<"A person "<<_name<<" was deleted"<<endl; delete [] _name;
}
```

Тепер, мабуть, найважливіше. *Насправді виконання списку ініціалізації складається з викликів конструкторів.* Цього досі не було помітно, оскільки члени структур мали стандартні типи. Розглянемо структуру, поля якої самі являють собою структури:

Лістинг 4.37

```
struct Point;
// Клас трикутників, заданих своїми вершинами
struct Triangle {
   Point _a, _b, _c;
   Triangle(const double x1=0, const double y1=0,
      const double x2=1, const double y2=0,
      const double x3=0, const double y3=1);
   ~Triangle();
   double area();
};
```

Тепер запишемо його конструктор:

Лістинг 4.38

```
// Конструктор трикутника за координатами вершин
Triangle::Triangle(const double x1, const double y1,
      const double x2, const double y2, const double x3, const double y3):
// Список ініціалізації складається з викликів конструкторів точок
      _a(x1, y1), _b(x2, y2), _c(x3, y3) {
   return;
}
```

Ініціалізація поля `_a(x1, y1)` типу `Point` – це виклик конструктора відповідного типу. Те саме стосується всіх інших членів.

Вправа 4.4

Доповніть конструктор трикутника перевіркою нерівності трикутника.

Вправа 4.5

Визначте структуру трикутників, наділивши її конструктором трикутника за трьома вершинами, заданими за допомогою екземплярів структури точок.

4.3. Права доступу, селектори та модифікатори

Два попередні приклади ілюструють іще одну важливу проблему, пов'язану з правами доступу до членів структури. Важливо відрізняти можливість бачити структуру, навіть її складові, від прав доступу до них. З яких частин складається структура, бачить кожен, кому доступний за-

головний файл з її визначенням. Поки що кожен, хто бачить структуру, має також доступ до її «нутрощів». Це добре чи зле?

Наприклад, можливість безконтрольного доступу до координат точки площини, швидше за все, не призведе до великої біди, оскільки значенням координати може бути довільне дійсне число. Якщо ж дозволити неконтрольований доступ до вершин трикутника, то де гарантія, що через звичайну некомпетентність або неуважність координати якоїсь однієї чи кількох вершин не буде модифіковано настільки, що трикутник урешті-решт виродиться до відрізка чи точки? Якби йшлося, скажімо, про кадрову систему, то доступ до вмісту особової картки `Person` залежав би від повноважень працівника, зайнятого обробкою даних. Поки що не вдаючись до деталей, лише зауважимо, що є проблема регламентації прав доступу до полів структури з боку різних її клієнтів.

Візьмемо, наприклад, трикутник (лістинг 4.9) із заданими за умовчанням вершинами та замінимо одну з них іншою, виконавши присвоєння для вершини `_c`:

Лістинг 4.39

```
Triangle ABC; ABC._c=Point(-1, 0);
```

Результат присвоєння формально залишається екземпляром структури, що як і раніше, складатиметься з трьох членів, але перестане бути трикутником у математичному розумінні цього слова, оскільки тепер точки `_a`, `_b`, `_c` лежатимуть на одній прямій. Висновок дуже простий: не завжди можна дозволяти безконтрольно змінювати значення частин структур. Тому згідно з правами доступу структуру поділяють на дві частини – *відкриту* (public) і *закриту* (private).

Перше правило, яким будемо користуватися, полягає в тому, що дані в структурі слід закривати від доступу ззовні, а конструктор і деструктор – відкривати:

Лістинг 4.40

```
// Вершини закриті, конструктор і деструктор відкриті
struct Triangle {
private:        // Закрита частина
   Point _a, _b, _c;
public:         // Відкрита частина
   Triangle(const double x1=0, const double y1=0,
      const double x2=1, const double y2=0,
      const double x3=0, const double y3=1);
   ~Triangle();
};
```

Однак тепер трикутник став «річчю в собі» – ніхто не може ні змінити, ні навіть побачити його вершини. Тому у відкритій частині структури мають бути функції, що забезпечують доступ до вершин трикутника: *селектор* (selector) для читання та *модифікатор* (modifier) для змінення значень вершин:

Лістинг 4.41

```
// Вершини закриті, конструктор,деструктор,
// селектори та модифікатори відкриті
struct Triangle {
private: // Закрита частина
   Point _a, _b, _c;
public: // Відкрита частина
   Triangle(const Point& a=Point(0, 0),
      const Point& b=Point(1, 0), const Point& c=Point(0, 1));
   ~Triangle();
// Селектори
   const Point& get_a() const; const Point& get_b() const;
   const Point& get_c() const;
// Модифікатори
   void set_a(const Point&); void set_b(const Point&);
   void set_c(const Point&);
};
```

Селектор дає змогу прочитати вміст закритого поля:

Лістинг 4.42

```
// Реалізація селектора вершини _a
const Point& Triangle::get_a() const {return _a;}
```

У його сигнатурі є додаткова специфікація **const**, яка повідомляє компілятору, що ця функція не має права змінювати структуру, а тому її можна використовувати як для сталих, так і для змінних об'єктів:

Лістинг 4.43

```
const Triangle ABC; ABC.get_a();
```

Про спробу змінити об'єкт у сталій функції компілятор повідомить як про синтаксичну помилку. Ось приклад, який це ілюструє:

Лістинг 4.44

```
// Помилкова реалізація селектора вершини _a
const Point& Triangle::set_a() const {
   return _a=_a; // Спроба обчислити лівобічне значення в сталому об'єкті
}
```

Зміну значення вершини за допомогою модифікатора можна доповнити запобіжником, який перевірятиме коректність такої дії:

```
// Реалізація модифікатора вершини _a                    Лістинг 4.45
void Triangle::set_a(const Point& a) {
   Point _old_a=_a; _a=a;
   if(area()==0) _a=_old_a;
   return;
}
```

Модифікатор не може бути сталою функцією. Про спробу змінити об'єкт у сталій функції компілятор повідомить як про синтаксичну помилку.

```
// Помилкова реалізація модифікатора вершини _a          Лістинг 4.46
void Triangle::set_a(const Point& a) const {
   Point _old_a=_a; _a=a;
   if(area()==0) _a=_old_a;
   return;
}
```

До розгляду інкапсульованих сталих функцій повернемося пізніше в підрозділі 4.7.

Модифікаторів може бути кілька, якщо вони різняться типами параметрів. Так, до класу трикутників можна було б додати ще один модифікатор:

```
void Triangle::set_a(const double x, const double y) {  Лістинг 4.47
   Point _old_a=_a; _a._x=x; _a._y=y;
   if(area()==0) _a=_old_a;
   return;
}
```

В обох прикладах `area()` – це функція обчислення площі трикутника. Її можна додати до структури, відповідно розширивши визначення трикутника:

```
// Трикутник, доповнений обчисленням площі               Лістинг 4.48
struct Triangle {
private:             // Закрита частина
   Point _a, _b, _c;
public:              // Відкрита частина
   Triangle(const Point& a=Point(0, 0),
      const Point& b=Point(1, 0), const Point& c=Point(0, 1));
   ~Triangle();
// Селектори
   const Point& get_a() const; const Point& get_b() const;
   const Point& get_c() const;
```

```
// Модифікатори за точками
   void set_a(const Point&); void set_b(const Point&);
   void set_c(const Point&);
// Модифікатори за координатами
   void set_a(const double x, const double y);
   void set_b(const double x, const double y);
   void set_c(const double x, const double y);
// Площа трикутника
   double area() const;
};
```

Зауваження. *Функція обчислення площі трикутника ґрунтується на відомому з лінійної алгебри факті, що площа* S *трикутника, утвореного вершинами, що мають координати* $(x_1, y_1), (x_2, y_2), (x_3, y_3)$, *дорівнює половині модуля визначника* $S = \frac{1}{2}|\Delta|$, *утвореного з координат цих вершин:*

$$\Delta = \begin{vmatrix} 1 & 1 & 1 \\ x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \end{vmatrix}.$$

Очевидно, що обчислення площі, як і селектор, – стала функція, а модифікатор, звісно ж, ні.

Запишемо реалізацію функції обчислення площі трикутника:

Лістинг 4.49

```
// Обчислення площі за допомогою функції - члена структури
double Triangle::area() const {
   return 0.5*fabs((_a._x*_b._y-_a._y*_b._x)-
      (_a._x*_c._y -_a._y*_c._x)+(_b._x*_c._y-_b._y*_c._x));
}
```

Її можна визначити і як окрему функцію з такою сигнатурою:

```
double area (const Triangle&);
```

Вправа 4.6

Реалізуйте функцію обчислення площі трикутника, зовнішню щодо його структури, використовуючи вищенаведену сигнатуру. Візьміть до уваги обмеження доступу до його вершин.

Якщо закриття доступу до вершин трикутника було виправдано захистом їх від некоректної модифікації, то чи варто залишати відкритим доступ до членів інших структур, де такої небезпеки немає?

Розглянемо координати точок площини. Аргумент на користь залишити їх відкритими – простота доступу, аргументом проти може бути складність аналізу текстів програм. Уявімо собі ситуацію пошуку помилки в програмі, що пов'язана із занесенням некоректного значення координати до однієї з точок. За відкритого доступу до координат не залишається нічого іншого, як читати текст програми й аналізувати підозрілі місця. У разі ж доступу через модифікатор достатньо помістити в ньому додаткову перевірку чи контрольну точку налагоджувача.

Друге міркування на користь закритого доступу – технологічне. Трохи раніше було наведено функцію обчислення площі трикутника, у якій використано відкритий доступ до координат його вершин. Розглянемо тепер процес роботи над створенням програм у часовому вимірі. Припустімо, на певному етапі буде прийнято рішення змінити доступ до певної частини структури. Якщо зміна полягатиме в тому, що частина закритих членів структури стануть відкритими, це не вплине на створені на цей час програмні компоненти. Якщо ж закрити доступ до відкритих раніше членів структури, то доведеться змінювати всі компоненти, які користувалися цим доступом. Тому віддаємо перевагу розміщенню членів структури в її закритій частині, пам'ятаючи, що винесення до відкритої частини – це безпечна операція, яка проте завжди має бути обґрунтованою.

Тому з технологічної точки зору краще визначити точку площини з закритими координатами, доповнивши їх селекторами і модифікаторами:

Лістинг 4.50

```
// Точки площини із селектором і модифікатором
struct Point {
private:
   double _x; double _y;
public:
   Point(const double x=0, const double y=0);
   ~Point() {}
// Селектори: const гарантує незмінність об'єкта
   const double& getX() const {return _x;}
   const double& getY() const {return _y;}
// Модифікатори:
   const Point& setX(const double x){_x=x; return *this;}
   const Point& setY(const double y){_y=y; return *this;}
};
```

Зверніть увагу на типи результатів селекторів і модифікаторів. Селектор повертає сталу відсилку до атрибуту, що зчитується, без додаткового його копіювання. Модифікатор повертає сталу відсилку до зміненого

об'єкту, знову ж таки уникаючи копіювання. Тому змінений об'єкт може бути одразу використано в лівосторонній позиції, наприклад, так:

```
cout<<a.setX(10).getX()<<endl;
```

Альтернативою могла б бути функція, що не повертає значення:

```
void setX(const double x){_x=x;}
```

Тоді використання зміненого об'єкта вимагає додаткового звертання до нього:

```
a.setX(10); cout<<a.getX()<<endl;
```

Через те, що не стало прямого доступу до координат точок площини, функції обчислення площі трикутника доведеться оперувати селекторами координат його вершин. Проте непрямий доступ до атрибутів за допомогою селекторів і модифікаторів навіть в тих випадках, коли існує можливість прямого звертання до них, завжди кращий, оскільки залишає можливість контролю за використанням атрибутів і спробами заміни їхніх значень.

Лістинг 4.51

```
// Обчислення площі за допомогою функції - члена структури
double Triangle::area() const {
   return 0.5*fabs((_a.getX()*_b.getY()-_a.getY()*_b.getX())-
      (_a.getX()*_c.getY()-_a.getY()*_c.getX())+
      (_b.getX()*_c.getY()-_b.getY()*_c.getX()));
}
```

Вправа 4.7

Створіть функцію обчислення периметра трикутника, використовуючи селектори координат його вершин.

Розглянемо один своєрідний прийом суміщення селектора з модифікатором. Назвемо його селектором-модифікатором.

Лістинг 4.52

```
// Точки площини із селектором-модифікатором
struct Point {
private:
   double _x; double _y;
public:
   Point(const double x=0, const double y=0); ~Point() {}
// Селектор-модифікатор: double& дає доступ як для читання,
// так і для змінення поля)
   double& x() {return _x;}
```

```
   double& y() {return _y;}
};
```

Варто пригадати особливість повернення функцією відсилки (підрозділ 3.3.2). Таку функцію можна викликати як зліва, так і справа від присвоєння, наприклад

Лістинг 4.53

```
Point u(1, 2);
double a=u.x();   // Виклик селектора
u.y()=a;          // Виклик модифікатора
```

Однак у цьому рішенні не враховано, що селектор-модифікатор не можна застосувати до сталих точок:

Лістинг 4.54

```
const Point zero;
// Виклик селектора призведе до синтаксичної помилки
cout<<zero.x()<<',' <<zero.y();
```

оскільки він має повертати лівобічне значення, не визначене для сталих.

Вправа 4.8

Створіть функцію обчислення периметра трикутника, використовуючи селектори-модифікатори координат його вершин. Проаналізуйте, чи можна застосовувати її до сталого трикутника.

Виправимо ситуацію, додавши ще пару сталих селекторів, що відрізняються від селектора-модифікатора типом результату:

Лістинг 4.55

```
// Точки площини зі сталим селектором
// і селектором-модифікатором
struct Point {
private:
   double _x; double _y;
public:
   Point(const double x=0, const double y=0);
   ~Point() {}
// Сталі селектори: const гарантує незмінність об'єкта
   const double& x() const {return _x;}
   const double& y() const {return _y;}
// Селектор-модифікатор: double& дає доступ як для читання,
// так і для змінення координати
   double& x() {return _x;}
   double& y() {return _y;}
};
```

Тепер маємо по парі однойменних функцій доступу до координат. Цікаво простежити, які з них використає компілятор і в яких випадках. Для цього можна додати до текстів команди керованого виведення:

Лістинг 4.56

```
double& x() {
#ifndef NDEBUG
   cout<<"Abscissa of the point "<<*this<<"may be changed"<<endl;
#endif
   return _x;
}
const double&  x() const {
#ifndef NDEBUG
   cout<<"Abscissa of the point "<<*this<<"read only access"<<endl;
#endif
   return _x;
}
```

Вправа 4.9

Визначте структуру трикутників, наділену селекторами-модифікаторами. Візьміть два трикутники, один із яких сталий. Виведіть координати вершин кожного трикутника та проаналізуйте, які функції доступу було використано.

4.4. Класи та структури

Мабуть, настав час назвати речі своїми іменами: структуру, яку ми щойно визначили, зазвичай називають *класом* (class).

Лістинг 4.57

```
// Структура стала класом
class Point {
private:
   double _x; double _y;
public:
   Point(const double x=0, const double y=0);
   ~Point() {}
// Селектори: const гарантує незмінність об'єкта
   const double& x() const {return _x;}
   const double& y() const {return _y;}
// Модифікатори: double& дає доступ для змінення поля
   double& x() {return _x;}
   double& y() {return _y;}
};
```

Строго кажучи, класи в мові C++ нічим не відрізняються від структур. Навіть традиційні структури чистої мови C, з яких ми почали вивчати інкапсуляцію функцій за допомогою указників, потрапивши до компіляторів C++, наділяються конструктором і деструктором за умовчанням, а відтак поводяться як класи. Формально класи відрізняються від структур лише правилами надання прав доступу за умовчанням:

- усе, що явно не закрите в структурі, уважають відкритим;
- усе, що явно не відкрите в класі, уважають закритим.

Якщо ж не користуватися правилами дій за умовчанням і завжди явно задавати права доступу (що, до речі, вважають іще одним правилом хорошого тону), то зовсім немає відмінностей між структурами C та класами C++.

Почавши використовувати класи, застосовуватимемо термінологію, традиційну для об'єктного програмування.

Спочатку розглянемо поняття об'єкта. Його вживають у широкому та вузькому значеннях. Об'єкт у широкому значенні – це будь-яка визначена в програмі сутність – як змінна, так і стала – незалежно від її типу, наприклад числовий літерал, значення символьної змінної, константа `pi`, точка площини, трикутник тощо. Об'єкт у вузькому значенні – це *екземпляр класу (структури)*, що з'являється внаслідок виклику конструктора цього класу й існує доти, доки не буде викликано деструктор.

Ось приклади двох об'єктів-екземплярів класу `Point`:

```
Point u(1, 1); const Point zero;
```

Класи складаються з *членів класу* (class member), які можуть бути даними чи функціями. Поряд із терміном «член класу» для даних часто вживають термін *атрибут* (attribute), оскільки від цих членів залежать властивості класу. *Функції-члени класів* (member function) називають також *методами* (method) на відміну від *вільних функцій*, які не належать до класів (non-member function).

Варто зробити ще одне зауваження щодо співвідношення понять класу й *типу даних* (data type). Під типом у програмуванні розуміють певну сукупність даних із заданими на ній діями. Ми вже розглядали фундаментальні типи й типи указників. Структури C в повному розумінні цього слова важко віднести до типів, тому що в мові C немає механізмів повноцінного визначення операцій над структурами, зокрема операцій їх створення і видалення. Інша річ – класи в C++ чи навіть структури в процедурній частині C++. Поява функцій-членів класу дає можливість розглядати кожен клас як програмований тип. На рівні об'єктного програмування терміни «клас» і «програмований тип даних» стають синонімами.

Виникає запитання: чи варто переписувати структури як класи? На це можна дати дві відповіді. Передусім, у мові C++ структури збережено для того, щоб вона була сумісна з мовою C. Тому в робочих програмах мовою C можна все залишити як є. Складаючи ж нові програми, варто користуватися класами як стандартним поняттям об'єктно-орієнтованого програмування. Однак є також традиція залишати структури там, де об'єкту не потрібна власна функціональність, і його роль полягає лише в збиранні групи даних разом. Адже просте переписування структури

Лістинг 4.58

```
struct Complex {
   double _re; double _im;
};
```

як класу із заміною одного ключового слова на інше –

Лістинг 4.59

```
class Complex {
   double _re; double _im;
};
```

перетворює її на «річ у собі», бо в ньому тепер усе закрито, а відкритий конструктор за умовчанням лише ініціалізує атрибути сміттям. Те, що було простим для структур:

Лістинг 4.60

```
Complex z={1, 1}; z._re=2; z._im=3;
```

стало складним для класу: потрібен конструктор із параметрами, відкриті селектори, модифікатори.

Вправа 4.10

Визначте клас комплексних чисел із конструктором, деструктором, селекторами та модифікаторами в його відкритій частині.

Зауваження. *Ніколи не використовуйте задані за умовчанням права доступу. Завжди явно розміщуйте відповідні члени класів у відкритій та закритій частинах. Задані за умовчанням правила можна застосовувати хіба що в структурах, коли важливо зберегти сумісність із мовою програмування C.*

Коли використовувати класи, а коли – структури? Найчастіше ми працюватимемо з класами, і тоді:

- маємо стежити за створенням і видаленням об'єктів, регламентувати доступ до їхніх частин;
- будемо користуватися *золотим правилом доступу*: атрибути зазвичай закриті, а методи можуть бути відкритими;

- обов’язково визначатимемо конструктор(и) і деструктор, а також модифікатори та селектори для кожного з атрибутів, які призначено для використання ззовні.

Зробимо два зауваження до наведеного переліку.

1. Бувають закриті методи, використовувані лише всередині класу. Приклади таких методів наведемо пізніше.
2. Не всі атрибути наділено відкритими селекторами та модифікаторами. Є атрибути для внутрішнього вжитку – для них не буває відкритих селекторів. Бувають сталі атрибути – вони зовсім не мають модифікаторів.

Конструкторів може бути кілька. Розглянемо, наприклад, клас рядків символів `String` із двома конструкторами. Призначення атрибутів очевидне: `_allocator` – указник на сам рядок, `_len` – його довжина. Перший конструктор забезпечить створення порожнього рядка для визначення на зразок

```
String s;
```

Другий побудує рядок за оригіналом, на який вказує символьний указник або літерал, наприклад так:

```
String s("This is a string");
```

Такого вигляду набуде перше наближення до класу `String`. Назва нашої версії класу різнитиметься від стандартної тим, що починатиметься з великої літери:

Лістинг 4.61

```
// Перше наближення до класу рядків символів
// Лише конструктори та деструктор
class String {
private:
   size_t _len; // Довжина рядка
   char* _allocator; // Сам рядок
public:
   class BadString;      // Клас для обробки аварійних ситуацій
   String();             // Конструктор порожнього рядка
   String(const char);   // Конструктор за одним символом
   String(const char*);  // Конструктор C++-рядка за C-рядком
   ~String();            // Деструктор
};
```

Одразу зробимо попереднє зауваження стосовно обробки помилок. Дуже часто у визначенні класів можна побачити вкладений клас із назвою, схожою на `BadString`. Об’єкти таких класів буде створено в разі

виникнення аварійної ситуації під час виконання програми та використано для її обробки. Визначення цих класів зазвичай виносять за межі основного класу:

```
// Клас для обробки аварійних ситуацій у класі String        Лістинг 4.62
class String::BadString {
private:
   const string _reason; // Причина аварійної ситуації
   const size_t _index;  // Символ, на обробці якого вона виникла
public:
   BadString(string reason="", const size_t index=0):
            _reason(reason),_index(index) {
      return;
   }
   ~BadString() {}
// Діагностичне повідомлення
   void diagnose() const {
      cerr<<_reason<<endl;
      if(_index!=0)
         cerr<<' '<<_index<<endl;
   }
};
```

Повернімося до розгляду класу `String`. Альтернативою двом конструкторам міг би бути один, як, наприклад, у класі `Point`, але з нульовим значенням заданого за умовчанням параметра для порожнього рядка:

```
String(const char* ps=0);
```

На жаль, такий варіант не можна вважати вдалим, оскільки невизначений (тобто нульовий) указник у разі рядка символів вважають не порожнім рядком, а помилкою (порожній рядок, нагадаємо, складається з єдиного символу `'\0'`). Тому пропонуємо дві реалізації конструктора:

```
// Конструктори рядка                                        Лістинг 4.63
// Конструктор порожнього рядка
String::String():_allocator(new char[1]), _len(0) {
   _allocator[0]='\0';
   return;
}
// Конструктор рядка за символьним указником
String::String(const char* ps) {
   if(ps==0)
      throw BadString("Attempt to use not defined pointer");
```

```
   _len=strlen(ps); _allocator=new char[_len+1];
   strcpy(_allocator, ps);
   return;
}
```

Нагадаємо, що передавати указник як параметр завжди небезпечно. Якщо визначеність самого указника ще можна перевірити, то не гарантовано, що рядок, на який установлено цей указник, справді закінчується нульовим кодом. Зокрема, указнику можна просто передати адресу одного символу –

Лістинг 4.64

```
char c='a'; String s(&c);
```

Тоді застосування конструктора з параметром-указником для побудови рядка з одного символу завершиться невизначеним результатом, хоча синтаксично цей виклик коректний.

Потрібно завжди прагнути виявляти якомога більше помилок (а ще краще – усі) на етапі компіляції. Створення рядка з одного символу доцільно вилучити з конструктора за указником, додавши до класу ще третій конструктор. Однак сама лише наявність одного коректного інструмента ще не вбезпечує від некоректного використання іншого інструмента:

Лістинг 4.65

```
// Конструктор рядка з одного символу
String::String(const char c):_allocator(new char[2]), _len(1) {
   _allocator[0]=c; _allocator[1]='\0';
   return;
}
```

Про те, як надійніше працювати з указниками, ще йтиметься в підрозділі 5.16.

У вправі 4.11 наведено приклади визначення рядків і виклики відповідних конструкторів. Зауважимо, що порожній рядок можна визначити двома способами: за допомогою конструктора без параметра чи з параметром – порожнім рядком.

Вправа 4.11

Використайте налагоджувач і проаналізуйте, який із конструкторів буде викликано в кожному з наведених прикладів:

Лістинг 4.66

```
char* p="string"; String s(p);
String sliteral("literal"); char* pempty="";
String sempty(pempty); String empty; String sc('a');
```

Вправа 4.12

Самостійно реалізуйте деструктор класу `String`.

Створене перше наближення до класу рядків поки що ні для чого, крім створення та видалення, не придатне. Прийнявши «золоте правило доступу», ми втратили прямий доступ до значень атрибутів, а тому фактично ще не можемо обробляти рядки. Для цього потрібно додати до класу селектори та модифікатори.

Подібну проблему матимемо, перетворивши структуру загорнутого вектора на клас:

Лістинг 4.67

```
// Клас загорнутих векторів
class WrappedVector {
private:
   static const int _n; // Однакова розмірність усіх векторів класу
   double* _v;          // Указник на масив
public:
   WrappedVector();     // Конструктор створює об’єкт
   ~WrappedVector();    // Деструктор видаляє його
};
```

Поки атрибути структури були відкритими, вектором ми оперували за допомогою прямого доступу до його компонентів. У лістингу 4.68 наведено реалізацію операції скалярного множення векторів, коректну стосовно структури із лістингу 4.15, але неприйнятну для класу з лістингу 4.67:

Лістинг 4.68

```
// Пряме звернення до атрибутів не працює
// для класу із закритими атрибутами
double operator* (const WrappedVector& v1, const WrappedVector& v2) {
   double res=0;
// Немає доступу до v1._n, v1._v[i] і v2._v[i]
   for(int i=0; i<v1._n); i++) res+=v1._v[i]*v2._v[i];
   return res;
}
```

Щоб вийти з цієї ситуації, операцію скалярного множення векторів можна було б перенести всередину класу, зробивши її його членом. Про це ще йтиметься в наступному підрозділі, а потім детально у підрозділі 4.7. Але з огляду на майбутні використання класу загорнутих векторів із закритими атрибутами до нього слушно відразу додати селектори розмірності та доступу за індексом:

Лістинг 4.69

```
// Клас загорнутих векторів із примітивними селекторами
class WrappedVector {
private:
   static const size_t _n; double* _v;
```

```
public:
   WrappedVector();  // Конструктор створює об'єкт
   ~WrappedVector(); // Деструктор видаляє його
   size_t double size() const {return _n;} // Селектор розмірності n
// Селектор елементів вектора
   const double& elem(size_t i) const {return _v[i];}
};
```

Перепишемо реалізацію операції скалярного множення векторів:

Лістинг 4.70

```
// Множення векторів із закритими атрибутами
double operator* (const WrappedVector& v1, const WrappedVector& v2) {
   double res=0;
   for(size_t i=0; i<v1.size(); i++) // Доступ до розмірності вектора
                                     // через її селектор
// Доступ до елемента вектора через його селектор
      res+=v1.elem(i)*v2.elem(i);
   return res;
}
```

Вправа 4.13

Запрограмуйте операцію виведення загорнутого вектора.

Вправа 4.14

Доповніть клас комплексних чисел селектором і модифікатором. Реалізуйте операції введення та виведення комплексних чисел.

Відмову від звичних квадратних дужок для доступу до елемента вектора `v[i]` та заміну їх селектором елемента вектора `elem(i)` важко назвати вдалим рішенням. Утім, це тимчасове рішення. Крім природного бажання зберегти квадратні дужки (в лістингу 4.70 це були б вирази `v1[i]` і `v2[i]`) як звичний спосіб запису операції індексування, відкритим стає питання про модифікацію значень компонентів загорнутого вектора. Незабаром у підрозділі 4.8 ми навчимося позначати довільний компонент загорнутого вектора звичним способом за допомогою квадратних дужок, одночасно розв'язавши також проблему його модифікації.

4.5. Функції-члени класу та функції-утиліти класу

Кожна функція-член класу має повний доступ до його відкритої та закритої частин. Візьмемо версію класу `Point` із лістингу 4.55, визначимо кілька точок і поекспериментуємо з ними:

```
const Point zero; Point w(1, 1);
Point* pw=&w; Point* pa = new Point(1, 2);
```

Лістинг 4.71

Команда виведення абсцис кожної з точок із застосуванням методу – селектора абсциси матиме такий вигляд:

Лістинг 4.72

```
// x() використано як селектор
cout<<zero.x()<<endl; cout<<w.x()<<endl;
cout<<pw->x()<<endl; cout<<pa->x()<<endl;
```

Модифікатор абсциси можна застосувати лише до несталих об'єктів – точки a, указників pw та pa:

Лістинг 4.73

```
// x() використано як модифікатор
w.x()=2; pw->x()=3; pa->x()=4;
```

Як бачимо, виклик функції-члена класу має синтаксично такий самий вигляд, як і звернення до атрибута об'єкта: із крапкою, якщо об'єкт задано ім'ям, або стрілкою, якщо його задано указником. Оскільки виклик кожної функції класу прив'язано до конкретного об'єкта, то функції «відомо», до чого її застосовано. Це можна уявляти собі як прихований формальний параметр, який називають **this** – *указник на поточний об'єкт* (this pointer). Його тип – сталий указник на клас, до якого належить об'єкт. У довільному класі T тип **this** – T* **const**. Наприклад, у класі Point указник **this** має тип Point* **const**.

Отож, до окремого атрибута всередині функції класу можна звернутися просто за його іменем, а до цілого об'єкта – за допомогою указника **this**. Суттєво, що цей указник сталий, його не можна переспрямувати на інший об'єкт, але за його допомогою можна змінювати атрибути об'єкта, на який його встановлено.

Запитання. *Подумайте, яким має бути тип указника* **this** *для сталого об'єкта?*

Використання функцій класу не означає повної відмови від вільних функцій, які прийматимуть об'єкти певного класу як параметри. Їх зазвичай називають функціями обслуговування чи *утилітами класу* (class utility). Методи відрізняються від утиліт двома властивостями: у них, узагалі кажучи, більше прав доступу і їм відомий об'єкт, до якого їх застосовують. Утиліти мають одержати цей об'єкт як параметр.

Узагалі кажучи, кожну утиліту можна (проте не обов'язково потрібно) перетворити на функцію класу. Згадаймо функцію swap(), визначену, зокрема, і для точок площини:

```
// Утиліта обміну точок                                  Лістинг 4.74
void swap(Point& u, Point& v) {
   const Point w=u; u=v; v=w;
   return;
}
```

Після перетворення її на метод класу `Point` у ній стане на один параметр менше:

```
// Метод обміну точок                                    Лістинг 4.75
void Point::swap(Point& v) {
   const Point w=*this; *this=v; v=w;
   return;
}
```

Явно указник поточного об'єкту використовується рідко. Так звертаючись до атрибута в методах, указник на поточний об'єкт звичайно пропускають. Розглянемо, наприклад, тексти селектора і модифікатора записані в скороченому вигляді

```
double x() const {return _x;}
double& x() {return _x;}
```

Повністю їх треба було б подати так:

```
double x() const {return this->_x;}
double& x() {return this->_x;}
```

Так само, наприклад, повний запис деструктора загорнутого вектора мав би вигляд

```
WrappedVector::~WrappedVector() {                        Лістинг 4.76
   cout<<"Destructor WrappedVector"<<endl;
   delete [] this->_v;
   this->_v=0;
   return;
}
```

Однак скорочений запис можна застосувати не завжди. Із функцією `swap()` уже була ситуація (лістинг 4.75), коли не обійтися без указника на поточний об'єкт: так звертаються до цілого об'єкта, а не його окремого атрибута.

Методи суттєво відрізняються від вільних функцій правами доступу до частин об'єкта. Якщо перемістити задану раніше операцію множення векторів усередину класу `WrappedVector`, визначивши її як його метод, то

всі члени цього класу знову стануть доступними їй, оскільки функція класу також міститься в ньому, на відміну від утиліт, розміщених зовні:

Лістинг 4.77

```
// Метод скалярного множення векторів
// з використанням прямого доступу до атрибутів
double WrappedVector:: operator*
   (const WrappedVector& vec) const {
   double res=0;
   for(size_t i=0; i<_n); i++)
      res+=_v[i]*vec._v[i];  // res+= this->_v[i]*vec._v[i];
   return res;
}
```

У цьому разі клас загорнутих векторів доведеться доповнити новим методом:

Лістинг 4.78

```
// Клас загорнутих векторів з операцією скалярного множення
class WrappedVector {
private:
   static const size_t _n; double* _v;
public:
   WrappedVector();   // Конструктор створює об'єкт
   ~WrappedVector();  // Деструктор видаляє його
// Селектори
   size_t size() const {return _n;}
   size_t elem(int i) const {return _v[i];}
// Операція множення, переміщена всередину класу
   double operator*(const WrappedVector& v) const;
};
```

Зазначена операція матиме тепер на один параметр менше: першим множником у ній став указник **this**. Хоча в реалізації операції скалярного множення параметр **this** не вжито явно, він неявно є в кожному зверненні до членів поточного об'єкта, як видно у лістингу 4.77. Зазначимо також, що при цьому змінився його тип: в утиліті перший параметр мав тип **const** WrappedVector&, а тепер замість нього використано указник **this**, у типі якого відображено також сталість об'єкта, зумовлену заголовком функції. Отже, матимемо **const** WrappedVector* **const**. Наслідки цієї зміни проаналізуємо пізніше, коли порівнюватимемо операції-члени класу з операціями-утилітами.

Звісно, можна використовувати селектори не лише в утилітах, але й у методах класу, зокрема так:

```
// Метод множення векторів із використанням селекторів      Лістинг 4.79
double WrappedVector:: operator* (const WrappedVector& vec) const {
   double res=0;
   for(int i=0; i<size()); i++)
      res+= elem(i)*vec.elem(i);  // res+= this->elem(i)*vec.elem(i);
   return res;
}
```

Вправа 4.15

Визначте операцію додавання загорнутих векторів як член класу. Що потрібно додати до класу, щоб можна було задати цю операцію як утиліту?

Вправа 4.16

Визначте арифметичні операції над комплексними числами двома способами – як утиліти і як члени класу.

Досягнути оптимального балансу між функціями класу та його утилітами – одне з найважливіших наших завдань.

4.6. Копіювання об'єктів

Копіювання об'єктів – одна з найважливіших операцій в імперативному програмуванні, пов'язаному з лінійною пам'яттю комп'ютера, у комірках якої розміщено дані. Вони постійно переміщуються з місця на місце – скажімо, з пам'яті до арифметико-логічного пристрою чи програмного стека й назад. Власне, навіть центральний оператор імперативного програмування над пам'яттю – присвоєння – фактично виконує копіювання правої частини виразу до лівої. Тому для розробника програм дуже важливо правильно розуміти місце та суть копіювання, а ще важливіше – побачити можливість уникнути зайвого копіювання.

4.6.1. Конструктор копіювання

У попередніх розділах ви вже ознайомилися з двома обов'язковими методами класу – конструктором і деструктором. Зараз розглянемо третій обов'язковий метод – *конструктор копіювання* (copy constructor).

Почнемо формувати загальну схему будови класу:

```
// Загальна схема будови класу T. Версія 1.0              Лістинг 4.80
class T {
// Конструктор(и). Їх може бути багато з різними типами параметрів
   T(T1, …, Tn);
// Конструктор копіювання створює новий об'єкт,
// ідентичний до переданого за допомогою параметра
```

```
   T(const T&);
// Можливий варіант T(T&), але не T(T)
   ~T();// Деструктор
};
```

Найважливіше використання конструктора копіювання пов'язане з викликом функцій. Щоразу, коли параметр або результат передаються за допомогою значень, працює конструктор копіювання. Як приклад розглянемо функцію-перетворювач об'єктів класу T2 на об'єкти класу T1, яка одержує на вході об'єкт x класу T1 і повертає на виході об'єкт y класу T2:

Лістинг 4.81

```
// T2–T1-перетворювач
T1 f(T2 x) {T1 y(x); return y;}
```

Виклик цієї функції a=f(b) супроводжуватиметься викликами трьох конструкторів:

Лістинг 4.82

```
T2 x=T2(b); // Копіювання фактичного параметра b
            // копіювальним конструктором T2
T1 y=T1(x); // Виклик звичайного конструктора T1 із параметром типу T2
a=T1(y);    // Копіювання результату у копіювальним конструктором T1
```

Щоб відстежувати переміщення об'єктів, доцільно вести своєрідну інвентаризацію. Розглянемо приклад інвентаризації в класі Point. Створимо для кожної точки один незмінний атрибут _pointID – її інвентарний номер. Окрім цього, додамо до класу Point спільний статичний атрибут, значенням якого буде вільний інвентарний номер _freeID. Його значення в кожний момент часу – кількість створених до цього точок. Після створення нової точки він збільшуватиметься на одиницю та передаватиметься до цієї точки:

Лістинг 4.83

```
// Інвентаризовані точки площини
class Point {
private:
   static int _freeID; // Вільний інвентарний номер
   const int _pointID; // Інвентарний номер точки
// Координати точки
   double _x; double _y;
public:
   Point(const double x=0, const double y=0);
// Копіювальний конструктор
   Point(const Point&);
   ~Point() {}
```

```
// Селектори: const гарантує незмінність об'єкта
   const double& x() const {return _x;}
   const double& y() const {return _y;}
// Селектори-модифікатори: double& надає доступ для змінення координат
   double& x() {return _x;}
   double& y() {return _y;}
};
```

Як і раніше, тут ми розглядаємо демонстраційні версії конструкторів і деструктора, наділені операторами виведення своїх об'єктів. Доведеться підправити звичайний конструктор, щоб він міг запам'ятати й повідомити інвентарний номер створеної точки:

Лістинг 4.84

```
// Конструктор інвентаризованих точок
Point::Point(const double x, const double y):
      _x(x), _y(y), // Ініціалізація координат
// Надання точці власного інвентарного номера
      _pointID(++_freeID) {
   cout<<_pointID<<": created "<<*this<<endl;
   return;
}
```

Для застосування копіювального конструктора потрібно попередньо мати якусь точку, яка відіграватиме роль оригіналу. Її слід передавати сталою відсилкою, тому що, узагалі кажучи, копіювальний конструктор не має змінювати оригіналу. Координати копії ініціалізують атрибутами оригіналу, але копія одержує власний інвентарний номер:

Лістинг 4.85

```
// Копіювальний конструктор інвентаризованих точок
Point::Point(const Point& u):
// Копіювання координат
      _x(u._x), _y(u._y),
// Надання точці власного інвентарного номера
      pointID(++freeID) {
   cout<<pointID<<": copied "<<*this<<endl;
   return;
}
```

Деструктор так само, як конструктори, повідомляє номер видаленої точки:

Лістинг 4.86

```
// Деструктор інвентаризованих точок
Point::~Point() {
   cout<<_pointID<<": removed "<<*this<<endl;
```

```
   return;
}
```

Тепер можна відстежувати моменти створення, копіювання та видалення точок, зокрема дізнаватися, що відбувається під час передавання параметрів і повернення результату.

Вправа 4.17

Формально визначте класи `T1` і `T2` за схемою із лістингу 4.80 та простежте за викликами конструкторів під час роботи функції-перетворювача (лістинг 4.81).

Візьмемо найпростіший варіант операції додавання точок:

Лістинг 4.87

```
// Додавання точок: параметри та результат
// передано за допомогою значень
const Point operator+(const Point u, const Point v) {
   Point res(u.x()+v.x(), u.y()+v.y());
   return res;
}
```

Розглянемо приклад, у якому фігурують три об’єкти – `a`, `b` й `a+b`:

Лістинг 4.88

```
int main() {
   Point a(1, 2); Point b(5); a+b;
   return 0;
}
```

Конструктори тепер інформують про кожне виникнення нового об’єкта; дію їх усіх можна побачити на екрані. Загалом об’єктів виявиться шість, тобто вдвічі більше, ніж було передбачено в програмі. Ось протокол виведення:

Лістинг 4.89

```
1: created (1, 2)  // a
2: created (5, 0)  // b
3: copied (5, 0)   // v
4: copied (1, 2)   // u
5: created (6, 2)  // res
6: copied (6, 2)   // return
```

У коментарях записано, об’єкт із яким іменем було створено чи скопійовано (рядки 1–5). Шостий об’єкт не має власного імені, тому його позначено оператором, у якому він з’явився, – `return`.

Подальша частина протоколу виведення дає нам змогу пересвідчитися, що пам’ять не витрачено марно: усі шість створених об’єктів своєчасно видалено:

Лістинг 4.90

```
5: removed (6, 2) // res
4: removed (1, 2) // u
3: removed (5, 0) // v
6: removed (6, 2) // returned
2: removed (5, 0) // b
1: removed (1, 2) // a
```

Спробуємо зменшити кількість зайвих копій. Передусім виникає запитання: чи потрібно було заводити нову змінну для зберігання результату? Може, краще було прямо записати підсумковий вираз в операторі виходу?

Лістинг 4.91

```
// Додавання точок: параметри та результат передано
// за допомогою значень без використання допоміжної змінної
const Point operator+(const Point u, const Point v) {
/* замість
   Point res(u.x()+v.x(), u.y()+v.y());
   return res; */
   return Point(u.x()+v.x(), u.y()+v.y());
}
```

Поняття «краще» в цьому разі має дуже простий зміст – «дешевше». Тепер буде створено на один об’єкт менше:

Лістинг 4.92

```
1: created (1, 2) // a
2: created (5, 0) // b
3: copied (5, 0)  // v
4: copied (1, 2)  // u
5: created (6, 2) // return
4: removed (1, 2) // u
3: removed (5, 0) // v
5: removed (6, 2) // returned
2: removed (5, 0) // b
1: removed (1, 2) // a
```

Спробуємо зрозуміти, що сталося. Змінна `res` була локальним об’єктом функції, тому мала бути видалена й не могла зберігати значення результату після виходу з функції. Для того, щоб це значення можна було зберегти для передавання з функції, система створює додатковий неіменований об’єкт. Відмова від допоміжної змінної зумовлює збереження

результату відразу в неіменованому об'єкті, який можна винести за межі функції. Тепер немає потреби в одному копіюванні.

Заміна параметрів-значень сталими відсилками

Лістинг 4.93

```
// Додавання точок: параметри передано за допомогою
// сталих відсилок,результат – за допомогою значення
const Point operator+(const Point& u, const Point& v) {
   return Point(u.x()+v.x(), u.y()+v.y());
}
```

скоротить кількість об'єктів іще на два:

Лістинг 4.94

```
1: created (1, 2) // a
2: created (5, 0) // b
3: created (6, 2) // return
3: removed (6, 2) // returned
2: removed (5, 0) // b
1: removed (1, 2) // a
```

Проаналізуємо одержаний результат. Спочатку ми створили конструктор копіювання та виконали експеримент, який виявив зайві копіювання під час передавання параметрів і повернення результату. Після цього ми відмовилися від копіювання й одержали оптимальну програму. Чи не простіше було б узагалі не визначати копіювальний конструктор? Виявляється, що ні. Якщо взяти клас точок без власного копіювального конструктора та передавати параметри й повертати результат за допомогою значень, то всі копіювання відбуватимуться в тому самому порядку, але просто не будуть помітними.

Як це вже було у випадку конструктора і деструктора, система програмування доповнює кожен клас (структуру), у якому не визначено конструктор копіювання, *конструктором копіювання за умовчанням* (default copy constructor), який для точок площини нічим не відрізняється від запрограмованого нами, окрім обліку точок: копія одержить той самий інвентарний номер, який мав оригінал. Конструктор копіювання за умовчанням копіює об'єкти почленно, що виявилося не зовсім коректним уже в простому випадку інвентаризованих точок. Гірше за все, що таке копіювання відбувається неявно, і залишається небезпека просто не помітити виникнення проблеми.

Оскільки проблеми, пов'язані з копіювальним конструктором за умовчанням, не обмежуються зазначеними, ми повністю відмовимося від використання таких конструкторів. Із цього моменту оголошуємо будь-який конструктор за умовчанням «method non grata», насамперед тому, що хочемо контролювати всі випадки створення об'єктів.

Вправа 4.18

Доповніть клас комплексних чисел копіювальним конструктором.

Тепер перейдемо до нетривіального копіювання. Повернімося до класу `WrappedVector`:

Лістинг 4.95

```
// Загорнутий вектор із копіювальним конструктором
class WrappedVector {
private:
   static const int _n; double* _v;
public:
   WrappedVector(); WrappedVector(const WrappedVector&);
   ~WrappedVector();
};
```

Якщо в класі `WrappedVector` не передбачити власного конструктора копіювання, то блок програми, який містить фрагмент

Лістинг 4.96

```
WrappedVector u; WrappedVector w(u);
```

завершиться аварійно в момент виклику деструктора (поясніть самостійно, чому, або читайте про це далі).

Справді, якщо обмежилися конструктором копіювання за умовчанням, він мав би такий вигляд:

Лістинг 4.97

```
// Конструктор копіювання за умовчанням
WrappedVector::WrappedVector(const WrappedVector& vec):
// Поверхневе копіювання атрибута
      _v(vec._v) {
   return;
}
```

Раніше ми вже розглядали проблему копіювання структур даних, що містять указники, а тому маємо знати, що просте копіювання указника – груба помилка. Спочатку потрібно виділити пам'ять, достатню для копії вектора, запам'ятати її адресу в указнику та скопіювати значення елементів вектора до виділеної пам'яті. Назвемо цей варіант копіювання *глибинним* на відміну від *поверхового* почленного копіювання:

Лістинг 4.98

```
// Конструктор копіювання векторів
WrappedVector::WrappedVector(const WrappedVector& vec):
// Виділення пам'яті під копію вектора
      _v(new double[_n]) {
// Копіювання вектора на нове місце
```

```
   for(int i=0; i<_n; i++)
      _v[i]=vec._v[i];
   return;
}
```

Вправа 4.19

Доповніть реалізацію класу `WrappedVector` обліком створюваних, копійованих і видалених об’єктів.

Вправа 4.20

Визначте копіювальний конструктор для класу `String` і доповніть цей клас засобами інвентаризації.

Залишилось остаточно з’ясувати питання про тип параметра в копіювальному конструкторі. Доповнимо клас `String` іще одним атрибутом `_amountOfCopies`, призначення якого – зберігати кількість копій, зроблених із певного рядка:

Лістинг 4.99

```
// Клас рядків символів з обліком зроблених копій
// Наведено лише конструктори та деструктор
class String {
private:
   size_t _len;                   // Довжина рядка
   char* _allocator;              // Сам рядок
   unsigned int _amountOfCopies; // Кількість копій із цього рядка
public:
   class BadString;      // Клас для обробки аварійних ситуацій
   String();             // Конструктор порожнього рядка
   String(const char);   // Конструктор за одним символом
   String(const char*);  // Конструктор C++-рядка за C-рядком
   String(String&);      // Конструктор облікового копіювання
   ~String();            // Деструктор
};
```

Новий варіант конструктора облікового копіювання збільшуватиме вміст цього атрибута на одиницю для кожного копіювання оригіналу. Тому тепер передавання параметра за допомогою сталої відсилки стає неприйнятним:

Лістинг 4.100

```
// Конструктор облікового копіювання
String::String(String& s):
// ініціалізація атрибутів
      _amountOfCopies(0), _len(s._len),
```

```
// Виділення пам’яті під копію рядка
      _allocator(new char[_len+1]) {
// Кількість копій, зроблених з оригіналу, збільшується на одиницю
   s._amountOfCopies++;
   strcpy(_allocator, s._allocator); // Власне копіювання рядка
   return;
}
```

Отож, маємо два способи передачі параметра конструктору копіювання: за допомогою звичайної та сталої відсилки. На перший погляд цілком безпечна заміна сталої відсилки на звичайну може мати серйозні наслідки. У такому разі сталі рядки буде виключено з копіювання, оскільки оригінал стає недоступним для модифікацій.

Зауважимо нарешті, що розглянуті два способи передачі параметра копіювальному конструктору вичерпують усі допустимі можливості. Уявіть собі такий клас:

Лістинг 4.101

```
// Груба помилка: клас нескінченного копіювання
class Loop {
public:
   Loop() {}     // Конструктор
   Loop(Loop) {} // Некоректний конструктор копіювання
   ~Loop() {}    // Деструктор
};
```

Припустимо, що компілятор пропустив його. Що тоді відбулося б під час виконання такого коду?

Лістинг 4.102

```
   Loop x;    // Створили об’єкт класу Loop
   Loop y(x); // Намагаємося скопіювати його: Loop y=Loop(x)
```

Щоб виконати конструктор `Loop(x)`, потрібно передати йому параметр `x` за допомогою значення, а для цього слід знову викликати копіювальний конструктор для створення об’єкта `Loop y=Loop(x)`: процес продовжуватиметься нескінченно.

Оскільки під час передачі параметрів копіювальні конструктори викликаються неявно, стає надзвичайно важливим відстежувати всі такі випадки. Тому варто відмовитися від копіювальних конструкторів за умовчанням. Не менш важливо бути впевненим, що в класі реалізовано саме ту версію копіювання, яку передбачив розробник, а не, скажімо, поверхове копіювання за умовчанням.

4.6.2. Конструктор мультикопіювання

Дослідимо інший випадок копіювання, за допомогою якого, ми, зокрема, з'ясуємо, де містяться значення заданих за умовчанням параметрів.

Розглянемо ще один варіант класу String:

Лістинг 4.103

```
// Клас рядків із конструктором мультикопіювання
class String {
private:
   size_t _len;      // Довжина рядка
   char* _allocator; // Сам рядок
public:
   String();// Конструктор порожнього рядка
   String(const char);  // Конструктор за окремим символом
   String(const char*); // Конструктор С++-рядка за С-рядком
// Конструктор мультикопіювання
   String(const String& s, const unsigned int multiplayer=1);
   ~String();// Деструктор
};
```

Сюди додано *конструктор мультикопіювання* (multicopy constructor) із двома параметрами. Його також призначено для копіювання, але з повторенням оригіналу таку кількість разів, яку задає другий параметр multiplayer. Зрозуміло, що коли параметр має замовчуване значення 1, цей конструктор поводиться як звичайний конструктор копіювання та викликається саме так. Переглянемо текст його реалізації:

Лістинг 4.104

```
// Конструктор мультикопіювання
String:: String(const String& s, const unsigned int multiplayer):
// Обчислення довжини мультирядка
      _len(s._len*multiplayer),
// Виділення пам'яті під multiplayer рядків
      _allocator(new char [_len+1]) {
// Ініціалізація указника для початку копіювання
   char* target=_allocator;
   for(int i=0; i<multiplayer; i++) {
      strcpy(target, s._allocator); // Створення однієї копії
      target+=s._len; // Переміщення указника для продовження копіювання
   }
   return;
};
```

Наведемо приклади виклику:

```
char* p="Тратата,";
String s(p);     // Звичайний конструктор
String ss(s);    // Копіювальний конструктор
String s2(s, 2); // Конструктор мультикопіювання
```
Лістинг 4.105

Переконайтеся, що першого разу буде викликано звичайний конструктор, другого й третього – один і той самий конструктор мультикопіювання: спочатку для створення копії, а потім – подвоєної копії.

Унесемо у визначення класу на перший погляд незначну зміну – видалимо з оголошення методу задане за умовчанням значення параметра `multiplayer`:

```
// Клас рядків із конструктором мультикопіювання,
// заданий за умовчанням копіювальний конструктор
class String {
private:
   size_t _len; // Довжина рядка
   char* _allocator; // Сам рядок
public:
   String();            // Конструктор порожнього рядка
   String(const char*); // Конструктор С++-рядка за С-рядком
// Конструктор мультикопіювання
   String(const String& s, const unsigned int multiplayer);
   ~String();// Деструктор
};
```
Лістинг 4.106

Перенесемо його до реалізації (на жаль, синтаксис це дозволяє):

```
// Конструктор мультикопіювання
String::String(const String& s, const unsigned int multiplayer=1):
// Обчислення довжини мультирядка
      _len(s._len*multiplayer),
// Виділення пам'яті під multiplayer рядків
      _allocator(new char [_len+1]) {
// Ініціалізація указника для початку копіювання
   char* target=_allocator;
   for(int i=0; i<multiplayer; i++) {
      strcpy(target, s._allocator); // Створення однієї копії
      target+=s._len; // Переміщення указника для продовження копіювання
   }
   return;
}
```
Лістинг 4.107

Тепер розглянемо таку саму програму:

```
int main() {                                                   Лістинг 4.108
   char* p="Тратата,";
   String s(p); // Звичайний конструктор
   String ss(s); // Копіювальний конструктор за умовчанням
   String s2(s, 2); // Конструктор мультикопіювання
   return 0;
}
```

Рядок визначення ss виявиться помилковим. Проаналізуємо поведінку компілятора під час створення класу String за визначенням, наведеним у лістингу 4.106. Оскільки сигнатура конструктора мультикопіювання тепер не містить замовчування, а файл реалізації не бере участі в цьому етапі формування програми, то компілятор дійде висновку, що в класі немає програмованого копіювального конструктора, і доповнить цей клас заданим за умовчанням конструктором копіювання. Отже, під час створення рядка ss викликатиметься не мультиконструктор копіювання, а заданий за умовчанням конструктор, який виконає поверхове, а тому некоректне копіювання. Виконання програми завершиться аварійно.

Вправа 4.21

Проведіть комп'ютерний експеримент на підтвердження поведінки програми з лістингу 4.108

4.6.3. Оператор копіювального присвоєння

В об'єктному програмуванні копіювання об'єктів відбувається двома способами. Перший – використання копіювального конструктора – ми щойно розглянули. Другий полягає в застосуванні *оператора копіювального присвоєння* (copy assignment operator).

Як уже було зазначено, присвоєння – центральна операція в імперативному програмуванні. Тому поряд із конструктором, деструктором і копіювальним конструктором присвоєння стало четвертим стандартним методом у кожному класі. Знову ж таки для забезпечення сумісності з мовою C кожен клас у мові C++, зокрема кожну структуру, наділено присвоєнням – власним або заданим за умовчанням. Це дає нам змогу зробити наступний крок до створення загальної схеми будови класів.

```
// Загальна схема будови класу T з присвоєнням          Лістинг 4.109
class T {
// 1. Конструктор (їх може бути багато з різними типами параметрів)
   T(T1, …, Tn);
// 2. Деструктор
   ~T();
```

```
// 3. Конструктор копіювання створює новий об'єкт, ідентичний до
// переданого за допомогою параметра; найчастіше має параметр
// типу const T&. Можливий варіант: T(T&); але не T(T)
   T(const T&);
// 4. Копіювальне присвоєння
   T& operator=(const T&);
};
```

Зовсім просте для фундаментальних типів даних присвоєння стає складним у разі його почленного поширення на структури даних. Більше того, як було показано в підрозділі 2.9.1.4, застосоване до структур з указниками почленне (поверхове) копіювання виконується некоректно та призводить до аварійного завершення програми. Виникає певна суперечність між потребами у створенні копій структур і їх присвоєнням за умовчанням. Тому таке присвоєння доводиться програмувати окремо.

Візьмемо, наприклад, клас загорнутих векторів (лістинг 4.67). Їх копіювання можна було б реалізувати за допомогою утиліти – наприклад, із такою сигнатурою:

```
// Утиліта копіювання
void copy(WrappedVector& target, const WrappedVector& source);
```

або за допомогою функції класу

```
// Утиліта копіювання
void WrappedVector::copy(const WrappedVector& source);
```

Однак при цьому залишилося б чинним звичайне присвоєння за умовчанням

```
u=v;
```

яким компілятор наділяє кожну структуру, коли в ній немає програмованого присвоєння, з усіма його наслідками.

Візьмемо тепер клас інвентаризованих точок площини (лістинг 4.83) і спробуємо застосувати до них присвоєння за умовчанням:

Лістинг 4.110

```
// Спроба виконати стандартне присвоєння
   Point u(1,2); Point v; v=u;
```

Почленне присвоєння за умовчанням кожному з атрибутів виявиться неприйнятним, оскільки один із атрибутів класу сталий. Компілятор повідомить про помилку, спричинену неможливістю змінити значення сталого члена класу – інвентарного номера. Як бачимо, навіть у дуже простих ситуаціях присвоєння за умовчанням виявляється помилковим.

Тому у власних класах потрібно взагалі відмовитися від оператора присвоєння за умовчанням, завжди визначаючи його як програмований оператор.

Доповнимо клас точок площини `Point` програмованим присвоєнням:

Лістинг 4.111

```
// Інвентаризовані точки площини
// з програмованим присвоєнням
class Point {
private:
   static int _freeID; // Вільний інвентарний номер
   const int _pointID; // Інвентарний номер точки
   double _x, _y;      // Координати точки
public:
   Point(const double x=0, const double y=0);
   Point(const Point&); // Копіювальний конструктор
   ~Point();
   Point& operator=(const Point&); // Програмоване присвоєння
// Селектори-модифікатори: double& надає доступ для змінення координат
   double& x() {return _x;}
   double& y() {return _y;}
// Селектори: const гарантує незмінність об’єкта
   const double& x() const {return _x;}
   const double& y() const {return _y;}
};
```

Запропонуємо реалізацію присвоєння, у якій змінюються лише координати, а інвентарний номер залишається незмінним:

Лістинг 4.112

```
// Програмоване присвоєння інвентаризованих точок
Point& Point::operator=(const Point& u) {
// Змінюються лише значення координат
   _x=u._x; _y=u._y;
   return *this;
}
```

Як виявилося, у цьому разі довелося змінювати не всі атрибути, як у стандартній версії, а тільки їх частину. Це ще один приклад, де використано указник на поточний об’єкт **`this`**.

Обґрунтуємо запропоновану сигнатуру присвоєння

```
T& operator=(const T&);
```

Узагалі кажучи, мова програмування залишає за розробником програм право задавати присвоєння на власний смак. Головний критерій, яким

при цьому слід керуватися, полягає в тому, щоб поведінка програмованого присвоєння для програмованих структур даних узгоджувалася з поведінкою простого присвоєння для фундаментальних типів. Це особливо важливо в узагальненому програмуванні. Адже в разі розбіжностей у сигнатурах присвоєння для різних типів ними не вдасться конкретизувати шаблон узагальненої функції.

Утім розглянемо й інші, на перший погляд допустимі, варіанти сигнатур. Можливий різновид присвоєння, коли його параметр не має бути сталою відсилкою, –

```
T& operator=(T&);
```

Проте це зробить неможливим присвоєння сталої величини. Справді, якби сигнатура присвоєння мала вигляд

```
Point& operator=(Point&);
```

то код

```
const Point zero; Point u; u=zero;
```

Лістинг 4.113

став би некоректним (перевірте). Водночас аналогічне присвоєння для цілих –

```
const int zero=0; int x; x=zero;
```

Лістинг 4.114

повністю коректне.

Інший можливий варіант – оператор без результату

```
void operator=(const T&);
```

знову ж слід визнати невдалим, оскільки неможливим став би ланцюжок присвоєнь

```
const Point u=(1,2); Point v, w; w=v=u;
```

Лістинг 4.115

знову ж таки допустимий для цілих змінних:

```
int x, y; x=y=1;
```

Лістинг 4.116

Отже, в оператора присвоєння має бути значення, і ним має бути відсилка (поясніть, чому).

Вправа 4.22

Проаналізуйте варіанти сигнатури присвоєння

```
T operator=(T&)                                      Лістинг 4.117
T& operator=(T)
T operator=(T)
```

і обґрунтуйте їх непридатність.

Вправа 4.23

Визначте клас комплексних чисел із присвоєнням.

Присвоєння дуже просто реалізувати й у класі `WrappedVector`:

```
// Загорнутий вектор із копіювальним присвоєнням      Лістинг 4.118
class WrappedVector {
private:
   static const int _n; double* _v;
public:
   WrappedVector(); WrappedVector(const WrappedVector&);
   ~WrappedVector();
// Копіювальне присвоєння
   WrappedVector& operator=(const WrappedVector&);
};
```

Однак це вже не почленне копіювання (дослідіть, як поводитиметься клас `WrappedVector` у разі присвоєння за умовчанням):

```
// Копіювальне присвоєння векторів                    Лістинг 4.119
WrappedVector& WrappedVector::operator= (const WrappedVector& vec) {
   for(int i=0; i<_n; i++)
      _v[i]=vec._v[i];
   return *this;
}
```

Присвоєння відрізняється від копіювального конструктора загорнутого вектора тим, що воно не звертається до системи за виділенням вільної пам'яті, а розміщує нові значення елементів вектора на місцях попередніх значень. Це стало можливим лише завдяки збігу розмірностей обох векторів.

Нарешті, найзагальніший вигляд копіювального присвоєння можна проілюструвати на прикладі класу рядків:

```
// Рядки символів із присвоєнням                      Лістинг 4.120
class String {
private:
   char* _allocator; size_t _len;
```

```
public:
   class BadString;
   String();                       // Конструктор порожнього рядка
   String(const char*);            // Конструктор C++-рядка за C-рядком
   String(const String&);          // Копіювальний конструктор
   String& operator=(const String& s); // Копіювальне присвоєння
   ~String();                          // Деструктор
};
```

На перший погляд, реалізація присвоєння могла б бути такою:

Лістинг 4.121

```
// Присвоєння рядків не захищене від випадку s=s:
String& String::operator=(const String& s) {
   delete [] _allocator;          // Видалення старого рядка
   _len=s._len;                   // Нове значення довжини
   _allocator=new char[_len+1];   // Виділення місця для нового значення
   strcpy(_allocator, s._allocator); // Копіювання рядка на нове місце
   return *this;
}
```

Тепер згадаємо про умову, якої було домовлено дотримувати, а саме про узгодженість із присвоєнням для фундаментальних типів. Розглянемо, здавалося б, смішне присвоєння:

Лістинг 4.122

```
int x=1; x=x;
```

Перевіримо поведінку запрограмованого нами присвоєння для рядків:

Лістинг 4.123

```
String s="Скопіюй мене";
cout<<s<<':'; s=s; cout<<s<<endl;
```

Не варто дивуватися, якщо рядок виявиться спотвореним. Адже за пристайності лівої та правої частин, видаляючи рядок зліва, ми одночасно руйнуємо й правий: далі буде присвоєно сміття. Тому стандартна реалізація копіювального присвоєння зазвичай містить додаткову перевірку на пристайність лівої та правої частин. Якщо вони збігаються, то подальші дії стають зайвими. Наведемо остаточний текст реалізації копіювального присвоєння рядків:

Лістинг 4.124

```
// Копіювальне присвоєння рядків
String& String::operator=(const String& s) {
   if(this==&s)        // Перевірка пристайності лівої та правої частин
      return *this;
// Сюди потрапимо лише коли ліва та права частини різні
```

```
   delete [] _allocator;          // Видалення старого рядка
   _len=s._len;                   // Нове значення довжини
   _allocator=new char[_len+1];   // Виділення місця під нове значення
   strcpy(_allocator, s._allocator);  // Копіювання рядка на нове місце
   return *this;
}
```

Розглянемо відмінності між копіювальним присвоєнням і копіювальним конструктором. Останній завжди створює новий об'єкт, а копіювальне присвоєння замінює один об'єкт на інший. Тому перша особливість присвоєння полягає в тому, що спочатку треба видалити об'єкт, який буде заміщено. Однак перед видаленням потрібно перевірити ліву та праву частини на тотожність, у разі якої копіювання взагалі стає зайвим. Така перевірка – друга особливість копіювального присвоєння. Третя особливість полягає в поверненні результату за допомогою указника **this**.

На завершення порівняємо вживання копіювального конструктора та копіювального присвоєння в різних ситуаціях. Передусім тепер можна абсолютно точно пояснити, чим відрізняється присвоєння від ініціалізації, – конструкторами. Знову візьмемо для прикладу рядки символів. Виконання коду

Лістинг 4.125

```
String s("What a string!");
String ss; ss=s;
```

полягає у виклику конструктора рядка s, потім – конструктора порожнього рядка ss, а тоді – присвоєння: вивільнення пам'яті, виділення нової та копіювання. У коді

Лістинг 4.126

```
String s("What a string!"); String ss=s;
```

закладено на одну дію менше: услід за викликом конструктора рядка s іде виклик одного конструктора копіювання. Замість двох дій із попереднього лістингу – виклику звичайного конструктора та присвоєння – виконується одна: копіювальний конструктор.

Іншим прикладом із нетривіальним копіюванням може бути клас векторів різної розмірності:

Лістинг 4.127

```
// Клас векторів різної розмірності
class DissimilarVector {
private:
   size_t _n;  // Розмірність вектора
   double* _v; // Власне вектор
```

```
public:
// Стандартний набір із конструктора, деструктора, копіювань
   DissimilarVector(const size_t);
   DissimilarVector(const DissimilarVector&);
   ~DissimilarVector();
   DissimilarVector& operator=(const DissimilarVector&);
};
```

Вправа 4.24

Реалізуйте конструктор, деструктор, копіювальний конструктор і копіювальне присвоєння для класу векторів різної розмірності.

Як підсумок розгляду засобів створення, копіювання та видалення об'єктів зазначимо, що, розробляючи об'єктні програми, слід уникати об'єктів-фантомів, які через неявні виклики конструкторів виникають і зникають несподівано й часто непомітно для розробника. Тому потрібно вести ретельний облік об'єктів, стежити за їх створенням, копіюванням і видаленням.

4.7. Операції-члени класу й операції-утиліти класу

Серед функцій, з яких складаються програми, особливе місце займають ті, що реалізують стандартні операції. Більшість із них можна реалізувати двома способами – у вигляді методів або утиліт класу.

Для зручності використання операцій було прийнято рішення, згідно з яким виклик операції-методу синтаксично не відрізняється від виклику операції-утиліти, але інтерпретують їх по-різному. Нехай `ω` – довільна програмована операція. Її виклик із параметрами `u` та `v` завжди записують у звичному інфіксному вигляді `u ω v`. , наприклад, `u + v` , `u - v` тощо. Проте в разі реалізації операції за допомогою методу цей запис являє собою скорочення префіксного запису виклику методу з одним параметром `u.operatorω(v)`, у якому `u` передано як **`this`**, а `v` – відповідно до типу параметра методу, наприклад, `u.operator+(v)`. Якщо ж операцію реалізовано як утиліту, то той сам інфіксний запис виразу `u ω v` буде інтерпретовано як скорочення префіксного запису виклику вільної функції з двома параметрами `operatorω(u, v)`, наприклад, `operator+(u, v)`, тобто кожен фактичний параметр передається відповідно до специфікації формального. Зауважимо, що мова програмування допускає обидві форми запису, як повний, так і скорочений, за винятком деяких випадків, коли скорочений запис виявляється синтаксично некоректним.

Наведемо приклади.

```
// Клас із вбудованим додаванням                                  Лістинг 4.128
class BuiltInOps {
private:
   int _x;    // Атрибут для накопичення суми
public:
   BuiltInOps(const int x=0): _x(x) {}  // Конструктор
   const int& x() const {return _x;}    // Селектор
// Додавання-член класу
   const BuiltInOps operator+ (const BuiltInOps& x) const {
     return BuiltInOps(_x+x._x);
   }
};
// Утиліта виведення BuiltInOps
ostream& operator<<(ostream& os, const BuiltInOps& u) {
   os<<u.x();
   return os;
}
// Клас з утилітою додавання
class UtilityOps {
private:
   int _x; // Атрибут для накопичення суми
public:
   UtilityOps(const int x=0): _x(x) {}  // Конструктор
   const int& x() const {return _x;}    // Селектор
};
// Утиліта додавання
const UtilityOps operator+ (const UtilityOps& x, const UtilityOps& y) {
   return UtilityOps(x.x()+y.x());
}
// Утиліта виведення UtilityOps
ostream& operator<<(ostream& os, const UtilityOps& u) {
   os<<u.x();
   return os;
}
// Застосуємо кожну з операцій додавання
#include <iostream>
using namespace std;
int main() {
   BuiltInOps a(1), b(2);
   cout<<a+b<<',';                   // Скорочений запис
   cout<<a.operator+(b)<<endl;       // Повний запис
   UtilityOps c(3), d(4);
   cout<<c+d<<',':                   // Скорочений запис
```

```
   cout<<operator+(c,d)<<endl;       // Повний запис
   return 0;
}
```

Як бачимо, скорочений запис виклику операції-члена класу нічим не відрізняється від виклику операції-утиліти. Тому одну й ту саму операцію в класі не можна визначати водночас обома цими способами. Це обмеження не стосується функцій, оскільки їх не можна записувати скорочено. Тому вони різняться викликами (порівняйте повні виклики операцій).

Важливо! *Визначення програмованої арифметичної операції дозволено лише тоді, коли один із її аргументів сам має програмований тип. Якщо це перший аргумент, то незалежно від типу другого операцію можна реалізувати одним із двох способів. Якщо ж перший аргумент має стандартний тип, то другий обов'язково має бути програмованого типу, але операцію реалізують лише за допомогою утиліти (поясніть, чому).*

Кожен спосіб реалізації має свої особливості, першу з яких, а саме спосіб виклику функції, щойно було розглянуто, а про інші йтиметься далі.

Перший приклад програмованої операції – присвоєння – ми розглянули в попередньому підрозділі. ***Особливість присвоєння порівняно з іншими програмованими операціями полягає в тому, що воно може бути лише членом класу***. Причина зрозуміла: якщо компілятор не знайде присвоєння серед членів класу, він доповнить клас заданим за умовчанням присвоєнням.

Тепер ознайомимося з двома різними підходами до визначення операцій. Почнемо з класу комплексних чисел і наведемо його заголовний файл:

Лістинг 4.129

```
// Заголовний файл комплексних чисел Complex.h
#ifndef COMPLEX_H
#define COMPLEX_H
#include <iostream>
#include <math.h>
using namespace std;
// Клас комплексних чисел
// Арифметичні та логічні операції у вигляді утиліт
class Complex {
private:
   double _re; // Дійсна частина
   double _im; // Уявна частина
public:
   Complex(const double re=0, const double im=0): _re(re), _im(im) {}
   Complex(const Complex& z): _re(z._re), _im(z._im) {}
```

```
   ~Complex() {}
// Селектори
   const double& re() const {return _re;}
   const double& im() const {return _im;}
// Селектори-модифікатори
   double& re() {return _re;}
   double& im() {return _im;}
// Модуль і аргумент
   double mod() const {return sqrt(_re*_re+_im*_im);}
   double arg() const {return atan2(_im, _re);}
// Спряжене число
   const Complex conj() const {return Complex(_re, -_im);}
// Присвоєння просте й суміщені
   Complex& operator=(const Complex&);
   Complex& operator+=(const Complex&);
   Complex& operator-=(const Complex&);
   Complex& operator*=(const Complex&);
   Complex& operator/=(const Complex&);
};
// Утиліти класу Complex
   const double& re(const Complex& z) {return z.re();}
   const double& im(const Complex& z) {return z.im();}
// Операції введення-виведення
   ostream& operator<<(ostream&, const Complex&);
   istream& operator>>(istream&, Complex&);
// Арифметичні операції
   const Complex operator+ (const Complex&, const Complex&);
   const Complex operator- (const Complex&, const Complex&);
   const Complex operator* (const Complex&, const Complex&);
   const Complex operator/ (const Complex&, const Complex&);
// Порівняння
   bool operator==(const Complex&, const Complex&);
   bool operator!=(const Complex& a, const Complex& b) {return !(a==b);}
#endif
```

Зауважимо відразу, що деякі функції й навіть операції можуть бути одночасно реалізовані двічі: як операції та як утиліти. Для нас це буде важливо, бо дає змогу порівнювати зазначені способи реалізації.

Для простоти, перш ніж перейти до операцій, порівняємо утиліти доступу до дійсної та уявної частин із відповідними методами доступу.

Селектор визначено як сталий метод класу, який не може змінювати свій об'єкт. Раніше вже йшлося про так звані *сталі функції класу* (constant member function), до яких належать і селектори дійсної та уявної частин

```
double Complex::re() const {return _re;}
double Complex::im() const {return _im;}
```

Особливість будь-якого сталого методу полягає в тому, що йому заборонено змінювати значення членів поточного об'єкта, тому його указник **this** одержує тип сталого указника сталої (пригадайте сумніви в доцільності такого типу), тобто **const** Complex* **const**.

Утиліта, яку використовують як селектор, одержує параметр – сталу відсилку без права її змінення:

```
const double& re(const Complex& z) {return z.re();}
const double& im(const Complex& z) {return z.im();}
```

Тому утиліта за будь-якого типу фактичного параметра гарантовано відсилає до селектора, а не селектора-модифікатора, як це сталося б у разі безпосереднього виклику метода: виклик, ініційований сталим об'єктом, приводить до селектора, в усіх інших випадках викликається селектор-модифікатор.

Сигнатура утиліти могла бути трохи іншою – сталим значенням без відсилки

```
double re(const Complex z) {return z.re();}
```

Зауважимо, що зміна типу параметра веде до негайної зміни типу результату (чому?) Проте, як відомо, це призводило б до зайвого копіювання аргументу, а затим і результату. Тому ми надаємо перевагу сталій відсилці.

Зверніть увагу на те, що сталий параметр змушує компілятор викликати селектори, а не модифікатори-члени класу `re()` та `im()` в однойменних утилітах. Важливо також, що сталість методу – складова частина його сигнатури, за якою компілятор розпізнає однойменні методи. У класі комплексних чисел є пари однойменних методів. Наприклад, сигнатури методів

```
const double& re() const {return _re;}
double& re() {return _re;}
```

майже не відрізняються. Як компілятор розпізнає їх виклики? Лише за ознакою сталості. До сталого об'єкта буде застосовано лише селектор, до змінного – селектор-модифікатор.

Отже, кожним з імен `re()` та `im()` у класі комплексних чисел позначено три схожі функції: методи селектора та модифікатора, а також утиліту-селектор. Це зроблено передусім для того, щоб показати можливість статичної полісемії (підрозділ 3.8), яка полягає в застосовуванні різних

сигнатур до одного й того самого імені функції, чим виправдано однойменність селектора та модифікатора. Що ж до одночасного використання утиліти, то, як побачимо далі, утиліти надають більше свободи в перетворенні типів своїх аргументів.

Вправа 4.25

Дослідіть виклики селекторів і модифікаторів для різних видів виразів комплексного типу: константи, змінної, суми тощо.

Операцію виведення реалізують звичним способом як утиліту із застосуванням селекторів, – наприклад, так:

Лістинг 4.130

```
// Виведення комплексного числа
ostream& operator<<(ostream& os, const Complex& a) {
   os<<'('<<a.re()<<'+'<<a.im()<<"i)";
   return os;
}
```

Вправа 4.26

Запропонуйте операцію введення комплексних чисел і реалізуйте за допомогою утиліти класу.

Розглянемо тепер арифметичні операції. Усі їх визначимо однотипно як утиліти. Наведемо одну з них, наприклад, множення:

Лістинг 4.131

```
// Утиліта множення комплексних чисел
const Complex operator* (const Complex& a, const Complex& b) {
   return Complex(a.re()*b.re()-a.im()*b.im(),
      a.re()*b.im()+a.im()*b.re());
}
```

Параметри не копіюються, тому конструктори копіювання не буде викликано, але один виклик звичайного конструктора потрібен для створення результату. Для цього не вдасться використати жоден із параметрів, оскільки вони незмінні (цікаво буде далі порівняти арифметичні операції з операціями, суміщеними з присвоєнням, в яких новий об’єкт не створюється, оскільки присвоєння модифікує наявний об’єкт).

Вправа 4.27

Реалізуйте три інші арифметичні операції над комплексними числами – додавання, віднімання та ділення.

Розглянемо приклад застосування арифметичних утиліт:

```
Complex z1(1, 1), z2(0, 1);
cout<<z1*z2<<endl; cout<<10*z1<<endl; cout<<z1*10<<endl;
```

Лістинг 4.132

Він ілюструє одну важливу властивість конструктора як *перетворювача типів* (conversion constructor). Детальніше про це йтиметься в підрозділі 5.10.

Утім коротке пояснення цієї властивості конструктора варто зробити вже зараз.

Примітка. *Якщо клас `T` має конструктор з одним параметром типу `D` (параметрів може бути більше, але тоді значення всіх інших мають бути задані за умовчанням, щоб уможливити виклик конструктора з одним фактичним параметром), то цей конструктор можна використовувати як перетворювач типів, а саме від типу, що відповідає класу `D`, до типу класу `T`. Це означає, що скрізь, де має бути об'єкт типу `T`, можна замість нього використовувати об'єкт типу `D` за умови, що неявне перетворення в принципі дозволено (нагадаємо, що такі перетворення заборонено для указників і відсилок). Справді, наприклад, вираз `10*z1` не підходить під сигнатуру жодної з операцій множення дійсних або комплексних чисел. Компілятор знаходить варіант конструктора, який, приймаючи один дійсний аргумент `10`, створює комплексне число `10+0i`. Відбувається неявне перетворення дійсного числа на комплексне. Те саме станеться й у разі перетворення другого аргументу у виразі `z1*10`. Порівняйте ці дії з обробкою виразу `2*pi`: операції множення цілого числа на дійсне немає, але можна перетворити ціле число на дійсне та скористатися операцією множення дійсних чисел. Саме цю властивість мішаних операцій було узагальнено для класів. Засоби інвентаризації об'єктів дають змогу відстежувати всі випадки неявних викликів конструкторів.*

Вправа 4.28

Доповніть конструктори комплексних чисел засобами інвентаризації та проаналізуйте приклади неявного виклику конструкторів.

Вправа 4.29

Дослідіть випадки застосування перетворювачів типів у класі точок площини `Point`.

Аналогічно до арифметичних визначають і булеві операції:

```
// Порівняння комплексних чисел на рівність
bool operator==(const Complex& a, const Complex& b) {
   return a.re()==b.re()&&a.im()==b.im();
}
```

Лістинг 4.133

Раніше вже йшлося про те, що варто взяти за правило завжди задавати нерівність не явно, а за допомогою заперечення рівності. По-перше, так

важко припуститися помилки, по-друге, цей спосіб доцільний в ієрархічному програмуванні.

```
// Порівняння комплексних чисел на нерівність          Лістинг 4.134
bool operator!=(const Complex& a, const Complex& b) {
   return !(a==b);
}
```

Тепер перенесемо операції над комплексними числами всередину класу:

```
// Клас комплексних чисел                                 Лістинг 4.135
// Арифметичні та логічні операції у вигляді членів класу
class Complex {
private:
   double _re; // Дійсна частина
   double _im; // Уявна частина
public:
   Complex(const double re=0, const double im=0): _re(re), _im(im) {}
   Complex(const Complex& z): _re(z._re), _im(z._im) {} ~Complex() {}
// Селектори
   const double& re() const {return _re;}
   const double& im() const {return _im;}
// Селектори-модифікатори
   double& re() {return _re;}
   double& im() {return _im;}
// Модуль і аргумент
   double mod() const {return sqrt(_re*_re+_im*_im);}
   double arg() const {return atan2(_im, _re);}
// Спряжене число
   const Complex conj() const {return Complex(_re, -_im);}
// Присвоєння просте та суміщені
   Complex& operator=(const Complex&);
   Complex& operator+=(const Complex&);
   Complex& operator-=(const Complex&);
   Complex& operator*=(const Complex&);
   Complex& operator/=(const Complex&);
// Операції в класі
   const Complex operator+(const Complex&) const;
   const Complex operator-(const Complex&) const;
   const Complex operator*(const Complex&) const;
   const Complex operator/(const Complex&) const;
   bool operator==(const Complex&) const;
   bool operator!=(const Complex&) const;
};
```

Знову для прикладу обмежимося реалізацією одного методу, цього разу ділення:

Лістинг 4.136

```
// Утиліта ділення комплексних чисел
const Complex Complex::operator/ (const Complex& a) const {
   double m2=a.mod(); m2*=m2;
   return Complex((this->re()*a.re()+this->im()*a.im())/m2,
                  (this->im()*a.re()-this->re()*a.im())/m2);
}
```

Тепер можна відразу побачити важливу відмінність між утилітами та членами класу; її можна назвати *несиметричністю аргументів*. Знову повторимо попередній експеримент із застосуванням множення до комплексних чисел, заданого в лістингу 4.131:

Лістинг 4.137

```
Complex z1(1, 1), z2(0, 1);
cout<<z1*z2<<endl;  // Правильне множення
cout<<10*z1<<endl;  // Синтаксична помилка
cout<<z1*10<<endl;  // Правильне мжоження
```

Операція `10*z1`, на відміну від `z1*10`, виявиться помилковою.

Причина цього зрозуміла: оскільки перший аргумент виразу `10*z1` не є комплексним числом, то до цього виразу не можна застосувати операцію класу комплексних чисел. Справді, який об'єкт тоді виконував би роль **`this`**? Водночас вираз `z1*10` і надалі залишається коректним: указник **`this`** буде спрямованим на `z1`, а другий аргумент, як і раніше, буде створено за допомогою неявного виклику конструктора. Як бачимо, у разі реалізації операцій у вигляді методу перший і другий аргументи виявляються нерівноправними, тому що перший із них – це указник, а другий – стала відсилка.

Нагадаємо, що оператор присвоєння визначають лише як метод класу. Його реалізація очевидна:

Лістинг 4.138

```
// Присвоєння комплексних чисел
Complex& Complex::operator=(const Complex& a) {
   _re=a._re; _im=a._im;
   return *this;
}
```

Цікаво, що присвоєння, суміщене з операціями, дозволено у двох варіантах. Ось метод класу для присвоєння з додаванням:

```
// Суміщене з додаванням присвоєння в класі                Лістинг 4.139
Complex& Complex::operator+=(const Complex& a) {
   _re+=a._re; _im+=a._im;
   return *this;
}
```

Якщо визначити іншу версію класу комплексних чисел, у якому операції буде оформлено утилітами, то їх реалізації не матимуть прямого доступу до атрибутів класу, а тому потрібно буде використовувати селектори та модифікатори:

```
// Утиліта суміщеного з додаванням присвоєння              Лістинг 4.140
Complex& operator+=(Complex& a, const Complex& b) {
   a.re()+=b.re(); a.im()+=b.im();
   return a;
}
```

Зрозуміло, що ліворуч і праворуч від присвоєння використано різні однойменні методи: відповідно модифікатори та селектори (поясніть, чому).

Операцію додавання часто визначають за допомогою суміщеного присвоєння – виявляється, що селектори для цього не потрібні. Розглянемо приклад члена класу:

```
// Операція додавання в класі, реалізована                 Лістинг 4.141
// за допомогою суміщеного присвоєння
const Complex Complex::operator+ (const Complex& a) const {
   Complex res(*this);
   return res+=b;
}
```

й утиліту:

```
// Утиліта додавання, реалізована за допомогою             Лістинг 4.142
// суміщеного присвоєння
const Complex operator+ (const Complex& a, const Complex& b) {
   Complex res(a);
   return res+=b;
}
// Або навіть так, покладаючи копіювання першого аргумента
// на передачу параметра
const Complex operator+ (Complex a, const Complex& b) {
   return a+=b;
```

Вправа 4.30

Порівняйте ефективність прямої реалізації додавання та реалізації за допомогою суміщеного присвоєння з погляду затрат на створення й копіювання об'єктів.

Вправа 4.31

Реалізуйте інші операції, суміщені з присвоєнням, у класі комплексних чисел із лістингу 4.135.

Вправа 4.32

Визначте клас комплексних чисел, у якому всі операції, для яких це допустимо, буде оформлено як утиліти.

4.8. Властивості арифметичних і логічних операцій

Для економії позначень у програмованих типах даних варто довизначити стандартні операції тоді, коли вони мають сенс. Так, для багатьох типів даних доцільно визначити порівняння на рівність і його заперечення. Інші типи даних можна впорядковувати відношеннями «більше» та «менше». Неважко довизначити й арифметичні операції, передусім додавання та віднімання, а в деяких випадках – і множення, у багатьох нечислових класах: рядках, векторах, матрицях тощо. Крім того, уніфікація позначень особливо важлива для того, щоб можна було використовувати узагальнені функції. Детальніше про це йтиметься в підрозділі 5.20.

Визначаючи ту чи іншу стандартну операцію для програмованого типу даних, наприклад, класу, важливо пам'ятати, що кожна зі стандартних операцій має загальновідомі властивості. У програмованих операцій над класами мають бути ті самі властивості, що й у стандартних операцій над вбудованими типами даних. Рівність має бути рефлексивною, симетричною та транзитивною. Адже програма стане дуже дивною, якщо запрограмована операція віднімання не буде оберненою до додавання чи додавання стане неасоціативним. І як же тоді поводитиметься узагальнена функція, у якій використано таку операцію?

Знову звернімося до класу векторів:

Лістинг 4.143

```
// Клас загорнутих векторів
class WrappedVector {
private:
   static const size_t _n; double* _v;
public:
```

```
// Обов'язковий набір функцій
   WrappedVector(); WrappedVector(const WrappedVector&);
   WrappedVector& operator=(const WrappedVector&);
   ~WrappedVector();
// Селектор розмірності
   size_t size() const {return _n;}
// Набір програмованих операцій
   bool operator==(const WrappedVector&) const;
   bool operator!=(const WrappedVector&) const;
   double operator*(const WrappedVector&) const;
   const WrappedVector operator*(double) const;
   const WrappedVector operator+(const WrappedVector&) const;
};
```

Реалізація перших трьох із набору програмованих операцій очевидна:

Лістинг 4.144

```
// Порівняння на рівність (член класу)
bool WrappedVector::operator== (const WrappedVector& u) const {
   bool res=true;
   for(size_t i=0; i<size(); i++)
      res=res&&(_v[i]==u._v[i]);
   return res;
}
```

Вправа 4.33

Запропонуйте спосіб прискорення наведеного порівняння, закінчивши його за першої ж розбіжності.

Лістинг 4.145

```
// Порівняння на нерівність (член класу)
bool WrappedVector::operator!= (const WrappedVector& u) const {
   return !(*this==u);
}
// Скалярне множення (член класу)
double WrappedVector::operator* (const WrappedVector& u) const {
   double res=0;
   for(size_t i=0; i<size(); i++)
      res+=_v[i]*u._v[i];
   return res;
}
```

Операція множення вектора на число може спричинити проблеми. Легко піддатися спокусі перемножити вектор, використовуючи указник **this** як результат – наприклад, так:

```
// Помилкова, але синтаксично коректна версія!!!              Лістинг 4.146
// Множення вектора на число супроводжується спотворенням вектора
const WrappedVector& WrappedVector:: operator*(double x) const {
   for(int i=0; i<size(); i++)
      _v[i]*=x;
   return *this;
}
```

На перший погляд здається, що помилкова реалізація функції мала би бути й синтаксично некоректною, оскільки функцію визначено як сталу. Виявляється, однак, що поточний об'єкт не захищено від змінення навіть сталістю методу, яка поширюється лише безпосередньо на його атрибути. Справді, у сталій функції над вектором не дозволено змінювати значення атрибута `_v`, переспрямувавши його в інше місце, але цілком можливо змінити значення того, на що він показує.

Вправа 4.34

Візьміть вектор `u`, наприклад, із 5 елементів, усі координати якого дорівнюють 1. Тоді природно, що всі координати подвоєного вектора мають дорівнювати 2. Помноживши скалярно вектор `u` на подвоєний вектор `u`, маємо одержати 10. Виконайте комп'ютерний експеримент, узявши наведену в лістингу 4.146 функцію множення вектора на число та самостійно запрограмовану функцію скалярного множення, й обчисліть значення виразу `(u*2)*u`. Порівняйте його з очікуваним. Поясніть причину невідповідності.

Тому треба дуже акуратно програмувати операції над класами, де використано атрибути непрямого адресування.

Вправа 4.35

Проаналізуйте наведений нижче код і знайдіть помилку.

```
// Множення вектора на число з                                 Лістинг 4.147
// з поверненням відсилки тимчасового об'єкта
const WrappedVector& WrappedVector:: operator*(const double x) const {
   WrappedVector res;
   for(size_t i=0; i<size(); i++)
      res._v[i]=_v[i]*x;
   return res;
}
```

На жаль, тут «програмувати правильно» означає «з великими затратами на копіювання».

Може здатися, що зазначену неефективність легко усунути, явно виділяючи пам'ять для результату й повертаючи сталу відсилку на неї:

Лістинг 4.148

```
// Множення вектора на число з виділенням
// нової пам'яті для збереження результату
const WrappedVector& WrappedVector:: operator*(const double x) const {
   WrappedVector* res=new WrappedVector;
   for(size_t i=0; i<size(); i++)
      res._v[i]=_v[i]*x;
   return *res;
}
```

До цієї функції можна пред'явити претензії в неекономному використанні пам'яті. Більш-менш безпечною видається ситуація явного приймання результату якимсь іменованим об'єктом, оскільки тут виділену в функції пам'ять вдається вивільнити:

Лістинг 4.149

```
WrappedVector v=u*2; delete& v;
```

Утім хто відповідатиме за видалення зайвої пам'яті, виділеної, наприклад, унаслідок виконання команди виведення результату подвоєння вектора

```
cout<<u*2<<endl;
```

якого не прийняла жодна змінна чи указник?

Головне правило виділення пам'яті для концентрації відповідальності вимагає зосередити створення та видалення об'єкта в межах однієї програмної конструкції. Щойно була спроба порушити це правило, адже всередині операції множення є виклик функції **new**, але немає відповідної їй функції **delete**. Результат виконання цієї операції може стати сміттям, якого неможливо позбутися.

Якщо пам'ять виділяється прикладною програмою, то за її звільнення відповідає сама ця програма. Тому доводиться застосовувати стандартний механізм повернення результату у вигляді копії, створеної системою програмування, покладаючи на неї відповідальність за своєчасне звільнення використаної пам'яті:

Лістинг 4.150

```
// Множення вектора на число з поверненням
// результату у вигляді копії
const WrappedVector WrappedVector:: operator*(const double x) const {
   WrappedVector res;
   for(int i=0; i<size(); i++)
      res._v[i]=_v[i]*x;
```

```
   return res;
}
```

Вправа 4.36

Проаналізуйте проблеми, що виникають, якщо спробувати використати наведену далі реалізацію операції додавання векторів, зокрема для обчислення виразу u+v+w:

Лістинг 4.151

```
// Помилкова версія!!!
// Операція додавання векторів (член класу) з
// виділенням нової пам'яті для збереження результату
const WrappedVector& WrappedVector::
      operator+(const WrappedVector& u) const {
   WrappedVector* res=new WrappedVector;
   for(size_t i=0; i<size(); i++)
      res->_v[i]=_v[i]+u._v[i];
   return *res;
}
```

Отож, незважаючи на неефективність цього методу, знову акуратно копіюємо результат:

Лістинг 4.152

```
// Операція додавання векторів (член класу)
// з копіюванням результату
const WrappedVector WrappedVector::
      operator+(const WrappedVector& u) const {
   WrappedVector res;
   for(size_t i=0; i<size(); i++)
      res._v[i]=_v[i]+u._v[i];
   return res;
}
```

Поки що ми не розглядали оператори введення та виведення. Їх зазвичай реалізують як утиліти (поясніть, чому), але для цього бракує методів доступу до елементів вектора. Саме ними займемося зараз, тим паче що без функцій доступу вектор все одно був би неповноцінним.

У наступній версії загорнутого вектора не повторюватимемо всіх операцій, заданих для нього в попередніх версіях, залишаючи це читачеві для самостійної роботи. Зосередимося натомість на функціях доступу до елементів вектора – *операторах індексування* (subscript operator). Вони дадуть змогу запрограмувати у вигляді утиліт деякі корисні операції, серед яких виділимо введення-виведення, порівняння, додавання векторів і скалярне множення:

```
// Клас загорнутих векторів з операторами індексування      Лістинг 4.153
class WrappedVector {
private:
   static const size_t _n; double* _v;
public:
// Клас обробки аварійних ситуацій
   class BadVector;
// Стандартний набір із конструктора, деструктора, копіювань
   WrappedVector(); WrappedVector(const WrappedVector&);
   WrappedVector& operator=(const WrappedVector&);
   ~WrappedVector();
// Селектор розмірності
   size_t size() const {return _n;}
// Оператори індексування для читання елемента вектора
   const double& operator[](const size_t) const;
// для модифікації елемента вектора
   double& operator[](size_t);
};
// Утиліти введення-виведення
ostream& operator<<(ostream&, const WrappedVector&);
istream& operator>>(istream&, WrappedVector&);
// Утиліти порівняння
bool operator==(const WrappedVector&, const WrappedVector&);
bool operator!=(const WrappedVector&, const WrappedVector&);
// Утиліти множення вектора на число
const WrappedVector operator*(const WrappedVector& u, const double x);
const WrappedVector operator*(const double x, const WrappedVector& u);
// Скалярний добуток векторів
double operator*(const WrappedVector&, const WrappedVector&);
// Додавання векторів
const WrappedVector operator+
                    (const WrappedVector&, const WrappedVector&);
```

Окремо не будемо займатися побудовою класу `BadVector`. Зробіть це самостійно, використовуючи як зразок клас `BadString` із лістингу 4.62.

Насправді операторів індексування два: перший із них – селектор, другий – селектор-модифікатор. У реалізації операторів перевіряється умова виходу індексу за межі вектора зі створенням відповідної аварійної ситуації. Перевірка коректності індексів – іще одна перевага програмованих операцій; для стандартних масивів таку перевірку виконувати складніше.

Як і для інших операцій, можливі два варіанти застосування оператора індексування:

- повна форма `u.operator[](i)`,
- скорочена `u[i]`,

де `u` – вектор, `i` – індекс.

У наведених далі реалізаціях немає нічого складного, але потрібно добре розуміти, що програмовану операцію індексування векторів реалізовано за допомогою стандартної операції індексування масивів, яку застосовано в кожному з операторів виходу **return**. До речі, у цих операторах можна було б скористатись адресною арифметикою (поясніть, як).

Лістинг 4.154

```
// Оператор індексування (член класу) -
// селектор елемента масиву
const double& WrappedVector::operator[](const size_t i) const {
   if(_n<=i)
      throw BadVector("Bad index: ", i);
   return _v[i];
}
// Оператор індексування (член класу) -
// селектор-модифікатор елемента масиву
double& WrappedVector::operator[](const size_t i) {
   if(_n<=i)
      throw BadVector("Bad index: ", i);
   return _v[i];
}
```

Вправа 4.37

Доповніть клас вектора класом для обробки аварійних ситуацій `WrappedVector::BadVector` і запрограмуйте реакцію на вихід індексу за допустимі межі.

Тепер можна реалізувати операції як утиліти, вживаючи програмовані операції індексування як селектори і, за необхідності, як модифікатори. Наводимо реалізації операцій без коментарів.

Лістинг 4.155

```
// Утиліта перевірки на рівність
bool operator== (const WrappedVector& u, const WrappedVector& v) {
   bool res=true;
   for(size_t i=0; i<u.size(); i++)
      res=res&&(u[i]==v[i]);
   return res;
}
// Утиліта перевірки на нерівність
bool operator!= (const WrappedVector& u, const WrappedVector& v) {
```

```
   return !(u==v);
}
// Утиліта скалярного множення
double operator* (const WrappedVector& u, const WrappedVector& v) {
   double res=0;
   for(size_t i=0; i<u.size(); i++)
      res+=u[i]*v[i];
   return res;
}
// Утиліта множення числа на вектор
const WrappedVector operator* (const double x,const WrappedVector& u) {
   WrappedVector res;
   for(size_t i=0; i<u.size(); i++)
      res[i]=u[i]*x;
   return res;
}
// Утиліта додавання векторів
const WrappedVector operator+
      (const WrappedVector& u, const WrappedVector& v) {
   WrappedVector res;
   for(size_t i=0; i<u.size(); i++)
      res[i]=u[i]+v[i];
   return res;
}
```

Додамо до них іще дві операції – введення та виведення:

Лістинг 4.156

```
// Утиліта виведення вектора
ostream& operator<<(ostream& os, const WrappedVector& u) {
   char ch='(';
   for(size_t i=0; i<u.size(); i++) {
      os<<ch<<u[i]; ch=' ';
   }
   os<<')'<<endl;
   return os;
}
// Утиліта введення вектора
istream& operator>>(istream& is, WrappedVector& u) {
   cout<<u.size()<<':'<<endl;
   for(size_t i=0; i<u.size(); i++) {
      cout<<i<<':'; is>>u[i];
   }
   return is;
}
```

Вправа 4.38

Складіть тестову програму та перевірте, чи правильно реалізовано операції над векторами з урахуванням їхніх властивостей.

Повернімося до класу String, поширивши на нього деякі нові операції крім розглянутих раніше:

Лістинг 4.157

```
// Клас рядків із присвоєннями, індексуванням,
// порівнянням і конкатенацією
class String {
private:
   size_t _len;         // Довжина рядка
   char* _allocator;    // Сам рядок
public:
   class BadString;     // Клас обробки аварійних ситуацій
   String();            // Конструктор порожнього рядка
   String(const char);  // Конструктор за окремим символом
   String(const char*); // Конструктор С++-рядка за С-рядком
   // Конструктор мультикопіювання
   String(const String& s, const unsigned int multiplayer=1);
   ~String();          // Деструктор
// Набір операторів присвоєння
   String& operator=(const String&);  // рядка рядку
   String& operator=(const char*);    // С-рядка рядку
   String& operator=(const char);     // символу – рядку
   size_t length() const {return _len;} // Селектор довжини рядка
   bool empty() const {return _len==0;} // Перевірка, чи рядок порожній
   void clear() {*this=String();}       // Очищення рядка
// Оператори індексування
   char& operator[](const size_t);           // для модифікації символу
   const char& operator[](const size_t) const; // для читання символу
// Оператори порівняння (рівність-нерівність)
   bool operator==(const String&) const;
   bool operator!=(const String&) const;
// Оператори лексикографічного порівняння
   bool operator<(const String&) const;
   bool operator<=(const String&) const;
   bool operator>(const String&) const;
   bool operator>=(const String&) const;
// Оператори конкатенації
   String operator+(const String&) const;
// Присвоєння конкатенації
```

```
   String& operator+=(const String&);
// Присвоєння конкатенації із С-рядком
   String& operator+=(const char*);
};
// Утиліта виведення
ostream& operator<<(ostream&, const String&);
```

Рядки символів із присвоєнням було розглянуто раніше (див. лістинг 4.120). Пригадавши конвертувальні властивості конструкторів, можна зауважити, що за допомогою стандартного копіювального присвоєння можна присвоїти рядку як окремий символ, так і С-рядок – наприклад, так:

Лістинг 4.158

```
String source; char a='a';
source=a;               // source=String(a);
char* ps="Take me";
source=ps;              // source=String(ps);
```

У коментарях наведено неявні перетворення типів, які згенерує компілятор.

Для підвищення ефективності клас `String` доповнено ще двома версіями оператора присвоєння – присвоєння за сталим символьним указником (можливо, літералом) і присвоєння символу. Вони не потребують попереднього перетворення правої частини оператора присвоєння на екземпляр класу `String`, як для звичайного оператора присвоєння, а тому економніше використовують ресурси. Спочатку розглянемо складніший оператор:

Лістинг 4.159

```
// Оператор присвоєння С-рядка рядку
String& String::operator=(const char* ps) {
   if(ps==0)                       // Якщо С-рядок невизначений,
      return *this;                // то виходимо, не змінюючи this
   delete [] _allocator;           // Видаляємо старий рядок
   for(_len=0; ps[_len]; _len++); // Знаходимо нове значення довжини
    _allocator=new char[_len+1];  // Виділяємо місце для нового значення
   strcpy(_allocator, ps);         // Копіюємо рядок на нове місце
   return *this;
}
```

Вправа 4.39

Проаналізуйте доцільність створення аварійної ситуації:

Лістинг 4.160

```
if(ps==0)
   throw BadString("Attempt to use not defined pointer");
```

Зауважимо, що оператор присвоєння С-рядка не можна застосовувати до окремого символу. Справді, присвоєння

```
char c = '?'; String s; s = &c;
```

призведе до непередбаченого результату (поясніть, чому).

Із цієї причини присвоєння осібного символу реалізуємо окремо:

Лістинг 4.161

```
// Оператор присвоєння символу рядку
String& String::operator=(const char c) {
   delete [] _allocator;   // Видаляємо старий рядок
   _len=1;                 // Присвоюємо нове значення довжині
   _allocator=new char[2]; // Виділяємо місце для нового значення
   _allocator[0]=c;        // Копіюємо рядок на нове місце
   _allocator[1]='\0';
   return *this;
}
```

Гадаємо, що зрозуміло, для чого призначені методи `length()`, `empty()` та `clear()` у визначенні класу `String`.

Вправа 4.40

Протестуйте методи `length()`, `empty()` та `clear()`. Обґрунтуйте їхні сигнатури. Запропонуйте ефективнішу реалізацію методу `clear()`.

За допомогою операторів індексування можна застосовувати до рядків класу `String` таку саму операцію доступу за індексом, як і до масиву символів:

Лістинг 4.162

```
// Оператор індексування для модифікації символу
char& String::operator[](const size_t i) {
   if(_len<=i) throw BadString("Bad index: ", i);
   return _allocator[i];
};
```

Знову ж таки, лише цієї операції недостатньо. Спробуємо реалізувати операцію виведення рядків, наділивши клас лише однією наведеною операцією індексування:

Лістинг 4.163

```
// Утиліта виведення
ostream& operator<<(ostream& os, const String& s) {
   for(size_t i=0; i<s.length(); i++) os<<s[i];
   return os;
}
```

Компілятор не дозволить застосувати операцію індексування, реалізовану у лістингу 4.163, до сталого рядка. Тому реалізуємо ще одну операцію індексування, яка, власне, виконуватиме функцію селектора, тоді як попередня була також і модифікатором:

Лістинг 4.164

```
// Оператор індексування для читання символу
const char& String::operator[](const size_t i) const {
   if(_len<=i) throw BadString("Bad index: ", i);
   return _allocator[i];
}
```

Коротко прокоментуємо реалізацію ще деяких методів.

Перевірку на рівність виконують за допомогою поелементного порівняння символів. У разі виявлення першої ж розбіжності виконання циклу закінчується хибою.

Лістинг 4.165

```
// Оператор перевірки на рівність
bool String::operator==(const String& s) const {
// Якщо довжини не однакові, то рядки не рівні
   bool equal=_len==s._len;
   for(int i=0; (i<_len)&&equal; i++)
      equal=*(_allocator+i)==*(s._allocator+i);
   return equal;
}
```

Нерівність зазвичай реалізують за допомогою заперечення рівності:

Лістинг 4.166

```
// Оператор перевірки на нерівність
bool String::operator!=(const String& s) const {
   return !(*this==s);
}
```

Наведемо також реалізацію двох операцій лексикографічного порівняння:

Лістинг 4.167

```
// Перевірка, чи *this < s
bool String::operator<(const String& s) const {
// Припускаємо істинність
   bool res=true; int i=0;
// Доки не порушиться істинність і не вийдемо за межі одного з рядків
   while(res&&(i<_len)&&(i<s._len)) {
      res=res&&(*(_allocator+i)==*(s._allocator+i)); i++;
   }
// Якщо всі попередні символи збігалися, то коротший рядок – «менший»
```

```
   if(res) return _len<s._len; i--;
// Якщо знайшли першу розбіжність, то «менший» символ
// відповідає «меншому» рядку
   return *(_allocator+i)<*(s._allocator+i);
}
```

Нестроге лексикографічне порівняння можна одержати, скомбінувавши строге порівняння та рівність:

Лістинг 4.168

```
// Перевірка, чи *this <= s
bool String::operator<=(const String& s) const {
   if(*this==s) return true;
   return *this<s;
}
```

Вправа 4.41

Реалізуйте операцію `<=` безпосередньо, не використовуючи два інші порівняння.

Вправа 4.42

Запропонуйте реалізацію двох інших операцій лексикографічного порівняння.

Вправа 4.43

Створіть нову версію класу рядків, у якій операції порівняння було б реалізовано утилітами. Порівняйте два визначення класу. Який із них доцільніше використовувати? За якими критеріями?

Далі розглянемо операцію дописування (конкатенації) рядків. У ній суттєво використано мультиконструктор копіювання, щоб виділити місце для результату. Як і для додавання векторів (лістинг 4.155), знову доведеться копіювати результат:

Лістинг 4.169

```
// Оператор конкатенації, пряма реалізація
String String::operator+(const String& s) const {
   String chr('.'); // Рядок-наповнювач
   size_t len=_len+s._len; // Довжина результату конкатенації
// Мультикопіювання для резервування місця під результат конкатенації
   String res(chr, len);
// Переміщення першого рядка на початок зарезервованого місця
   strcpy(res._allocator, _allocator);
// Дописування другого рядка в кінець зарезервованого місця
   strcpy(res._allocator+_len, s._allocator);
```

```
   return res; // Вихід із передаванням копії результату
}
```

Вправа 4.44

Доповніть конструктори рядків засобами обліку та проаналізуйте кількість створень нових рядків і копіювань наявних в операції дописування рядків.

Розглянемо тепер присвоєння, суміщене з конкатенацією. Порівнюючи тип його результату `String&` із типом результату конкатенації `String`, бачимо, що копіювання на виході з функції не буде, оскільки результат формується безпосередньо в поточному об'єкті `this`:

Лістинг 4.170

```
// Оператор присвоєння з конкатенацією
String& String::operator+=(const String& s) {
   size_t len=_len+s._len;          // Довжина результату конкатенації
   char* newAllocator=new char[len+1]; // Виділення місця під результат
   strcpy(newAllocator, _allocator);  // Переміщення першого рядка
                                      // на нове місце
   delete [] _allocator; // Вивільнення місця попереднього розташування
   _allocator=newAllocator;     // Переспрямування указника на рядок
   strcpy(_allocator+_len, s._allocator); // Дописування другого рядка
                                    // в кінець зарезервованого місця
   _len=len;                  // Запам'ятовування довжини нового рядка
   return *this;              // Вихід без копіювання результату
}
```

Суміщене присвоєння, як це продемонстровано раніше в лістингу 4.142 для класу комплексних чисел, дає змогу дуже просто реалізувати конкатенацію:

Лістинг 4.171

```
// Оператор конкатенації, реалізація за допомогою
// суміщеного присвоєння
String String::operator+(const String& s) const {
   String res(*this);
   return res+=s;
}
```

Вправа 4.45

Порівняйте ефективність двох реалізацій оператора конкатенації (лістинги 4.169 та 4.171) стосовно створення допоміжних рядків.

Наведена далі операція суміщеного присвоєння за символьним указником, узагалі кажучи, надлишкова. Оскільки конструктор класу `String`

здатний працювати в режимі перетворювача, це присвоєння можна звести до звичайного, створивши допоміжний рядок. Проте пряма операція надійніша й ефективніша, бо використовує менше допоміжної пам'яті:

```
// Присвоєння конкатенації із С-рядком                    Лістинг 4.172
String& String::operator+=(const char* ps) {
   if(ps==0) return *this;        // Невизначений указник ігноруємо
   size_t len=strlen(ps);         // Обчислення довжини рядка ps
   char* newAllocator=new char[len+_len+1]; // Виділення місця під
                                            // результат
   strcpy(newAllocator, _allocator);  // Переміщення першого рядка
                                      // на нове місце
   delete [] _allocator; // Вивільнення місця попереднього розташування
   _allocator=newAllocator;    // Переспрямування указника на рядок
   strcpy(_allocator+_len, ps);  // Дописування другого рядка
                                 // в кінець зарезервованого місця
   _len+=len;                 // Запам'ятовування довжини нового рядка
   return *this;              // Вихід без копіювання результату
}
```

Вправа 4.46

Запропонуйте варіант класу рядків з операціями, реалізованими за допомогою утиліт. Порівняйте з реалізацією за допомогою членів класу.

Як наступний приклад розглянемо клас векторів різної розмірності, реалізацію якого виконайте самостійно.

```
// Клас векторів різної розмірності                       Лістинг 4.173
class DissimilarVector {
private:
   size_t _n;        // Розмірність вектора
   double* _v;       // Власне вектор
public:
   class BadVector;    // Клас обробки аварійних ситуацій
// Стандартний набір із конструктора, деструктора, копіювань
   explicit DissimilarVector(const size_t);
   explicit DissimilarVector (const DissimilarVector&);
   ~DissimilarVector();
   DissimilarVector& operator= (const DissimilarVector&);
// Селектор розмірності
   size_t size() const {return _n;}
// Оператори індексування для читання елемента вектора
   const double& operator[](const size_t) const;
```

```
   double& operator[](const size_t); // модифікатор елемента вектора
};
// Утиліти порівняння
bool operator==(const DissimilarVector&, const DissimilarVector&);
bool operator!=(const DissimilarVector&, const DissimilarVector&);
// Утиліти введення-виведення
ostream& operator<<(ostream&, const DissimilarVector&);
istream& operator>>(istream&, DissimilarVector&);
```

Вправа 4.47

Реалізуйте клас векторів за специфікацією, наведеною в лістингу 4.173.

Досі ми займалися бінарними операціями, але серед стандартних операцій є також унарні, наприклад операції інкременту й декременту. Їх визначено на фундаментальних типах і указниках, але ці операції можна поширити на інші класи, наприклад коли множину значень класу лінійно впорядковано. Інкремент заміняє об'єкт наступним у лінійному порядку, а декремент – попереднім.

Для прикладу розглянемо клас проміжків часу:

Лістинг 4.174

```
// Клас проміжків часу
class Time {
private:
   unsigned int _hours;    // Години
   unsigned int _minutes;  // Хвилини
   unsigned int _seconds;  // Секунди
// Функція нормалізації часу; вона службова, а тому закрита
   void normalizeTime();
public:
// Стандартний набір із конструктора, деструктора, копіювань
   Time(unsigned int s=0, unsigned int m=0, const unsigned int h=0);
   Time(const Time&); ~Time() {}
   Time& operator=(const Time&);
// Набір селекторів
   const unsigned int& hours() const {return _hours;}
   const unsigned int& minutes() const {return _minutes;}
   const unsigned int& seconds() const {return _seconds;}
// Набір модифікаторів
   void setHours(const unsigned int h);
   void setMinutes(const unsigned int m);
   void setSeconds(const unsigned int s);
};
```

```
// Інкремент префіксний
const Time& operator++(Time&);
// Інкремент постфіксний
const Time operator++(Time&, int);
// Декремент префіксний
const Time& operator--(Time&);
// Декремент постфіксний
const Time operator--(Time&, int);
// Утиліта виведення
ostream& operator<<(ostream&, const Time&);
```

Перший із конструкторів приймає параметр – час у секундах, хвилинах і годинах, або лише в секундах і хвилинах, або лише в секундах. Він же завдяки використанню значень двох останніх параметрів, заданих за умовчанням, може бути перетворювачем із цілого типу, що виражає значення часу в секундах, до часу в канонічному форматі (детальніше про перетворювачі йтиметься в наступному підрозділі).

Лістинг 4.175

```
// Конструктор за умовчанням:
// у секундах, у секундах і хвилинах, у секундах, хвилинах і годинах.
// Він же – перетворювач часу в секундах у нормальну форму
Time::Time
(const unsigned int s, const unsigned int m, const unsigned int h):
      _seconds(s), _minutes(m), _hours(h) {
   normalizeTime();
   return;
}
```

Копіювальний конструктор і оператор присвоєння звичайні:

Лістинг 4.176

```
// Копіювальний конструктор
Time::Time(const Time& t):
      _seconds(t._seconds), _minutes(t._minutes), _hours(t._hours) {
   return;
}
// Оператор присвоєння
Time& Time::operator=(const Time& t) {
   _hours=t._hours; _minutes=t._minutes; _seconds=t._seconds;
   return *this;
}
```

Закрита функція `normalizeTime()` перетворює час до канонічного вигляду, у якому секунди й хвилини містяться в інтервалі від 0 до 59:

```
// Функція нормалізації часу                                    Лістинг 4.177
void Time::normalizeTime() {
// обробляє атрибути поточного об'єкта
   _hours+=((_minutes+(_seconds/60))/60);
   _minutes=(_minutes+(_seconds/60))%60; _seconds%=60;
}
```

Селектори тут звичайні, а от модифікатори доводиться програмувати явно, бо треба перетворити час до канонічного вигляду (поясніть, чому):

```
// Модифікатори                                                 Лістинг 4.178
// Модифікатор годин
void Time::setHours(const unsigned int h) {
   _hours=h; normalizeTime();
   return;
}
// Модифікатор хвилин
void Time::setMinutes(const unsigned int m) {
   _minutes=m; normalizeTime();
   return;
}
// Модифікатор секунд
void Time::setSeconds(const unsigned int s) {
   _seconds=s; normalizeTime();
   return;
}
```

Вправа 4.48

З'ясуйте, як зміняться код і застосування модифікаторів, якщо їхні сигнатури замінити на такі:

```
// Набір модифікаторів                                          Лістинг 4.179
   Time& Time::setHours(const unsigned int h);
   Time& Time::setMinutes(const unsigned int m);
   Time& Time::setSeconds(const unsigned int s);
}
```

Як відомо, є два варіанти операцій інкременту й декременту – префіксна та постфіксна. Кожен варіант має своє визначення та навіть окрему сигнатуру. Щоб розрізняти ці форми, домовилися використовувати в постфіксній формі фіктивний аргумент так, наче застосування операції (наприклад, інкременту) мало б вигляд `t++i`. Це певна вільність із боку системи програмування, оскільки насправді такого виразу не буває, використовують лише вираз `t++`.

Префіксну форму операції інкременту реалізуємо як зазвичай:

```
// Інкремент префіксний                              Лістинг 4.180
const Time& operator++(Time& t) {
   t.setSeconds(t.seconds()+1);
   return t;
}
```

Позаяк параметр `t` передають в операцію за допомогою відсилки, він являє собою об'єкт, зовнішній щодо операції. Тому можна без копіювання повернути результат, яким стане попередньо змінене значення параметра.

Постфіксна форма має дві особливості. Перша з них несуттєва: це наявність фіктивного параметра, про який уже йшлося. Друга ж особливість зумовлена принциповими міркуваннями. У постфіксній формі доводиться повертати результат за допомогою значення, а не відсилки, хоча параметр передано за допомогою відсилки. Адже результат операції та значення параметра на виході різняться: результат повертає вхідне значення параметра до того, як буде збільшено його вихідне значення. Це призводить до створення ще одного екземпляра значення параметра, й операція повертає як результат цю незмінну копію, на яку не вплинуло збільшення вихідного значення параметра. Водночас сам оригінал параметра буде збільшено, що виявиться лише в наступних спробах його використання.

```
// Інкремент постфіксний                             Лістинг 4.181
const Time operator++(Time& t, int) {
   const Time res(t);     // Копіювання вхідного значення параметра
   t.setSeconds(t.seconds()+1); // Створення вихідного значення параметра
   return res;         // Повернення запам'ятованого вхідного значення
}
```

Порівнюючи реалізації префіксної та постфіксної операцій, доходимо висновку, що префіксна операція ефективніша за постфіксну. Справді, перша оперує одним об'єктом як для результату операції, так і для вихідного значення параметра, а друга має постійно утримувати два значення: результат операції та вихідне значення параметра. Тому для одноманітності надаватимемо перевагу префіксній формі навіть у разі програмування простих циклів:

```
for (int i=0; i<n; ++i);
```

замість

```
for (int i=0; i<n; i++);
```

маючи на увазі можливі узагальнення.

Особливо відчутним зайве копіювання стає тоді, коли потрібно застосувати операції постфіксного інкременту до об'єктів великих розмірів (розміри об'єкта залежать від структури програмованого класу).

Вправа 4.49

Розгляньте клас векторів із заданим у ньому поточним індексом. Операція інкременту зумовлює збільшення цього індексу на одиницю. Проаналізуйте результат застосування префіксного та постфіксного інкременту на прикладі функції наповнення вектора довільними значеннями та виведення результату такого наповнення, які запрограмуйте самостійно.

Лістинг 4.182

```
// Клас векторів із заданим індексом
class DissimilarVector {
private:
   size_t _n;             // Розмірність вектора
   double* _v;            // Власне вектор
   size_t _index;         // Індекс у векторі
public:
// Стандартний набір із конструктора, деструктора, копіювань
   DissimilarVector(const size_t size, const size_t index=0);
   DissimilarVector(const DissimilarVector&);
   ~DissimilarVector();
   DissimilarVector& operator= (const DissimilarVector&);
// Селектор елемента за індексом
   const double& get() const {return _v[_index];}
// Модифікатор елемента за індексом
   double& set() const {return _v[_index];}
// Повернення індексу на початок масиву
   void start() {_index=0;}
};
// Інкремент префіксний
const DissimilarVector& operator++ (DissimilarVector&);
// Інкремент постфіксний
const DissimilarVector operator++ (DissimilarVector&, int);
```

Вправа 4.50

Реалізуйте дві форми операції декременту часу.

Оператор виведення зазвичай виводить дані у заданому в ньому форматі:

Лістинг 4.183

```
// Утиліта виведення
ostream& operator<<(ostream& os, const Time& t) {
```

```
   os<<"("<<t.hours()<<":"<<t.minutes()<<":"<<t.seconds()<<")";
   return os;
}
```

Нарешті, як завжди, можна визначати арифметичні операції всередині класу:

Лістинг 4.184

```
// Клас проміжків часу з операціями-методами
class Time {
private:
   unsigned int _hours;    // Години
   unsigned int _minutes;  // Хвилини
   unsigned int _seconds;  // Секунди
   void normalizeTime();   // Функція службова, а тому закрита
public:
// Стандартний набір із конструктора, деструктора, копіювань
   Time(const unsigned int s=0, const unsigned int m=0,
        const unsigned int h=0);
   Time(const Time&); ~Time() {} Time& operator=(const Time&);
// Набір селекторів
   const unsigned int& hours() const {return _hours;}
   const unsigned int& minutes() const {return _minutes;}
   const unsigned int& seconds() const {return _seconds;}
// Набір модифікаторів
   void setHours(const unsigned int h);
   void setMinutes(const unsigned int m);
   void setSeconds(const unsigned int s);
// Операція префіксного інкременту
   const Time& operator++();
// Операція постфіксного інкременту
   const Time operator++(int);
// Операція префіксного декременту
   const Time& operator--();
// Операція постфіксного декременту
   const Time operator--(int);
// Оператор виведення
   ostream& operator>>(ostream&);
};
```

Вправа 4.51

Реалізуйте в класі `Time` вбудовані операції інкременту й декременту.

Вправа 4.52

Реалізуйте в класі `Time` вбудований оператор виведення.

4.9. Конвертори типів

Є багато причин, які зумовлюють потребу в перетворенні типів. Одна з них полягає в тому, що одну й ту саму величину можна подавати порізному, наприклад: комплексні числа – в арифметичному та тригонометричному вигляді, відстань – у метричній, британській або давньоруській системі мір тощо. Ми вже наводили приклади використання мішаних виразів, у яких є водночас цілі та дійсні чи комплексні та дійсні числа. Перетворення можливі й у рядках символів. Усе це – приклади застосування *конверторів типу* (type converter), які можуть бути викликані як явною командою в програмі, так і неявно командою, згенерованою компілятором.

Перший тип конвертора типів – це сам конструктор, якщо його можна викликати з одним параметром. Як уже було з'ясовано (підрозділи 4.6.1, 4.8, лістинг 4.158), такі конструктори можна застосовувати як явно, так і для забезпечення неявних перетворень типів об'єктів. Інколи такі перетворення зводяться до виконання кількох команд і не потребують додаткових витрат пам'яті, як, наприклад, у разі комплексних чисел; в інших же ситуаціях вони можуть бути ємнішими й вимагати виділення додаткових ресурсів, як це було з рядками символів.

Некеровані механізми можуть виявитися небезпечними, тому можна забороняти неявні перетворення, не дозволяючи неявні виклики конструкторів перетворення типів за допомогою *функціонального специфікатора явного виклику* **`explicit`**.

У класі `String` є два конструктори, які можна викликати неявно. Розглянемо їх. Конструктор `String(const char)` забезпечує перетворення окремого символу на рядок – наприклад, у таких ситуаціях:

Лістинг 4.185

```
String str='a';
// Неявні виклики конструктора String(const char);
cout<<(str=='a')<<endl;          // Результат - істина
cout<<str+'b'<<endl;             // Результат - рядок "ab"
cout<<(str+='c')<<endl;          // Результат - рядок "ac"
```

Схоже перетворення могло б відбуватися також у разі присвоєння `str='d'`, якби визначенням класу не було передбачено окремої версії присвоєння, що виконується явно без попереднього виклику конструктора. Якщо ж застосувати функціональний специфікатор **`explicit`**, який заборонить неявні виклики конструктора

```
explicit String(const char);
```

то всі спроби неявного перетворення стануть незаконними. Коли замість рядка знадобиться використати символ, його доведеться попередньо перетворити на рядок за допомогою явного виклику конструктора – наприклад, так:

```
str==String('a');
```

З іншим конструктором –

```
String(const char*);
```

ситуація аналогічна.

Ось інші приклади неявних викликів конструктора:

Лістинг 4.186

```
// Неявні виклики конструктора String(const char*);
String ss("That is a string");
cout<<(ss=="That is a string")<<endl; cout<<ss+", indeed."<<endl;
```

Водночас кожне з присвоєнь

Лістинг 4.187

```
ss="That is another string";
cout<<(ss+=", indeed.")<<endl;
```

виконується безпосередньо.

Вправа 4.53

За допомогою налагоджувача простежте за викликами конструкторів і присвоєнь у кодах з лістингів 4.185 і 4.186.

Знову ж, використання специфікатора **explicit** при конструкторі String(**const char***) заборонить неявні перетворення.

Проектуючи класи, слід уважно стежити за втратами ефективності, яку спричиняють конструктори-перетворювачі. Тому слід шукати розумний компроміс між застосуванням неявних перетворень і потребою визначати додаткові операції. Для рядків символів можна заборонити неявний виклик конструктора з параметром – символьним указником, але дозволити неявні перетворення окремого символу на рядок символів.

Вправа 4.54

Побудуйте варіанти класу String, які б дозволяли чи забороняли неявні виклики конструкторів. Спроектуйте відповідний набір додаткових операцій у разі заборони.

Другий тип конвертора типів – програмований оператор перетворення типів. Почнемо з варіанта простого прикладу вимірювання часових інте-

рвалів і пригадаємо клас часу `Time` із попереднього підрозділу, доповнивши його оператором перетворення часу на ціле число:

Лістинг 4.188

```
// Клас проміжків часу з операціями-методами
class Time {
   unsigned int _hours;      // Години
   unsigned int _minutes;    // Хвилини
   unsigned int _seconds;    // Секунди
   void normalizeTime();    // Функція нормалізації часу;
                            // вона службова, а тому закрита
public:
// Стандартний набір ізконструктора, деструктора, копіювань
   Time(const unsigned int s=0, const unsigned int m=0,
        const unsigned int h=0);
   Time(const Time&); ~Time() {} Time& operator=(const Time&);
// Оператор перетворення часу на ціле число
   operator unsigned int() const;
// Набір селекторів
   const unsigned int& hours() const {return _hours;}
   const unsigned int& minutes() const {return _minutes;}
   const unsigned int& seconds() const {return _seconds;}
// Набір модифікаторів
   void setHours(const unsigned int& h);
   void setMinutes(const unsigned int& m);
   void setSeconds(const unsigned int& s);
};
```

Особливу роль у цьому варіанті класу відіграє новий для нього *оператор перетворення часу на* ціле число, яке позначатиме час, виражений у секундах:

Лістинг 4.189

```
// Оператор перетворення часу на ціле число
Time::operator unsigned int() const {
   return _hours*3600+_minutes*60+_seconds;
}
```

Спосіб запису викликів цього оператора трохи незвичний. Оператор перетворення унарний: у нього один аргумент – об'єкт, до якого застосовано оператор. Можливі кілька варіантів його виклику.

Лістинг 4.190

```
Time t(1,2,3);
cout<<t.operator unsigned int()<<endl; // Повний запис виклику
cout<< unsigned int(t)<<endl; // Скорочений запис виклику
```

```
// Явний запис виклику за допомогою операції перетворення типів
cout<<static_cast<unsigned int>(t)<<endl;
```

Як було вже згадано, останній з явних варіантів найбільш вдалий, і його рекомендовано вживати (це ще одне правило гарного тону). Інші способи запису наведено тому, що доводиться читати програми, складені без дотримання правил хорошого тону.

Проте крім явних викликів можуть бути й неявні. Наприклад, тут цілу змінну ініціалізовано значенням часу:

```
unsigned int tint=t; // Неявне перетворення під час ініціалізації
```

Якби оператора перетворення не було, то такий запис був би синтаксичною помилкою. Коли ж оператор перетворення часу до цілого типу є, компілятор згенерує його виклик – і матимемо коректний код.

Те саме станеться й у разі спроби вивести значення часу, якщо немає спеціальної утиліти виведення. Компілятор знає, як виводити цілі числа, має спосіб відповідного перетворення, а тому застосує його до такого коду:

```
cout<<t<<endl; // Неявне перетворення під час виведення
```

Вправа 4.55

За допомогою налагоджувача простежте за викликами перетворень у кодах із лістингу 4.190.

Подивимось, як перетворення типу використовують у програмуванні іншого оператора, а саме оператора множення, що збільшує часовий проміжок у задану кількість разів:

Лістинг 4.191

```
// Множення часу на число
const Time operator*(const int mult, const Time& t) {
   return Time(mult*(int)t);
}
```

Вправа 4.56

Поясніть, що станеться, якщо в операторі виходу

```
return Time(mult*t);
```

видалити явне перетворення до цілого типу.

Доходимо першого висновку: наявність конвертора типу у вигляді оператора перетворення з програмованого типу (у нашому прикладі – часу) до стандартного (цілого) дає змогу звести операції над програмованим типом до операцій над стандартним. Це економить зусилля на про-

грамування операцій, утім такі перетворення надають компілятору більше свободи й часто роблять програму не такою зрозумілою чи навіть спричиняють несподівані ефекти. Тому потрібно бути абсолютно впевненим, яке саме перетворення в якому разі буде застосовано.

Розглянемо тестовий приклад. Визначимо об'єкт класу часу

```
Time t(100);
```

Припустімо, що в цьому класі є оператор виведення (див., наприклад, оператор з лістингу 4.184). Застосуємо його до створеного об'єкта

```
cout<<t<<endl; // 1. Виводимо час
```

й одержимо результат у форматі часу

```
(0:1:40)
```

Тепер виведемо суму часів:

```
cout<<t+t<<endl; // 2. Виводимо ціле число
```

Оскільки в цьому класі не визначено операцію додавання часів, відбудеться неявне перетворення кожного часу до цілого числа, додавання цілих чисел і виведення цілого результату `200`.

Якщо застосувати явне перетворення, викликавши конструктор часу, то цілу суму буде перетворено на час і виведено як час:

```
cout<<Time(t+t)<<endl; // 3. Виводимо час
```

Неважко перевірити відповідність результатів двох виведень часу – `(0:1:40)` та `(0:3:20)`.

Самостійно подумайте, як потрібно визначити клас, щоб його було виведено й у разі застосуванні операції множення часу на число

```
cout<<2*t<<endl; // 3. Виводимо час
```

з тим самим результатом `(0:3:20)`.

Висновок. *Перетворення з програмованого типу, тобто класу, у стандартний може виконувати тільки оператор перетворення, а зі стандартного типу в програмований – конструктор перетворення.*

Можна заборонити неявні перетворення. Щоб зменшити ризик виникнення несподіваних ситуацій під час виконання програми, оператор перетворення замінюють на функцію, яку завжди буде викликано явно:

Лістинг 4.192

```
// Функція перетворення часу на ціле число
unsigned int Time::toInt() const {
```

```
   return _hours*3600+_minutes*60+_seconds;
}
```

Перетворення типів в іншому напрямку можна заборонити за допомогою специфікатора **explicit**.

Вправа 4.57

Визначте арифметичні операції для класу часу та проаналізуйте їх виконання з лише явними перетвореннями цілого типу на час і навпаки.

Перетворення типів тісно пов'язані зі способами визначення операцій, наприклад арифметичних. Розглянемо, скажімо, операцію додавання часу, наведену в попередніх лістингах. Її було реалізовано за допомогою операції додавання цілих чисел із перетворенням одиниць виміру часу на секунди. Результат так само виражено в секундах.

Розглянемо інший приклад. Доповнимо клас `Time` можливістю подавати час у годинах, що потрібно, скажімо, для обчислення робочого часу в розрахунках погодинної оплати. У такій задачі, природно, треба додавати часові інтервали. Якщо в класі `Time` буде лише операція перетворення до цілого типу, то додавання виконуватиметься з попереднім перерахунком часу в секундах, що можна вважати зайвим для нашої задачі. Тому визначимо особливу операцію додавання

```
double Time::operator+(const Time&) const;
```

яка за двома часовими доданками визначає їх дійсну суму – час у годинах. Проте, як виявляється, ця задача не така проста.

Візьмемо, для прикладу, такі об'єкти:

```
Time t1(30, 20, 1), t2(40, 50, 7), t3(1000), t4(0,0,1);
```

Розглянемо різні варіанти розставляння дужок для обчислення суми `t1+t2+t3+t4`. Спочатку додамо перші два значення:

```
t1+t2;   // double Time::operator+(const Time&) const;
```

Операція виконується без проблем, проте в результаті одержимо дійсне число. Якщо не вжити відповідних заходів, то далі воно буде заокруглене до цілого, а потім використане в операції додавання цілих чисел із попереднім перетворенням одиниць виміру часу на секунди:

```
((t1+t2)+t3)+t4;          // (t1+t2)+int(t3)+int(t4)
```

Так сталося тому, що бракувало утиліти додавання часу до числа

```
double operator+(const double, const Time&);
```

Із нею попередня операція виконується правильно:

```
((t1+t2)+t3)+t4;      // double operator+(const double, const Time&);
```

Проте виникає проблема в іншому разі, коли до часу додають число:

```
t1+(t2+t3)+t4;        // int(t1)+(t2+ t3)+int(t4)
```

Виявляється, потрібна ще одна операція – додавання дійсного числа до часу:

```
double Time::operator+(const double) const;
```

Зауважимо, що стандартну операцію додавання теж буде використано, наприклад, для

```
(t1+t2)+(t3+t4);
```

Тепер наведемо визначення наступної версії класу `Time`:

Лістинг 4.193

```
// Клас проміжків часу, призначений для
// розрахунків у годинах
class Time {
private:
   unsigned int _hours;    // Години
   unsigned int _minutes;  // Хвилини
   unsigned int _seconds;  // Секунди
   void normalizeTime();   // Функція службова, а тому закрита
public:
// Стандартний набір із конструктора, деструктора, копіювань
   Time(const unsigned int s=0, const unsigned int m=0,
        const unsigned int h=0);
   Time(const Time&); ~Time() {}; Time& operator=(const Time&);
// Оператор перетворення часу на ціле число
   operator unsigned int() const;
// Селектори
   const unsigned int& hours() const {return _hours;}
   const unsigned int& minutes() const {return _minutes;}
   const unsigned int& seconds() const {return _seconds;}
// Набір модифікаторів
   void setHours(const unsigned int h);
   void setMinutes(const unsigned int m);
   void setSeconds(const unsigned int s);
// Операції додавання
```

```
   double operator+(const Time&) const;
   double operator+(const double) const;
};
// Утиліта додавання
double operator+(const double, const Time&);
Time operator*(const double, const Time&);
ostream& operator<<(ostream&, const Time&);
```

Константу, потрібну для перетворення секунд на години, ініціалізуємо очевидним значенням:

```
// Константа для перетворення секунд на години
const double Time::hourToDec=1./3600.;
```

Основну операцію додавання реалізуємо за допомогою перетворення до цілого типу:

Лістинг 4.194

```
// Операція додавання часових інтервалів
double Time::operator+(const Time& t) const {
   return int(*this)*hourToDec+int(t)*hourToDec;
}
```

Наведемо реалізації додаткових операцій. Вони очевидні:

Лістинг 4.195

```
// Операція додавання числа до часу
double Time::operator+(const double x) const {
   return int(*this)*hourToDec+x;
}
// Операція додавання часу до числа
double operator+(const double x, const Time& t) {
   return x+int(t)*Time::hourToDec;
}
```

Тепер можна остаточно з'ясувати, як конвертори пов'язані з програмованими операціями. Зазначимо, що в кожному класі конверторів може бути більше ніж один. Так, доповнивши клас `Time` конвертором до дійсного типу **`operator double() const`**, можна взагалі обійтися без програмованих операцій додавання, завжди застосовуючи стандартну операцію:

Лістинг 4.196

```
// Клас проміжків часу,
// наділений конверторами як у секунди int(), так і в години double()
class Time {
private:
   unsigned int _hours; // Години
```

```
   unsigned int _minutes; // Хвилини
   unsigned int _seconds; // Секунди
   void normalizeTime(); // Функція службова, а тому закрита
public:
// Стандартний набір із конструктора, деструктора, копіювань
   Time(const unsigned int s=0, const unsigned int m=0,
        const unsigned int h=0);
   Time(const Time&); ~Time() {} Time& operator=(const Time&);
// Оператор перетворення часу на ціле число
   operator int() const;
// Оператор перетворення часу на дійсне число
   operator double() const;
// Селектори
   const unsigned int& hours() const {return _hours;}
   const unsigned int& minutes() const {return _minutes;}
   const unsigned int& seconds() const {return _seconds;}
// Набір модифікаторів
   void setHours(const unsigned int h);
   void setMinutes(const unsigned int m);
   void setSeconds(const unsigned int s);
};
Time operator*(const double, const Time&);
ostream& operator<<(ostream&, const Time&);
```

Вправа 4.58

Реалізуйте оператор перетворення часу до дійсного типу.

Тепер стають зайвими додаткові визначення додавань, але виникає потреба явно вживати перетворення типів у програмі, бо з'явиться неоднозначність (поясніть, чому):

Лістинг 4.197

```
Time t1(30,20,1), t2(40,50,7), t3(1000), t4(0,0,1);
cout<<static_cast<double>(t1)+static_cast<double>(t2)+
   static_cast<double>(t3)+static_cast<double>(t4)<<endl;
```

Вправа 4.59

Запропонуйте варіант класу часу, наділений конвертором до дійсного типу без конвертора до цілого типу. Проаналізуйте виконання арифметичних операцій.

Інший приклад співіснування різних конверторів пов'язаний із рядками символів. Для них уже можна назвати три подання: стандартне для мови С у вигляді символьного указника чи масиву, подання за допомо-

гою класу string зі стандартної бібліотеки STL, а також за допомогою створеного раніше класу String. Якщо в класі String визначити перетворення типів у вигляді як операторів, так і конструкторів, то застосування багатьох операцій, визначених водночас над різними типами символів, зіткнеться з проблемою неоднозначності щодо того, які перетворення й операції застосовувати:

Лістинг 4.198

```
// Клас рядків з усіма можливими конверторами
// в обидва боки: у клас і з класу
class String {
private:
   size_t _len; // Довжина рядка
   char* _allocator; // Сам рядок
public:
   String();                  // Конструктор порожнього рядка
   // Конструктор і конвертор char* до String
   explicit String(const char*);
   String(const char);       // Конструктор і конвертор char до String
// Конструктор і конвертор string до String
   explicit String(const string&);
// Конструктор мультикопіювання
   String(const String& s, const int multiplayer=1);
   ~String();                       // Деструктор
// Оператори перетворення типів String до string
   operator string() const;
// Дуже небезпечний String до char*
   operator char* () const;
// const String до const char*
   const char* c_str() const {return _allocator;}
   size_t length() const {return _len;}
   bool empty() const {return _len==0;}
   void clear() {*this=String();}
   char& operator[](const size_t);
   const char& operator[](const size_t) const;
   bool operator==(const String&) const;
   bool operator!=(const String&) const;
   String& operator=(const String&);
   String& operator=(const char*);
   String& operator=(const char);
   String operator+(const String&) const;
   String& operator+=(const String&);
   String& operator+=(const char*);
};
```

Наведемо реалізації конструктора й оператора перетворення до типу `string` із бібліотеки STL:

Лістинг 4.199

```
// Конструктор і конвертор string до String
String::String(const string& s):
   _allocator(new char[s.length()+1]), _len(s.length()) {
   strcpy(_allocator, s.c_str());
   return;
}
// Оператор перетворення типу String до string
const string String:: string() const {
   return string(_allocator);
}
```

А ось такий «сталий» оператор повністю порушив би інкапсуляцію:

Лістинг 4.200

```
// Невиправдано небезпечний оператор перетворення
// надає прямий доступ до місця розташування рядка
String::operator char* () const {
   return _allocator;
}
```

Він, на перший погляд, не змінює об'єкт, але насправді робить його доступним для неконтрольованих змін:

Лістинг 4.201

```
// Створюємо рядок
String str("This string will be corrupted");
// Визначаємо символьний указник
char* ch=0;
// Начебто перетворюємо рядок str на C-рядок
ch=str;
// а насправді відкриваємо неконтрольований доступ
for (size_t i=0; i<strlen(ch); ++i)
   ch[i]='?';
// На що перетвориться рядок str?
```

Варто додати ще один рівень сталості:

Лістинг 4.202

```
// Місце розташування рядка доступне лише для читання
String::operator const char* () const {
   return _allocator;
}
```

Вправа 4.60

Реалізуйте оператори перетворень і проведіть випробування класу з лістингу 4.198. Проаналізуйте перетворення з погляду контролю над дос-

тупом до оригіналу. Наведіть приклади як неявних перетворень типів, так і випадків, коли перетворення доводиться задавати явно.

Надлишок перетворювачів типів призводить до надто вільного стилю програмування, часом із несподіваними наслідками та додатковими затратами. Оскільки створення нових рядків завжди пов'язане з виділенням вільної пам'яті, варто за допомогою специфікатора **explicit** заборонити конвертувальну дію конструкторів. Так само потрібно вчинити й із перетвореннями до стандартних типів рядків. Тепер усі перетворення мають бути виписані в програмі явно за допомогою відповідного конструктора чи особливої функції перетворення, яку, на відміну від операторів, компілятор не викликає неявно:

Лістинг 4.203

```
// Клас рядків без конверторів:
// перетворення як у клас, так і з класу запрограмовано явно
class String {
private:
   size_t _len; // Довжина рядка
   char* _allocator; // Сам рядок
public:
   String();  // Конструктор порожнього рядка
// Явні конструктори
   explicit String(const char*); explicit String(const char);
   explicit String(const string&);
// Конструктор мультикопіювання
   String(const String& s, const unsigned int multiplayer=1);
   ~String(); // Деструктор
// Функція явного перетворення до string
   const string to_string() const;
// Функція явного перетворення до const char*
   const char* c_str() const {return _allocator;}
   size_t length() const {return _len;}
   bool empty() const {return _len==0;}
   void clear() {*this=String();}
   char& operator[](const size_t);
   const char& operator[](const size_t) const;
   bool operator==(const String&) const;
   bool operator!=(const String&) const;
   String& operator=(const String&);
   String& operator=(const char*);
   String& operator=(const char);
   String operator+(const String&) const;
   String& operator+=(const String&);
```

```
   String& operator+=(const char*);
};
```

Вправа 4.61

Самостійно реалізуйте й протестуйте клас `String` з лістингу 4.203.

Повернемося до комплексних чисел, які тепер розглянемо у двох поданнях: алгебричному і тригонометричному. Одні арифметичні операції легше виконувати в одному поданні, а інші – у другому. Неявні перетворення типів дають компілятору змогу автоматично вибирати придатніше подання для кожної з операцій.

Перетворення форматів комплексних чисел почнемо з комплексних чисел в алгебричній формі:

Лістинг 4.204

```
// Випереджальне оголошення класу комплексних чисел
// у тригонометричному вигляді
class TComplex;
// Клас комплексних чисел в алгебричному вигляді
// Арифметичні та логічні операції в класі
class Complex {
private:
   double _re; // Дійсна частина
   double _im; // Уявна частина
public:
   Complex(const double re=0, const double im=0);
   Complex(const Complex& z);
   ~Complex();
// Оператор перетворення до тригонометричного вигляду
   operator TComplex() const;
// Присвоєння
   Complex& operator=(const Complex&);
// Модифікатори
   double& re() {return _re;}
   double& im() {return _im;}
// Селектори
   const double& re() const {return _re;}
   const double& im() const {return _im;}
// Модуль і аргумент
   double mod() const {return sqrt(_re*_re+_im*_im);}
   double arg() const {return atan2(_im, _re);}
// Спряжене число
   const Complex conj() const {return Complex(_re, -_im);}
// Операції
   const Complex operator+(const Complex&) const;
```

```
    const Complex operator-(const Complex&) const;
    const Complex operator*(const Complex&) const;
    const Complex operator/(const Complex&) const;
    bool operator==(const Complex&) const;
    bool operator!=(const Complex&) const;
};
// Утиліти введення-виведення
ostream& operator<<(ostream&, const Complex&);
istream& operator>>(istream&, Complex&);
```

У класі Complex передбачено повний набір операцій. У реалізації деяких із них використано інше подання комплексних чисел, а саме тригонометричний вигляд. Оператор перетворення комплексного числа з алгебричного вигляду до тригонометричного можна записати, виразивши його через модуль і аргумент:

Лістинг 4.205

```
// Оператор перетворення до тригонометричного вигляду
Complex::operator TComplex() const {
    return TComplex(mod(), arg());
}
```

Тут TComplex() – конструктор комплексного числа в тригонометричному вигляді.

Наведемо тексти реалізації операцій, виділивши випадки використання тригонометричного вигляду комплексного числа:

Лістинг 4.206

```
// Операція додавання: алгебричний вигляд
const Complex Complex::operator+ (const Complex& a) const {
    return Complex(_re+a._re, _im+a._im);
}
// Операція віднімання: алгебричний вигляд
const Complex Complex::operator- (const Complex& a) const {
    return Complex(_re-a._re, _im-a._im);
}
// Операція множення: тригонометричний вигляд
const Complex Complex::operator* (const Complex& a) const {
    TComplex u(mod()*a.mod(), arg()+a.arg());
    return Complex(u.re(), u.im());
}
// Операція ділення: тригонометричний вигляд
const Complex Complex::operator/ (const Complex& a) const {
    TComplex u(mod()/a.mod(), arg()-a.arg());
    return Complex(u.re(), u.im());
}
```

Тепер розглянемо визначення комплексних чисел у тригонометричному вигляді:

Лістинг 4.207

```
// Випереджальне оголошення класу комплексних чисел
// в алгебричному вигляді
class Complex;
// Клас комплексних чисел у тригонометричному вигляді
// Арифметичні та логічні операції в класі
class TComplex {
private:
   double _ro; // Модуль
   double _phi; // Аргумент
public:
   TComplex(const double ro=0, const double phi=0);
   TComplex(const TComplex& z); ~TComplex();
// Оператор перетворення до алгебричного вигляду
   operator Complex() const;
// Присвоєння
   TComplex& operator=(const TComplex&);
// Модифікатори
   double& mod() {return _ro;}
   double& arg() {return _phi;}
// Селектори
   const double& mod() const {return _ro;}
   const double& arg() const {return _phi;}
// Дійсна та уявна частини
   double re() const {return _ro*cos(_phi);}
   double im() const {return _ro*sin(_phi);}
// Операції
   const TComplex operator+(const TComplex&) const;
   const TComplex operator-(const TComplex&) const;
   const TComplex operator*(const TComplex&) const;
   const TComplex operator/(const TComplex&) const;
   bool operator==(const TComplex&) const;
   bool operator!=(const TComplex&) const;
};
// Утиліта виведення
ostream& operator<<(ostream&, const TComplex&);
```

Цей клас фактично ідентичний до попереднього. Відмінність полягає лише в операторі перетворення: тепер він перетворює тригонометричний вигляд до алгебричного.

```
// Оператор перетворення до алгебричного вигляду
TComplex::operator Complex() const {
   return Complex(re(), im());
}
```
Лістинг 4.208

Окрім цього, є ще відмінності в реалізації операцій: адитивні операції реалізовано в алгебричному вигляді:

Лістинг 4.209
```
// Операція додавання: алгебричний вигляд
const TComplex TComplex::operator+ (const TComplex& a) const {
   Complex u(re()+a.re(), im()+a.im());
   return TComplex(u.mod(), u.arg());
}
// Операція віднімання: алгебричний вигляд
const TComplex TComplex::operator- (const TComplex& a) const {
   Complex u(re()-a.re(), im()-a.im());
   return TComplex(u.mod(), u.arg());
}
// Операція множення: тригонометричний вигляд
const TComplex TComplex::operator* (const TComplex& a) const {
   return TComplex(_ro*a._ro, _phi+a._phi);
}
// Операція ділення: тригонометричний вигляд
const TComplex TComplex::operator/ (const TComplex& a) const {
   return TComplex(_ro/a._ro, _phi-a._phi);
}
// Перевірка на рівність
bool TComplex::operator==(const TComplex& a) const {
   double s=sin(_phi), sa=sin(a._phi);
   double c=cos(_phi), ca=cos(a._phi);
   return _ro==a._ro&&s==sa&&c==ca;
}
// Перевірка на нерівність
bool TComplex::operator!=(const TComplex& a) const {
   return !(*this==a);
}
```

Розглянемо деякі приклади виконання операцій та прокоментуємо їх.

Лістинг 4.210
```
int main() {
Complex z1(1, 1), z2(0, 1); /* Визначено два комплексні числа
      в алгебричному вигляді */
cout<<z1+z2<<endl;   /* Результат додавання створено за допомогою
      виклику конструктора Complex(_re+a._re, _im+a._im) */
```

```
cout<<z1*z2<<endl; /* Під час виконання множення спочатку буде створено
      результат у тригонометричному вигляді TComplex u(mod()*a.mod(),
      arg()+a.arg()),якого виклик конструктора Complex(u.re(), u.im())
      потім перетворить до алгебричного*/
cout<<z1*10<<endl; /* Другий множник конструктор перетворює на
      комплексне число в алгебричному вигляді. Далі – як для
      попередньої команди*/
// cout<<10*z1<<endl; /*Неможлива операція, оскільки множення дійсного
      числа на комплексне не визначено, а перетворення першого
      аргумента методу на комплексне число не дозволено */
TComplex t(1, 2); // Визначено комплексне число в тригонометричному вигляді
cout<<z1+t<<endl; /* Додавання з класу Complex: t попередньо перетворено
      до алгебричного вигляду */
cout<<t+z1<<endl; /* Додавання з класу TComplex: z1 попередньо
      перетворено до тригонометричного вигляду, побудовано алгебричний
      вигляд результату Complex u(re()+a.re(), im()+a.im()), який
      потім перетворено до тригонометричного TComplex(u.mod(),u.arg()) */
cout<<z1*t<<endl; /* Множення з класу Complex: t попередньо перетворено
      до алгебричного вигляду, побудовано тригонометричний вигляд
      результату TComplex u(mod()*a.mod(), arg()+a.arg()), який потім
      перетворено до алгебричного Complex(u.re(), u.im()) */
cout<<t*z1<<endl; /* Множення з класу Tcomplex: z1 попередньо
      перетворено до тригонометричного вигляду */
return 0;
}
```

Зробимо висновки з наведеного прикладу. По-перше, у разі спроби помножити дійсне число на комплексне не виконується перетворення дійсного на комплексне, якщо перше – лівий операнд арифметичної операції, і таку операцію не буде відкомпільовано. По-друге, є невиправдані перетворення типів, зокрема для мішаних типів аргументів – випадки додавання в класі `TComplex` чи множення в класі `Complex`. По-третє, тут не використано всіх можливостей, які могло б надати перетворення типів, а тому визначення операцій дублювалося в кожному з класів.

Запропонуємо інший склад обох класів, замінивши методи утилітами. Почнемо з комплексних чисел в алгебричному вигляді:

Лістинг 4.211

```
// Випереджальне оголошення комплексних чисел
// у тригонометричному вигляді
class TComplex;
// Клас комплексних чисел в алгебричному вигляді
// Арифметичні та логічні операції-утиліти
```

```
class Complex {
private:
   double _re; // Дійсна частина
   double _im; // Уявна частина
public:
   Complex(const double re=0, const double im=0);
   Complex(const Complex&);
   ~Complex();
// Оператор перетворення до тригонометричного вигляду
   operator TComplex() const;
// Присвоєння
   Complex& operator=(const Complex&);
// Модифікатори
   double& re() {return _re;}
   double& im() {return _im;}
// Селектори
   const double& re() const {return _re;}
   const double& im() const {return _im;}
// Модуль і аргумент
   double mod() const {return sqrt(_re*_re+_im*_im);}
   double arg() const {return atan2(_im, _re);}
// Спряжене число
   const Complex conj() const
      {return Complex(_re, -_im);}
};
// Операції-утиліти
const Complex operator+(const Complex&, const Complex&);
const Complex operator-(const Complex&, const Complex&);
bool operator==(const Complex&, const Complex&);
bool operator!=(const Complex&, const Complex&);
// Утиліти введення-виведення
ostream& operator<<(ostream&, const Complex&);
istream& operator>>(istream&, Complex&);
```

Як бачимо, у класі комплексних чисел в алгебричному вигляді немає операцій множення та ділення, які є для тригонометричного вигляду.

У класі ж комплексних чисел у тригонометричному вигляді, навпаки, не буде додавання, віднімання та порівняння на рівність і нерівність:

Лістинг 4.212

```
// Випереджальне оголошення комплексних чисел
// в алгебричному вигляді
class Complex;
// Клас комплексних чисел у тригонометричному вигляді
```

```
// Арифметичні та логічні операції-утиліти
class TComplex {
private:
   double _ro; // Модуль
   double _phi; // Аргумент
public:
   TComplex(const double ro=0, const double phi=0);
   TComplex(const TComplex& z); ~TComplex();
// Оператор перетворення до алгебричного вигляду
   operator Complex() const;
// Присвоєння
   TComplex& operator=(const TComplex&);
// Модифікатори
   double& mod() {return _ro;}
   double& arg() {return _phi;}
// Селектори
   const double& mod() const {return _ro;}
   const double& arg() const {return _phi;}
// Дійсна та уявна частини
   double re() const {return _ro*cos(_phi);}
   double im() const {return _ro*sin(_phi);}
};
// Операції-утиліти
const TComplex operator*(const TComplex&, const TComplex&);
const TComplex operator/(const TComplex&, const TComplex&);
ostream& operator<<(ostream&, const TComplex&); // Утиліта виведення
```

Можна було б обійтись і без оператора виведення, тоді всі комплексні числа виводилися б в алгебричному вигляді.

Проаналізувавши наведену далі тестову програму (лістинг 4.213), можна побачити, що для нової версії комплексних чисел не лише виконуються всі операції з попередньої версії, зокрема з мішаними аргументами, але й стають можливими інші, які не могли бути відкомпільовані раніше.

Вправа 4.62

Реалізуйте класи комплексних чисел із лістингів 4.211 та 4.212.

Вправа 4.63

Доповніть класи комплексних чисел із лістингів 4.211 та 4.212 суміщеними присвоєннями. Арифметичні операції реалізуйте через відповідні суміщені присвоєння (див. лістинг 4.159).

```
// Тестування операцій над комплексними числами          Лістинг 4.213
int main() {
// Визначення чисел в алгебричному вигляді
   Complex z1(1, 1), z2(0, 1); cout<<"Conversion operator"<<endl;
   cout<<"Utilities as operation"<<endl;
   cout<<"Complex=========="<<endl;
// Операції над аргументами різних типів із застосуванням перетворень
   cout<<z1+z2<<endl; cout<<z1*z2<<endl; cout<<10*z1<<endl;
   cout<<z1*10<<endl; cout<<z1*z1.conj()<<endl; cout<<(z1=z2=1)<<endl;
   cout<<(z2=z1+=1)<<endl; cout<<((z1+=z2)=1)<<endl;
// Визначення чисел у тригонометричному вигляді
   TComplex t1(1, 2), t2(2, 3); cout<<"TComplex=========="<<endl;
   cout<<t1<<endl; cout<<t2<<endl;
// Операції над аргументами різних типів із застосуванням перетворень
   cout<<t1+t2<<endl; cout<<t1+10<<endl; cout<<10+t1<<endl;
   cout<<"Mixed=========="<<endl;
   cout<<z1+t1<<endl; cout<<t1+z1<<endl; cout<<z1*t1<<endl;
   cout<<t1*z1<<endl; cout<<(t1==t1)<<endl;
   cout<<(Complex(t1)==t1)<<endl; cout<<(TComplex(z1)==z1)<<endl;
   return 0;
}
```

Вправа 4.64

Виконайте тестову програму, простежте за ходом викликів за допомогою налагоджувача та підрахуйте кількість виконаних перетворень типів.

Проблему перетворення типів уважають однією з найделікатніших у програмуванні передусім через непередбачуваність або несподіваність неявних перетворень, виконуваних компілятором самостійно. З іншого боку, наявність перетворень усуває потребу в дублюванні коду одних і тих самих операцій у разі різних подань даних. Частини програми, що відповідають за перетворення типів, важливо тримати під постійним контролем.

Важливо також пам'ятати, у якому з класів зосереджено конвертори типів: у вхідному, об'єкти якого буде перетворено, чи цільовому, до якого ведуть перетворення. Оператор перетворення завжди поміщають у вхідному класі, конструктор перетворення – у цільовому. Тому, прийнявши рішення про потребу в конвертуванні, варто проаналізувати ситуацію та вибрати його тип, виходячи з двох крайніх випадків:

- якщо цільовий тип стандартний або недоступний, слід користуватись оператором перетворення;

- якщо ж стандартний або недоступний вхідний тип, то варто використовувати конструктор перетворення.

В інших випадках можливий як один, так і другий способи.

У табл. 4.1, 4.2 наведено приклади допустимих комбінацій конверторів. Для пари `Time` та **`int`** їх можна поміщати лише в класі `Time`. Для пари `Complex` і `TComplex` можливих комбінацій чотири. Перевагу варто надавати перетворенню за допомогою пари конструкторів як найпрозорішому в реалізації. Кожен рядок таблиці показує можливі комбінації конверторів для типів, зазначених у заголовках стовпців.

Таблиця 4.1. Можливі комбінації конверторів `Time` та **`int`**

class Time	**int**
operator int() **const**; Time (**const int**);	

Таблиця 4.2. Можливі комбінації конверторів `Complex` і `TComplex`

class Complex	**class** TComplex
Complex(**const** TComplex& z);	TComplex(**const** Complex& z);
operator TComplex() **const**;	**operator** Complex() **const**;
operator TComplex() **const**; Complex(**const** TComplex& z);	
	operator Complex() **const**; TComplex(**const** Complex& z);

Не можна лише визначати одне й те саме перетворення двічі – водночас у вхідному та цільовому класах. Так, не дозволено водночас мати перетворення тригонометричного вигляду до алгебричного в класі `Complex` за допомогою конструктора, а в класі `TComplex` – за допомогою оператора перетворення так само, як не можна визначити перетворення алгебричного вигляду до тригонометричного в класі `TComplex` за допомогою конструктора, а в класі `Complex` – за допомогою оператора перетворення. Зрозуміло, що подібні дублювання створювали б для компілятора неоднозначність під час генерування коду для неявних перетворень.

Наведене правило відображено далі в табл. 4.3.

Таблиця 4.3. Недопустимі комбінації конверторів

class Complex	**class** TComplex
Complex(**const** TComplex& z);	**operator** Complex() **const**;
operator TComplex() **const**;	TComplex(**const** Complex& z);

Вправа 4.65

Реалізуйте всі можливі варіанти визначення перетворень комплексних чисел з алгебричного вигляду до тригонометричного й навпаки.

4.10. Особливості статичних членів класу

Розглядаючи концепцію класу, ми виходили з того, що кожен його екземпляр має власний набір атрибутів і спільний для всього класу набір методів. Із цього правила є виняток. Тут уже було розглянуто атрибути, спільні для всіх об'єктів класу, – це статичні атрибути, чи *атрибути класу*. Так само є статичні методи, чи *методи класу*. Їхня особливість полягає в тому, що для їх виклику не потрібен об'єкт, отже немає поточного об'єкта, до якого їх застосовують. Тому для методів класу немає указника на поточний об'єкт **this**.

За своєю суттю концепція статичних членів класу проста. Розглянемо демонстраційний приклад.

Лістинг 4.214

```
#include<string>
using namespace std;
// Клас демонстрації статичних членів
class DemoStatic {
private:
   static string _salutation; // Статичний член класу
   string _ownSalutation; // Власний атрибут об'єкта
public:
// Конструктор
   DemoStatic(const string& x=_salutation): _ownSalutation(x+x+x) {}
// Селектор атрибута класу
   static const string& getSalut() {
      return _salutation;
   }
// Модифікатор атрибута класу
   static void setSalut(const string& newSalut) {
      _salutation=newSalut;
   }
// Селектор власного атрибута об'єкта
   const string& getOwnSalut() {
      return _ownSalutation;
   }
// Модифікатор власного атрибута об'єкта
   void setOwnSalut(const string& newOwnSalut) {
      _ownSalutation=newOwnSalut;
   }
};
```

Статичний член класу, як і належить, ініціалізуємо у файлі реалізації:

Лістинг 4.215

```
#include "Static.h"
// Ініціалізація статичного члена класу
string DemoStatic::_salutation=string("Salutation!");
```

Тепер проведемо простий експеримент, який покаже відмінності у зверненні до методів класу й об'єкта.

Лістинг 4.216

```
#include <iostream>
using namespace std;
#include "Static.h"
int main() {
   cout<<"No object of DemoStatic exists";
// Початок програми: не існує жодного об'єкта демонстраційного класу,
// але статичний атрибут уже визначено. Звернемося до статичного методу
// й виведемо значення його статичного атрибута
   cout<<DemoStatic::getSalut()<<endl;
// Створимо два об'єкти демонстраційного класу
// з різними значеннями власних атрибутів
   DemoStatic* pDemo1=new DemoStatic;
   DemoStatic* pDemo2=new DemoStatic("Ciao! ");
// Виведемо значення власного атрибута кожного з об'єктів:
   cout<<pDemo1->getOwnSalut()<<endl;
// Результат – Salutation! Salutation! Salutation!
   cout<<pDemo2->getOwnSalut()<<endl; // Результат – Ciao! Ciao! Ciao!
   delete pDemo1; delete pDemo2; // Видалимо обидва об'єкти
// Знову немає жодного об'єкта.
// Атрибут класу й надалі доступний, змінимо його значення
   DemoStatic::setSalut("Hallo! ");
// Знову виведемо його значення: результат – Hallo!
   cout<<DemoStatic::getSalut()<<endl;
// Знову створимо об'єкт демонстраційного класу
   pDemo1=new DemoStatic;
// Виведемо значення його власного атрибута:
   cout<<pDemo1->getOwnSalut()<<endl; // Результат – Hallo! Hallo! Hallo!
   delete pDemo1; // Видалимо об'єкт
// Знову немає жодного об'єкта
// Атрибут класу й надалі доступний, ще раз змінимо його
   DemoStatic::setSalut("Hi! ");
   cout<<DemoStatic::getSalut()<<endl; // Результат – Hi!
   return 0;
}
```

Підіб'ємо підсумки. Статичні атрибути існують упродовж усього часу виконання програми незалежно від процесів створення та видалення об'єктів класу. Тому їх називають іще атрибутами класу. Атрибути класу, як і атрибути об'єктів, зазвичай закриті, а тому недоступні ззовні класу.

Для виклику статичного методу не потрібен об'єкт, але треба кваліфікувати область дії імен (його імені передує ім'я класу, відділене двома двокрапками «::»). Статичний метод має доступ до всіх статичних атрибутів, навіть закритих.

Виходячи зі сказаного, можемо констатувати, що статичні члени класу – це ще один статичний спосіб інкапсуляції даних і дій. Завдяки їм сам клас також являє собою своєрідний об'єкт метакласу, оскільки він має атрибути й методи. Правда, особливість класу як об'єкта полягає в його унікальності, тоді як звичайних об'єктів класу може бути скільки завгодно.

Вправа 4.66

Чи можна визначити клас, який:

- ніколи не зможе мати жодного об'єкта?
- завжди матиме не більше ніж один об'єкт?
- завжди матиме не менше ніж один об'єкт?

Тепер розглянемо класи, статичні атрибути яких – власні об'єкти. Для прикладу візьмемо клас `Date`, призначений для роботи з датами. Його будова дуже проста: у ньому два звичайні конструктори й один конструктор копіювання, а також конструктор без параметрів, який надає задану за умовчанням дату. Як побачимо далі, до цієї ситуації буде внесено певний динамізм, оскільки поточна дата щодня інша, а саме її хотілося б використовувати за умовчанням, для коректного визначення якого тепер потрібні додаткові зусилля:

Лістинг 4.217

```
// Клас дат
class Date {
public:
   struct BadDate; // Клас для обробки аварійних ситуацій
// Перелік місяців
   enum Month {
      jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
   };
// Конструктор із заданими за умовчанням параметрами (місяць – із переліку)
   Date(const unsigned int d=0, const Month m=Month(0),
        const unsigned int y=0);
// Конструктор з обов'язково заданими параметрами (місяць – ціле число)
   Date(const unsigned int d, const unsigned int m,
```

```
        const unsigned int y);
    Date(const Date&); // Копіювальний конструктор
    ~Date();           // Деструктор
// Селектори
    unsigned int day() const; Month month() const;
    unsigned int year() const;
// Текстові позначення місяців
    const string getMonthName() const;
// Статичні методи
// Задати дату за умовчанням за допомогою параметра
    static void setDefault
            (const unsigned int, const Month, const unsigned int);
// Узяти задану за умовчанням дату із системного годинника
    static void setDefault();
    static void showDefault(); // Показати задану за умовчанням дату
private:
    unsigned int _day, _month, _year; // День, місяць, рік
// Службові методи
    void fillDate                     // Модифікатор атрибутів
        (const unsigned int d, const Month m, const unsigned int y);
// Перевірка високосності року
    static bool leapYear(const unsigned int y);
// Статичні атрибути
    static char monthNames [12][4]; // Текстові імена місяців
// Статус дати за умовчанням
    enum TypeOfDefault {
        initial,     // до встановлення замовчування
        standard,    // стандартне замовчування за системним годинником
        user_defined // замовчування, визначене користувачем
    };
    static Date _defaultDate; // Задана за умовчанням поточна дата
};
ostream& operator<<(ostream&, const Date&); // Утиліта виведення дати
// Клас для обробки аварійних ситуацій
struct Date::BadDate {
    unsigned int _day, _month, _year;
    BadDate(const unsigned int d, const unsigned int m,
            const unsigned int y): _day(d), _month(m), _year(y) {}
};
// Утиліта виведення діагностичних повідомлень
ostream& operator<<(ostream&, const Date::BadDate&);
```

Клас починається з оголошення структури BadDate, екземпляри якої буде створено в разі виявлення спроби використати некоректну дату.

Визначення цієї структури – технічна деталь, тому його винесено за межі класу.

У лістингу 4.218 задано символічні позначення для місяців, які утворюють перелічуваний тип `Month`. Їх можна використовувати поряд зі звичайними позначеннями місяців за допомогою чисел, однак тоді виникають проблеми: наприклад, залежно від формату дати значення "9.2" можна розуміти як дев'яте лютого чи друге вересня.

Клас має два конструктори для двох можливих подань дат. Копіювальний конструктор і деструктор звичайні. Селектори `day()`, `month()` і `year()` дають змогу вибирати компоненти дати.

Нагадаємо, що визначення перелічуваного типу ще не дає символьних позначень для виведення. Їх доводиться кодувати окремо в масиві **`static char`** `monthNames [12][4]`, а функція `getMonthName()` надає символьне позначення місяця для виведення:

Лістинг 4.218

```
// Ініціалізація імен місяців
string Date::monthNames [12]= {
   "Jan", "Feb", "Mar", "Apr", "May", "Jun",
   "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
};
```

Розглянемо статичні методи пізніше, а поки що зробимо важливі зауваження стосовно конструкторів. Оскільки вони обидва різняться тільки формою подання дати, то забезпечують лише інтерфейс із точкою виклику. Усю ж роботу зосереджено в особливому методі **`void`** `fillDate` `(`**`const unsigned int`** `d,` **`const`** `Month m,` **`const unsigned int`** `y)`, доступ до якого ззовні закрито. Інший службовий метод – **`bool`** `leapYear(`**`int`** `y)` – перевіряє, чи не високосний рік:

Лістинг 4.219

```
// Інтерфейсні конструктори
Date::Date(const unsigned int d, const Month m, const unsigned int y) {
   fillDate(d, m, y);
}
Date::Date
   (const unsigned int d, const unsigned int m, const unsigned int y) {
   fillDate(d, Month(m), y);
}
```

Метод `fillDate` виконується по-різному під час першого та всіх наступних викликів. Першого разу він звертається до методу `setDefault()`, щоб узяти дату із системного годинника та запам'ятати її в статичному атрибуті `_defaultDate`. Усі наступні виклики формуватимуть дату з фак-

тичних параметрів, якщо їх задано, а ні – то заданим за умовчанням значенням дати буде _defaultDate. Далі перевіряється коректність заданих значень дня, місяця та року (якщо їх передано за допомогою параметрів). У разі виявлення помилки буде створено об’єкт BadDate(_day, _month, _year) та передано обробнику помилкових ситуацій:

Лістинг 4.220

```
// Модифікатор дати
void Date::fillDate
         (const unsigned int d, const Month m, const unsigned int y) {
// Ця умова виконується рівно один раз, оскільки перший об’єкт -
   if (_type==initial) // _defaultDate, створений до початку виконання
      setDefault();    // програми
#ifndef NDEBUG
      cout<<"Data constructor _defaultDate: "<<*this;
#endif
// Тепер аналізуємо параметри: ті, яких немає (вони нульові), беремо
// із заданої за умовчанням дати, яку вже гарантовано визначено
   _day=d? d :_defaultDate._day;
   _month=m? m :_defaultDate._month;
   _year=y? y: _defaultDate._year;
// Готуємося визначити кількість днів у місяці 28, 29, 30 чи 31
   unsigned int numberOfDays = 0;
// Аналізуємо місяці щодо кількості днів
   switch(_month) {
   // У лютому враховуємо, чи не високосний рік
      case feb: numberOfDays=28+leapYear(y); break;
      case apr: case jun: case sep: case nov: numberOfDays=30; break;
      case jan: case mar: case may:
      case jul: case aug: case oct: case dec: numberOfDays=31; break;
   // Місяць виявився некоректним
      default: throw BadDate(_day, _month, _year);
   }
// Перевірка коректності дня та року
   if((_day<1)||(numberOfDays<_day)||(_year<0))
      throw BadDate(_day, _month, _year);
#ifndef NDEBUG
   cout<<"Data constructor: "<<*this;
#endif
};
```

У визначенні класу було дві функції setDefault(): одна з параметрами, інша – без них. Тому й реалізацій буде дві. Перша дає можливість перевстановити задану за умовчанням дату: функція має параметри, а її виконання зводиться до тривіального виклику модифікатора:

```
// Задати дату за умовчанням за допомогою параметра        Лістинг 4.221
void Date::setDefault(const unsigned int d, const Month m,
                      const unsigned int y) {
   _type = user_defined; _defaultDate=Date(d, m, y);
}
```

У другому випадку – виклику без параметрів – функція сама бере із системного годинника дату, яку передає до `_defaultDate`:

```
#include <ctime>                                          Лістинг 4.222
// Задану за умовчанням дату взято із системного годинника
void Date::setDefault() {
// Обробка таймера й ініціалізація поточного дня в змінній timer
   time_t timer; time(&timer);
// Створення указника на поточну дату
// Тип таймера tm узято з бібліотеки ctime
   struct tm * today = gmtime(&timer);
// Вибір дня, місяця та року із указника today
   _defaultDate._day=today->tm_mday;
   _defaultDate._month=++(today->tm_mon);
   _defaultDate._year=today->tm_year+=1900;
}
```

Для компіляції зазначеної функції потрібно, як бачимо, підключити відповідний заголовний файл:

```
#include <ctime>
```

Метод `showDefault()` разом з операцією виведення дають змогу показувати дати в зручному для перегляду форматі:

```
// Утиліта виведення дати                                  Лістинг 4.223
ostream& operator<<(ostream& os, const Date& d) {
   os<<d.day()<<' '<<d.getMonthName()<<' '<<d.year()<<endl;
   return os;
}
```

Утиліту виведення безпосередньо використано в статичному методі виведення заданої за умовчанням дати:

```
// Показати задану за умовчанням дату                      Лістинг 4.224
void Date::showDefault() {cout<<_defaultDate;}
```

Оскільки об'єкт `_defaultDate` закритий і до нього не можна звернутися ззовні, його показує окремий метод. Сам об'єкт статичний, тому метод теж оголошено статичним, і він не потребує поточного об'єкта для свого виклику.

Нарешті, наведемо визначення самого об’єкта `_defaultDate`:

Лістинг 4.225

```
// Спершу встановлюємо невизначеність поточної дати
Date::TypeOfDefault Date::_type = Date::initial;
// Далі ініціалізуємо поточну дату звичайнісіньким конструктором
Date Date::_defaultDate; // Це скорочений вигляд, повний був би такий:
                         // Date Date::_defaultDate=Date();
```

Ініціалізація `_defaultDate` виконується за допомогою виклику конструктора без параметрів. Пересвідчимося, що конструктор буде викликано до початку виконання програми. Конструктор `Date` запрограмовано так, що його перший виклик відбувається під час визначення `_defaultDate`. Початкове значення керувальної змінної `_type` – `initial`, тому задане за умовчанням значення дати вибирають із системного годинника за допомогою функції `setDefault()`. Як видно з реалізації функції `fillDate()`, під час першого її виклику буде встановлено замовчувану дату, а значення `_type` буде змінено на `standard`.

Результати тестування цього класу за допомогою спеціальної програми продемонструємо в табл. 4.4.

Таблиця 4.4. Тестування класу Date

// Тестова програма **Лістинг 4.226**	Результати виведення

`int main() {`
`// Першим оператором програми робимо`
`// виведення, щоб побачити момент`
`// створення статичної дати за`
`// умовчанням із системного годинника`
`   cout<<"START:"<<endl;`
	`Data constructor`
	`_defaultDate: 1 Sep 2008`
	`START:`

`// Почнемо випробування`
`   try`
`   {`	

`// Покажемо задану за умовчанням дату` `      Date::showDefault();`	`1 Sep 2008`

`// Змінимо задану за умовчанням дату`
`      Date::setDefault(1,`
`            Date::Month(1),1970);`
`      Date::showDefault();`
	`1 Jan 1970`

`// Іще раз змінимо задану за умовчанням`
`// дату на системну`
`      Date::setDefault();`	

`// Визначимо три дати`
`    Date d, d1(14, 10, 1991), d2(d1);`
	`Data constructor: 1 Sep 2008`
	`Data constructor: 14 Oct 1991`
	`Data constructor: 14 Oct 1991`

`// Задамо нереальну дату` `    Date x(48, 14, 2008);`	

`  }`	`Destructor Date: 14 Oct 1991`
	`Destructor Date: 14 Oct 1991`
	`Destructor Date: 1 Sep 2008`

`// Обробка аварійної ситуації`
`  catch(Date::BadDate bd)`
`  {`
`    cerr<<"Bad date: "<<bd<<endl;`
`  }`
	`Bad date: 48:14:2008`

`  cout<<"END"<<endl;`
`  return 0;`
`}`
	`END`

Цікаво проаналізувати результат виконання цієї програми. Як бачимо, виклик конструктора справді відбувся до початку виконання основної функції, яка починається виведенням стартового повідомлення.

Зверніть увагу на те, що дані та повідомлення про помилки виводяться до різних вихідних потоків – відповідно `cout` і `cerr`.

Вправа 4.67

Доповніть клас `Date` операціями порівняння дат на рівність і нерівність; присвоєння; збільшення (зменшення) дати на задану кількість днів; присвоєння, суміщеного з операціями.

Як висновок, наголосимо на особливому статусі статичних методів і атрибутів. Зокрема, ми переконалися, що статичний атрибут може мати той самий тип (наприклад, `Date`), що й сам клас. Це абсолютно неможливо для нестатичного атрибута (поясніть, чому). Так само, як і до звичайних, доступ до статичних атрибутів закривають. Якщо ж потрібно надати доступ до них, це роблять за допомогою статичних методів.

Для виклику статичних методів не потрібні об'єкти, до яких їх застосовують, а тому вони можуть оперувати лише зі статичними атрибутами. Від вільних функцій їх відрізняють правила локалізації та доступу. Викликаючи статичний метод, потрібно задати область видимості, наприклад `Date::setDefault()`. Окрім того, статичні методи мають доступ до закритої частини класу, зокрема до статичних атрибутів.

Завдання до розділу 4

1. Точки площини задано структурою `Point`. Визначте і реалізуйте функції
   a. дзеркального відображення точки за довільною віссю координат;
   b. центрально-симетричного відображення точки відносно початку координат

   як члени класу і як його утиліти. Наведіть аргументи на користь того чи іншого рішення.
2. Визначте клас байтів `Byte`, що складаються з восьми бітів. Додатково до стандартних функцій класу реалізуйте облік об'єктів класу, модифікатор і селектор заданого біту, а також операцію циклічного зсуву байта вправо на один біт. Операція зсуву може бути утилітою або членом класу. Якій з них віддати перевагу?
3. Визначте клас машинних слів `Word`, що складається з чотирьох восьмирозрядних байтів, клас яких визначено в попередній задачі. Додатково до стандартних функцій класу реалізуйте модифікатор/селектор байту, двома способами операцію циклічного зсуву слова вліво на один байт. Чи можливе одночасне використання кожної з реалізацій?
4. У класі комплексних чисел `Complex` реалізуйте:
   a. операції додавання й додавання, суміщеного з присвоєнням, обґрунтувавши в кожному випадку вибір утиліти чи методу.
5. У класі комплексних чисел `Complex` реалізуйте:
   a. утиліти введення та виведення за допомогою довизначення операцій `>>` та `<<`;
   b. методи вводу та виводу за допомогою довизначення операцій `>>` та `<<`, помінявши місцями аргументи порівняно з утилітами (`z<<cin`, `z>>cout`).

   Зробіть висновки стосовно доцільності першого і другого рішення.
6. Визначте конструктор, деструктор, копіювальний конструктор і присвоєння для класу векторів довільного розміру.
7. В класі векторів тривимірного простору реалізуйте пару операцій: додавання та додавання, суміщеного з присвоєнням. Реалізацію обґрунтуйте.
8. Визначте для класу `Polynomial` многочленів довільного степеня з дійсними коефіцієнтами конструктор, деструктор, копіювальне присвоєння тамультиконструктор копіювання для періодичного повторення коефіцієнтів оригіналу. В класі `Polynomial` визначте операції додавання та присвоєння, суміщеного з додаванням.
9. Бітовим масивом називається масив, складений із нулів і одиниць Визначте конструктор, деструктор, копіювальний конструктор і присвоєння для класу бітових масивів.

10. Бітовим рядком називається рядок, складений із нулів і одиниць.
    a. Визначте клас BitString бітових рядків довільної довжини.
    b. У класі BitString реалізуйте конструктори, деструктор, копіювальне присвоєння та мультиконструктор копіювання.
11. Клас товарів Goods складається зіштрих-коду товару (**unsigned int**), вартості в копійках (**unsigned int**) і опису (char*).
    a. Визначте й реалізуйте конструктори, деструктор і присвоєння.
    b. Визначте оператор виведення товару.
    c. Клас касових апаратів CashRegister виконує операції над товарами: додавання полягає у додаванні вартостей, множення на натуральне число n дає вартість партії з n одиниць товару. Реалізуйте клас CashRegister.
    d. Клас покупок Purchase складається з послідовності пар (придбаний товар, кількість одиниць). Визначте операцію відображення покупки та функцію підрахунку її вартості.
12. Визначте клас комори Storehouse для зберігання товарів, відсортованих за штрих-кодами. В кожній програмі існує лише одна комора. Визначте функцію виведення інвентарної відомості комори.
    a. Реалізуйте метод find підрахунку за даним штрих-кодом кількості виробів у коморі.
    b. Реалізуйте метод include занесення виробу до комори.
    c. Реалізуйте метод exclude вилучення виробу з комори.
    d. Запрограмуйте створення і обробку аварійних ситуацій а) Lack, спричиненої наявністю критичної кількості виробів з даним штрих-кодом в коморі; б) ShelfLife – критичністю терміну зберігання.
    e. Визначте клас касових апаратів CashRegister для підрахунку вартості та створення чеку покупки за прейскурантом комори.
    f. Запропонуйте засоби підрахунку знижок залежно від:
        - терміну придатності продукту;
        - кількості одиниць виробів у одному чеку;
        - загальної вартості всього чеку.
13. Визначте клас SingleThing, який може мати тільки один об’єкт.
14. Побудуйте клас Nothing, який не може мати жодного об’єкта. Запропонуйте застосування цього класу.
15. Алгебричним одночленом називається вираз вигляду ax^b, де a,x,b – раціональні числа. Визначте клас Molinom алгебричних одночленів.

16. Алгебричною функцією називається сума алгебричних одночленів вигляду $A_n(x) = \sum_{i=1}^{n} a_i x^{b_i}$. Визначте клас алгебричних функцій `APolinom`. Реалізуйте додавання й виведення.
17. Визначте клас `Rpolinom` многочленів однієї дійсної змінної з дійсними коефіцієнтами вигляду $P_n(x) = \sum_{i=0}^{n} a_i x^i$. Реалізуйте додавання й виведення.
18. Визначте клас `Rational` раціональних функцій вигляду $R(x) = \dfrac{P_n(x)}{Q_m(x)}$, де $P_n(x)$ і $Q_m(x)$ – многочлени.
    a. Реалізуйте додавання й виведення.
    b. Запрограмуйте функцію табуляції значень раціональної функції на заданому відрізку $[a,b]$ із заданим кроком h.
19. Визначте клас `CPolinom` многочленів вигляду $C_n(z) = \sum_{i=0}^{n} a_i z^i$ від однієї комплексної змінної з комплексними коефіцієнтами.
    a. Реалізуйте додавання й виведення.
    b. Запрограмуйте функцію табуляції значень комплексного многочлена на заданому прямокутнику $[a_1+ib_1,\ a_2+ib_2]$ із заданим кроком h уздовж кожної вісі.
20. Визначте клас розріджених многочленів `Polynomial` (многочлен називається розрідженим, якщо більшість його коефіцієнтів рівні нулю). Реалізуйте функцію обчислення значення многочленна в заданій точці та операцію додавання многочленів.
21. В коло заданого радіуса r з центром в початку координат вписано правильний многокутник `Polygon`. Конструктор будує його вершини в напрямку, протилежному руху годинникової стрілки, починаючи з вершини $(r,0)$. Визначте копіювальне присвоєння многокутників і мультиконструктор копіювання з відповідним збільшенням або зменшенням числа вершин на задане число.
22. Мірою відрізка на прямій служить його довжина. Запропонуйте варіанти визначення для класу відрізків `Segment` операції додавання їхніх мір (+).
23. Визначте клас розбиття `Partition` заданого відрізка $[a,b]$. Передбачте копіювальне присвоєння і мультиконструктор копіювання з відповідним подвоєнням, потроєнням і т.д. числа вузлів розбиття.

24. Визначте клас криволінійних трапецій `CurveTrapezium`, заданих відрізком $[a,b]$ та функцією, інтегрованою на цьому відрізку. Реалізуйте обчислення площі криволінійної трапеції.
25. Трапеція називається прямою, якщо функція, що обмежує її, лінійна. Визначте клас `RightTrapezium` прямих трапецій на відрізку $[a,b]$. Реалізуйте обчислення площі прямої трапеції.
26. Операція додавання трапецій полягає в обчислені суми їхніх площ. Визначте та реалізуйте кілька варіантів операції додавання. Виберіть оптимальний варіант і обґрунтуйте вибір.
27. Реалізуйте класи трапецій, заданих своїми основами – `TrapeziumBase` і трапецій, заданих бічними сторонами, – `TrapeziumSide`. Визначте оператори взаємного перетворення типів трапецій.
28. Визначте клас Британських імперських мір маси `Imperial`: 1 тонна = 20 квінталів, 1 квінтал = 112 фунтів, 1 фунт = 16 унцій, 1 унція = 16 драхм. Визначте операцію перетворення імперських мір на метричні, якщо 1 драхма = 1,7718451953125 г.
29. Визначте клас руських мір маси `Russian`: 1 ласт = 72 пуди, 1 пуд = 40 гривень, 1 гривня = 96 золотників, 1 золотник = 96 долей. Визначте операцію перетворення руських мір на імперські й навпаки, якщо 1 доля = 44,43494 мг.
30. Визначте клас підмножин універсальної множини з n елементів.
    a. Реалізуйте конструктор, деструктор, копіювальний конструктор і присвоєння для класу підмножин.
    b. Визначте й реалізуйте операції доповнення, об’єднання та перетину для класу підмножин.
    c. Визначте оператори виведення та введення підмножин.
31. Визначте клас ромбів. Запропонуйте можливі варіанти конструкторів: за вершинами, сторонами, діагоналями. Реалізуйте функції розбиття ромба
    a. на два рівнобедрених трикутники однією з діагоналей;
    b. на чотири прямокутних трикутники.
32. Кожну n-місну булеву функцію можна задати двійковим натуральним числом m розрядності 2^n таким чином, що i-й розряд $m[i]$ кодує значення функції на наборі значень аргументів, заданих двійковим зображенням числа i. Визначте клас n-місних булевих функцій `Bool`. За якого обмеження на значення n можна запропонувати економну реалізацію цього класу?
    a. Реалізуйте оператори введення й виведення булевих функцій. Продумайте адекватний формат введення і виведення.
    b. Визначте операції диз’юнкції, кон’юнкції та заперечення булевих функцій.

5. Методи об'єктного програмування

Програмування мовою С++ надає розробнику можливість використовувати прийоми та конструкції, що знижують складність програм, підвищують їх надійність, не завдаючи шкоди ефективності. До їх числа належать управління правами і засобами доступу, управління зв'язками між об'єктами, програмування контейнерів, інтелектуальних указників, функціональних об'єктів тощо.

5.1. Порушення прав доступу

Закриваючи, відповідно до золотого правила доступу (підрозділ 4.4), атрибути всередині класів, перш за все дбають про їхню безпеку. Методи теж закривають у тому випадку, коли вони не призначені для вжитку поза межами класу. Але, виходячи з природи задачі, клас може дозволити цільове використання своєї закритої частини окремим клієнтам. Надання таких виняткових прав порушує правила, що регламентують права доступу. Порушення полягають у наданні виняткових прав щодо таких аспектів:

- доступу до атрибутів;
- доступу до закритих методів;
- створення та видалення об'єктів.

Клас може надати виняткові права доступу до своєї закритої частини іншому класу, окремій функції іншого класу чи вільній функції, оголошення яких усередині класу позначено ключовим словом **friend**. Тоді відповідна програмна конструкція здобуває статус дружньої: у класу з'являються *дружні класи* (friend class) або *дружні функції* (friend function). Надання виняткових прав доступу – серйозне порушенням інкапсуляції, оскільки воно розширює загальновизнаний набір конструкцій, яким надано доступ до об'єкта. Цей набір, як виявляється, може стати значно ширшим, аніж сам набір дружніх конструкцій.

Проаналізуємо приклад необачних «дружніх стосунків». Для цього візьмемо клас із нестандартною схемою узгодженості атрибутів: один атрибут містить адресу іншого. Простежимо, як дружні функції можуть зіпсувати стан об'єкта, порушити узгодженість і навіть зруйнувати об'єкт.

Лістинг 5.1

```
// Випереджальне оголошення класу, що уможливлює
// оголошення функцій, у яких цей клас є параметром
class Friendship;
// Функція, яку важко «запідозрити» в зміні стану об'єкта,
// оскільки його передано як сталий параметр
```

```
void spy(const Friendship&);
// Функція – посередник до руйнівника
void traitor(Friendship&);
// Руйнівник власних параметрів
void destroyer(int& id, int*& pid)
// «Довірливий» клас
class Friendship {
// Надання виняткових прав
   friend void spy(const Friendship&);
   friend void traitor(Friendship&);
private:
// Пов'язані між собою атрибути
   int _id; int* _pid;
public:
// Конструктор, що установлює зв'язок між атрибутами
   Friendship(int id): _id(id),_pid(&_id) {}
// Селектори
   int getId() {return _id;}
   int getPid() {return *_pid;}
// Перевірка узгодженості атрибутів
   bool consistent() {return _pid==&_id;}
};
```

Тепер розглянемо дружні, але «недружелюбні» функції. Перша з них наносить «дрібну» шкоду, змінюючи значення атрибута нібито сталого об'єкта. Згадайте проблему сталості в адресуванні другого рівня (див. підрозділ 3.2.6) і уявіть собі наслідки такої зміни значення, якщо йтиметься, наприклад, про атрибут стану балансу вашого рахунку в банку:

Лістинг 5.2

```
// Функція неочікувано псує атрибут
void spy(const Friendship& fr) {*fr._pid=-*fr._pid;}
```

Інша функція передає атрибути «довірливого» об'єкта далі. При цьому лише за виглядом виклику не можна навіть з'ясувати, що буде передано: значення чи адреси.

Лістинг 5.3

```
// Функція-посередник до руйнівника
void traitor(Friendship& fr) {destroyer(fr._id, fr._pid);}
```

Нарешті – повний руйнівник об'єкта. Атрибути стають невизначеними, хоча могло б бути й гірше (достатньо згадати про небезпеку завислих указників). Наведена далі функція, одержавши атрибути «довірливого» класу за допомогою параметрів, здатна руйнувати що завгодно:

```
// Руйнівник власних параметрів                          Лістинг 5.4
void destroyer(int& id, int*& pid) {pid=0; id=0;}
```

Звіт про катастрофу, що сталася, відобразимо в табл. 5.1.

Таблиця 5.1. Тестування дружніх функцій

// Тестова програма **Лістинг 5.5**	**Результати виведення**

`int main() {`
`// Створюємо об’єкт і перевіряємо`
`// узгодженість його атрибутів`
`   Friendship credulity(100);`
`   if(credulity.consistent()) cout<<"Well"<<endl;`
`   else cout<<"Bad"<<endl;`
	`Well`

`// Виводимо значення атрибута` `   cout<<credulity.getId()<<endl;`	`100`

`// Запускаємо «шпигуна» та знову`
`// перевіряємо узгодженість`
`   spy(credulity);`
`   if(credulity.consistent()) cout<<"Well"<<endl;`
`   else cout<<"Bad"<<endl;`
	`Well`

`// Виводимо значення атрибута` `   cout<<credulity.getId()<<endl;`	`-100`

`// Руйнуємо об’єкт і перевіряємо`
`// узгодженість його атрибутів`
`   traitor(credulity);`
`   if(credulity.consistent()) cout<<"Well"<<endl;`
`   else cout<<"Bad"<<endl;`
	`Bad`

`// Виводимо значення атрибута` `   cout<<credulity.getId()<<endl;`	`0`

`// Крах програми`
`   cout<<credulity.getPid()<<endl;`
`   return 0;`
`}`
	`An exception 'System.`
	`NullReferenceExcep-`
	`tion' has occurred in`
	`Friendship.exe`

Рекомендація. *Будьте вибагливі у виборі «друзів» для вашого класу.*

На додачу звернемо увагу на поширену типову помилку, пов’язану з невиправданим наданням будь-якій утиліті класу виняткових прав доступу. Так, скажімо, бажання вжити в утиліті додавання точок площини (див. клас із лістингу 4.7) прямі звертання до атрибутів –

```
// Невиправдане втручання до внутрішніх справ об’єкта          Лістинг 5.6
const Point operator+(const Point& u, const Point& v) {
   return Point(u._x+v._x, u._y+v._y);
}
```

замість використання селекторів

```
// Виважене використання селекторів.                           Лістинг 5.7
// Модифікатори залишаються недоступними
const Point operator+(const Point& u, const Point& v) {
   return Point(u.x()+v.x(), u.y()+v.y());
}
```

ще не може бути підставою для долучення цієї утиліти до дружніх функцій класу `Point` за допомогою оголошення

```
friend const Point& operator+(const Point&, const Point&);
```

Щойно було продемонстровано деякі наслідки нерозбірливої «дружби». Загалом же для надання виняткових повноважень потрібні набагато серйозніші підстави, ніж просто бажання зекономити на виклику кількох селекторів. До речі, застосування відкритого способу виклику функцій (inline) для всіх функцій, реалізації яких подано в заголовному файлі, може зробити виклик селектора таким самим ефективним, як і пряме звертання до атрибута.

Умотивованою причиною порушень прав доступу можуть бути лише змістовні особливості задачі, до вивчення яких ми переходимо.

Розглянемо приклад ієрархії п’яти офісних класів: працедавців `Employer`, штатних розписів працівників `Employee`, самих службовців `Person`, посад `Position` і бухгалтерів `Accountant`. Це спрощений варіант кадрової системи, яку схематично

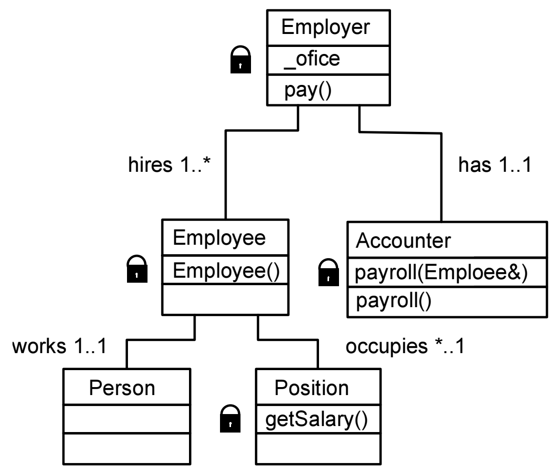


Рис. 5.1. Діаграма класів системи обліку кадрів

зображено на рис. 5.1. Асоціації між класами показано стрілками, які для зручності посилань пойменовано, хоча самі ці імена не ввійдуть до програми. Виразами вигляду `i..j`, де `i` та `j` – натуральні числа чи зірочка, позначено кратність відношень: `1..1` відображає взаємно однозначну відповідність; `1..*` відповідає відношенню «один до багатьох або, можливо, до жодного»; `*..1` – «багато чи, можливо, жоден до одного». На рис. 5.1 показано не всі атрибути й методи класів, а лише деякі суттєві з погляду порушення прав. Закриті члени класів позначено замком.

Розглянемо деякі особливості взаємозв'язків між класами.

- Тільки працедавець `Employer` має право створити робоче місце для службовця `Employee` та призначити особу `Person` на певну посаду `Position`.
- Тільки бухгалтерові `Accountant`, а точніше – його методу `payroll()` нарахування зарплати всьому особовому складу офісу, працедавець `Employer` надає доступ до свого штатного розпису.
- Тільки бухгалтерові `Accountant`, а точніше – його методу `void payroll (const Employee&)` нарахування зарплати окремому службовцеві, доступна інформація про заробітну плату для посади, яку займає службовець.

Розглядатимемо класи один за одним, пам'ятаючи, що наведені визначення далекі від повноти, а містять лише стільки деталей, скільки було потрібно для ілюстрації дружніх відношень між класами. Наводячи визначення класів і кодів їхніх функцій, не будемо більше згадувати про приєднання якихось заголовних файлів або випереджальні оголошення, залишаючи це читачеві для самостійної роботи. Корисно лише пам'ятати, що в усіх випадках, коли це можливо, перевагу доцільно віддавати випереджальним оголошенням. До речі, як це було продемонстровано, наприклад, в лістингах 4.204 і 4.207, вони бувають не тільки альтернативою команді препроцесора **`#include`**, але в певних випадках – єдино можливим способом установлення асоціацій між класами

Почнемо з двох найпростіших класів. Клас `Person` містить мінімальну інформацію про особу:

Лістинг 5.8

```
// Клас осіб потенційних працівників
class Person {
private:
// Прізвище, ім'я, по батькові
   string _name;
public:
```

```
// Конструктор за ПІБ
   Person(const char []);
// Деструктор
   ~Person();
// Селектор ПІБ
   const string& getName() const;
 };
```

а клас Position – про посаду:

```
// Клас номенклатури посад                                  Лістинг 5.9
class Position {
// Дружня функція нарахування зарплати одиниці штатного розпису,
// закріпленій за цією посадою
   friend void Accountant::payroll (const Employee&) const;
private:
// Назва посади
   string _name;
// Посадовий оклад
   unsigned int _salary;
// Селектор окладу: закрита функція – службова таємниця
   unsigned int getSalary() const;
public:
// Конструктор за назвою й окладом
   Position(char [],unsigned int);
// Деструктор
   ~Position();
// Селектор назви посади
   const string& getPositionName() const;
 };
```

Серед закритих атрибутів класу Position окрім назви посади є зарплата й селектор доступу до неї. Те, що метод являє собою селектор, а не модифікатор, видно з його сталості. Метод закритий, тому недоступний ззовні ні для кого, крім методу **void** payroll (**const** Employee&) класу Accountant, якому надано виняткові права доступу до закритої частини класу Position за допомогою ключового слова **friend**. Звісно, є певна надлишковість у наданні доступу до всієї закритої частини класу, а не лише окремого методу (у нашому випадку – **unsigned int** getSalary() **const**), але тонші методи інкапсуляції та приховування інформації ми розглядатимемо пізніше, у підрозділі 5.4. Метод payroll() одержить доступ до самого атрибута _salary та зможе змінити його. Однак раніше вже йшлося про те, що до питання добору «друзів» і надання їм прав дос-

тупу треба ставитися дуже обережно, а тому покладатимемося на коректну розробку бухгалтерського класу. Тут не так важливо регламентувати виняткові права дружнього класу, як заборонити всім іншим класам доступатися до закритої частини.

Наведемо реалізацію методів цих двох класів без коментарів. Налагоджувальні операції виведення тут не захищено командами препроцесора, що зроблено з метою економії місця та зі сподіваннями на певну досвідченість читача в цьому питанні, набуту в попередніх розділах.

Лістинг 5.10

```
// Конструктор особи потенційного службовця
Person::Person(const char name[]): _name(name) {
   cout<<"A person "<<_name<<" was created"<<endl;
}
// Деструктор особи потенційного службовця
Person::~Person() {
   cout<<"A person "<<_name<<" was deleted"<<endl;
}
// Селектор ПІБ
const string& Person::getName() const {return _name;}
// Конструктор посади за назвою й окладом
Position::Position(char name[], int salary):
      _name(name), _salary(salary) {
   cout<<"A Position "<<_name<<'('<<_salary<<")was created"<<endl;
}
// Деструктор посади
Position::~Position() {
   cout<<"A Position "<<_name<<" was deleted"<<endl;
}
// Селектор назви посади
const string& Position::getPositionName() const {return _name;}
// Селектор окладу: закрита функція – службова таємниця
int Position::getSalary() const {return _salary;}
```

Варто звернути увагу на проблеми з копіюванням. Розглянуті класи `Person` і `Position` – спрощені модельні варіанти. Зазвичай класи такого типу доволі об’ємні, вони містять багато полів великого розміру. Ненавмисне копіювання таких структур даних (наприклад, у разі присвоєння чи передавання параметрів значенням) може значно знижувати ефективність програми. Не слід забувати, що коли явно не визначено копіювальний конструктор чи оператор присвоєння, система програмування згенерує їх за замовчуванням.

Вправа 5.1

Перевірте коректність заданих за замовчуванням версій копіювального конструктора й оператора присвоєння для класів Person і Position. Результати обґрунтуйте.

Заборонити присвоєння легко: для цього достатньо помістити його в закритій частині класу, не наділивши реалізацією. Тоді компілятор не дозволить викликати присвоєння тим, хто не має доступу до закритої частини, розпізнавши таку спробу як синтаксичну помилку. Якщо до присвоєння звернеться оператор, що має доступ до закритої частини класу, то настане черга компонувальника не допустити виконання програми, яка містить виклик нереалізованої функції. Тому будь-яка спроба скористатися забороненим присвоєнням стане діагностованою. Заборонити викликати копіювальний конструктор важче. Помістивши його в закритій частині, можна взагалі закрити доступ до всіх конструкторів. Тоді цю заборону не вдасться втілити в життя на етапі компіляції програми. Доведеться залишити копіювальний конструктор не реалізованим, через що на етапі компонування програми може виникнути помилка в разі спроби викликати його, тільки тепер вже для всіх клієнтів – як допущених, так і ні. Слід заборонити й застосовувати звичайний конструктор у режимі перетворювача типів, надавши йому статус явного використання за допомогою специфікатора **explicit**.

Лістинг 5.11

```
// Клас осіб потенційних працівників
class Person {
private:
   string _name; // Прізвище, ім'я, по батькові
// Нереалізоване, а тому закрите присвоєння
   Person& operator=(const Person&);
public:
// Конструктор за ПІБ із забороною конвертувань
   explicit Person(const char []);
// Нереалізований копіювальний конструктор для відхилення спроби
// компілятора надати класу заданий за замовчуванням конструктор
   Person(const Person&);
   ~Person(); // Деструктор
   const string& getName() const; // Селектор ПІБ
};
```

Вправа 5.2

Реалізуйте захист від несанкціонованого копіювання в класі Position.

Розглянемо клас службовців:

Лістинг 5.12

```
// Службовець як одиниця штатного розпису
class Employee {
   friend class Employer; // Надання виняткових прав працедавцеві
private:
   const Position& _what; // Посада згідно з номенклатурою
   const Person& _who;    // Особа, що обіймає цю посаду
// Створення одиниці штатного розпису: закритий конструктор,
// доступний лише дружньому класу
   Employee(const Person& who, const Position& what);
   Employee(const Employee&); // Нереалізований копіювальний конструктор
   ~Employee(); // Деструктор
   Employee& operator=(const Employee&); // Нереалізоване присвоєння
public:
   const Person& who() const;    // Селектор особи
   const Position& what() const; // Селектор посади
};
```

Тут дуже просто закрити несанкціоноване копіювання, оскільки сам конструктор міститься в закритій частині. Виняткові права на створення робочих місць належать виключно працедавцеві `Employer`.

Важливо також, що атрибути цього класу – відсилки, а не самі об'єкти. Якщо замінити відсилки об'єктами –

```
// Версія без використання відсилок
   const Person _who; const Position _what;
```

то для створення об'єкта `Employee` знадобиться виконувати копіювання, але компонувальник заборонить його.

Вправа 5.3

Проаналізуйте, які наслідки зумовить відмова від відсилок у типі результату в селекторах

```
const Person Employee::who() const;
const Position Employee::what() const;
```

Відмова від копіювань, до якої ми тут удалися, має насправді дуже серйозне підґрунтя. Зауважимо, що крім втрати ефективності наявність кількох копій об'єкта зменшує їх узгодженість. Важко уявити, до яких результатів призведуть неузгоджені зміни атрибутів у різних копіях записів про одного й того самого працівника. Тому дуже важливо, що наведені далі селектори повертають відсилку на об'єкт, а самі атрибути також є відсилками на оригінали.

Коротко про реалізацію класу Employee:

Лістинг 5.13

```
// Створення одиниці штатного розпису
// за допомогою закритого конструктора
Employee::Employee(const Person& who, const Position& what):
          _who(who), _what(what) {
   cout<<"Employee "<<_who.getName()<<" takes a position "
       <<_what.getPositionName()<<endl;
}
// Деструктор реалізовано як звільнення працівника
Employee::~Employee() {
   cout<<"Employee "<<_who.getName()
       <<" fired from "<<_what.getPositionName()<<endl;
}
// Селектор особи
const Person& Employee::who() const {return _who;}
// Селектор посади
const Position& Employee::what() const {return _what;}
```

Перейдемо до класу бухгалтерів:

Лістинг 5.14

```
// Клас бухгалтерів
class Accountant {
   friend class Employee; // Надання виняткових прав працедавцеві
   friend class Position; // Надання виняткових прав
                          // класу номенклатури посад
public:
// Центральна функція нарахування платні в закладі
   void payroll() const;
   ~Accountant(); // Деструктор
private:
// Конструктор: наймання бухгалтера працедавцем
   Accountant(const Employer&, const Person&);
   Accountant(const Accountant&); // Нереалізоване копіювання
   Accountant& operator=(const Accountant&); // Нереалізоване копіювання
// Нарахування платні одному працівникові
   void payroll(const Employee&) const;
   const Employer& _myEmployer; // Працедавець бухгалтера
   const Person& _who;          // Особа бухгалтера
};
```

У конструкторі класу передбачено внесення інформації про працедавця, у якого працює бухгалтер, і про особу, яка займе цю посаду (знову ж у вигляді посилань).

```
// Конструктор: наймання бухгалтера працедавцем                Лістинг 5.15
Accountant::Accountant(const Employer& employer,
     const Person& accountant): _myEmployer(employer), _who(accountant) {
   cout<<"Accountant "<<_who.getName()<<" is hired"<<endl;
}
// Деструктор
Accountant::~Accountant() {
   cout<<"Accountant was deleted"<<endl;
}
```

У цьому класі є два методи `payroll()`. Відкритий метод без параметрів ініціює нарахування зарплати всім працівникам фірми:

```
// Центральна функція нарахування платні в закладі             Лістинг 5.16
void Accountant::payroll() const {
   cout<<"---Start payroll"<<endl;
// Цикл за штатним розписом
   for(size_t i=0; i<_myEmployer.volume(); i++)
      if(!_myEmployer._office[i]._free)
// Якщо посада зайнята, нарахувати платню тому, хто її обіймає
         payroll(*_myEmployer._office[i]._pemployee);
   cout<<"---Stop payroll"<<endl;
}
```

Він викликає закритий службовий метод **`void`** `payroll (`**`const`** `Employee&)` **`const`**, який виконує розрахунки для кожного працівника:

```
// Нарахування платні одному працівникові                      Лістинг 5.17
void Accountant::payroll(const Employee& employee) const {
   cout<<"Pay "<<employee.who().getName()<<" amount "
       <<employee.what().getSalary()<<endl;
}
```

Зверніть увагу на транзитивність викликів. Виклик `employee.who()` дає змогу з'ясувати особу службовця, а застосований до результату цього виклику метод `getName()` повертає його ім'я. Так само ланцюжок `employee.what().getSalary()` зумовлює виклик селектора окладу посади, яку займає службовець. Функція `getSalary()` закрита, проте доступ до неї забезпечено винятковими правами функції `payroll()`.

Нарешті, клас працедавця:

```
// Клас працедавців                                            Лістинг 5.18
class Employer {
// Виняткові права функції нарахування платні
```

```
   friend void Accountant::payroll() const;
private:
// Структура закладу
   struct Staff {
      Employee* _pemployee; bool _free;
   };
   size_t _volume; // Чисельність працівників закладу
   Staff* _office; // Указник на заклад, що належить працедавцеві
   Accountant* _accountant; // Указник на бухгалтера
   const Person& _me; // Особа працедавця
// Нереалізовані копіювання
   Employer& operator=(const Employer&); Employer(const Employer&);
public:
// Конструктор працедавця
   Employer(const Person& employer, const size_t volume,
            const Person& accountant);
   ~Employer(); // Деструктор працедавця
   size_t volume() const {return _volume;}  // Селектор
   void hire(const Person& , const Position&); // Наймання працівника
   void fire(const size_t); // Звільнення працівника
   void pay() const; // Нарахування платні
};
```

Зверніть увагу на те, що атрибути `_office` і `_accountant` задано указниками, а не відсилками, як в інших класах, оскільки відповідні об'єкти створює конструктор класу `Employer`, а тому їх не існує до його виклику. Деструктор же видаляє ці атрибути.

Лістинг 5.19

```
// Конструктор працедавця
Employer::Employer (const Person& boss, const size_t volume,
 const Person& accountant): _me(boss), _volume(volume),
 _office(new Staff[_volume]), // Створення закладу на _volume посад
 _accountant(new Accountant(*this, accountant)) { // Наймання бухгалтера
   cout<<"Employer "<<_me.getName()<<" with "<<_volume
       <<" positions was created"<<endl;
// Оголошення всіх створених посад вакантними
   for(size_t i=0; i<_volume; i++)
      _office[i]._free=true;

}
// Деструктор працедавця
Employer::~Employer() {
   cout<<"Employer "<<_me.getName()<<" with "<<_volume
       <<" positions was deleted"<<endl;
```

```
   delete [] _office; // Ліквідація закладу
   delete _accountant; // Звільнення бухгалтера
}
```

У цьому прикладі показано різні способи використання указників. Обидва атрибути `_office` і `_accountant` мають один і той самий тип, але перший використовують як масив, а другий – як звичайну динамічну змінну.

Тут наведено метод створення робочого місця, у якому викликано закритий для сторонніх конструктор класу `Employee`:

Лістинг 5.20

```
// Наймання працівника
void Employer::hire(const Person& person, const Position& position) {
   Employee* pemployee=new Employee(person, position);
   size_t i=0;
   while((i<_volume)&&(!_office[i]._free)) i++;
   _office[i]._free=false; _office[i]._pemployee=pemployee;
}
```

Вправа 5.4

Доповніть функцію `Employer::hire()` реакцією на те, що немає вакантних посад.

Вправа 5.5

Реалізуйте метод звільнення робочого місця `void Employer::fire(const size_t i)`.

Наступний метод дуже важливий, хоча простий. Він ілюструє прийом, відомий під назвою *делегування* (delegating).

Лістинг 5.21

```
// Нарахування платні
void Employer::pay() const {_accountant->payroll();}
```

Суть цього прийому полягає в тому, що працедавець передає певну функцію (тут – нарахування зарплати) на виконання іншому класу, у цьому разі – бухгалтерові. Ми повернемося до делегування пізніше.

Розглянуті вище задачі демонструють типові приклади обґрунтованого порушення прав доступу. Як бачимо, питання полягає не в економії на обчисленнях завдяки прямому доступу до атрибутів, якого в разі порушення прав доступу варто було б уникати. Виняткові права доступу надаються зі змістовних міркувань щодо предметної області задачі, як-от виключні права на створення об’єкта (працівник і працедавець), дотримання конфіденційності (бухгалтер і посадовий оклад) тощо, а не з бажання спростити реалізацію.

5.2. Типізація членів класів і структур

Ми вже порівнювали функціонально рівносильні операції, реалізовані утилітами та методами класу. Виходячи з концепції видимості, важливо розуміти, чим різняться бачення функції – члена класу зсередини та ззовні класу. Для цього спочатку спробуємо розібратися в тому, як співвідноситься типізація методів і рівнозначних утиліт, наприклад чим відрізняється тип операції додавання, визначеної за допомогою методу класу, від типу цієї самої операції, визначеної як утиліта.

Візьмемо, наприклад, клас комплексних чисел (лістинг 4.7) і розглянемо одну з його операцій, наприклад операцію додавання, реалізовану за допомогою утиліти. Як відомо, цю операцію можна записати кількома схожими способами, два з яких повністю коректні, але різняться ефективністю:

Лістинг 5.22

```
const Complex operator+(const Complex,const Complex);
const Complex operator+(const Complex&,const Complex&);
```

Результат виконання цієї операції – знову об'єкт класу. Досі сума найчастіше була правою частиною присвоєння чи складовою іншого виразу, наприклад:

Лістинг 5.23

```
z=Complex(1,1)+Complex(3,2);
Complex(1,1)+Complex(3,2)+Complex(-1,-1);
```

Проте її можна використовувати і як об'єкт, до якого застосовано методи класу комплексних чисел, як, скажімо, у виразі

```
(Complex(1,1)+Complex(3, 2))._re();
```

Зауваження. *Пригадайте, як завдяки застосуванню умови сталості результату було заборонено використовувати суму безглуздим способом у присвоєннях на кшталт*

Лістинг 5.24

```
Complex(1,1)+Complex(3,2)=Complex(-1,-1);
Complex(1,1)+Complex(3,2)=0;
```

Однак можна коректно застосовувати до суми інші методи, наприклад селектори (але не модифікатори), дозволені для сталих об'єктів.

Сказане стосується не лише операції додавання, але й інших. Доречно також згадати, як можна типізувати всі такі операції – застосувавши функціональну полісемію. Для цього достатньо визначити такий тип конструкції, яка набувала б значення лише в разі використання операцій відповідного типу (у цьому випадку застосовують указники). Скажімо, у

контексті визначення операцій із двома сталими комплексними параметрами та сталим комплексним результатом, наприклад

```
const Complex operator+(const Complex, const Complex);
```

тип указника `PtrToFunc` на ці операції можна задати так:

```
typedef const Complex(*PtrToFunc)(const Complex, const Complex);
```

Якщо ж ідеться про операції, параметри яких – сталі відсилки, наприклад про

```
const Complex operator+(const Complex&, const Complex&);
```

то тип указника `PtrToFuncRef` на такі операції визначають як

```
typedef const Complex(*PtrToFuncRef)(const Complex&, const Complex&);
```

Указниками на операції можна користуватись як і будь-якими іншими указниками на функції. Доведеться лише записувати ім'я операції не скорочено, наприклад знаком **+**, а повністю:

Лістинг 5.25

```
PtrToFuncRef cp=operator+; cout<<cp(z1,z2)<<endl;
```

Якщо ж перенести операцію додавання всередину класу, то вона, як уже з'ясовано, утрачає один явний аргумент –

```
const Complex Complex::operator+(const Complex) const;
const Complex Complex::operator+(const Complex&) const;
```

Його роль неявно перебирає на себе указник на поточний об'єкт **`this`**.

Очевидно, що указник на функцію вигляду

```
typedef const Complex(*PF1)(const Complex);
typedef const Complex(*PF1R)(const Complex&);
```

у цьому разі не підійде, оскільки його значеннями були б одномісні утиліти, наприклад

Лістинг 5.26

```
// Одномісна утиліта спряження
const Complex conj(const Complex& u) {
   return Complex(u._re(),-u._im());
}
```

тип яких, звісно, відрізняється від типу операції додавання як члена класу комплексних чисел.

Щоб за межами класу визначити тип указника на метод цього класу, треба додати в текст визначення посилання на простір імен класу (у попередньому прикладі – `Complex::`).

Визначимо тип PtrMRef указника на метод у класі комплексних чисел. Метод матиме один параметр – сталу комплексну відсилку – і сталий комплексний результат:

```
typedef const Complex (Complex::*PtrMRef)(const Complex&) const;
```

Далі наведено приклад використання указника на метод. Зауважте, що потрібно знову ж записувати операції не скорочено, а повністю:

Лістинг 5.27

```
// Установлюємо указник методу в класі Complex
// на операцію множення
PtrMRef cp=Complex::operator*;
// В об’єкті z1 беремо метод, на який показує указник cp,
// і застосовуємо його до об’єкта z2
cout<<(z1.*cp)(z2)<<endl;
```

Тепер указник cp в будь-якому об’єкті типу Complex може показувати на довільний метод заданої у визначенні сигнатури, тобто сталий метод з одним параметром типу сталої відсилки та сталим результатом:

Лістинг 5.28

```
PtrMRef p=Complex::operator*;
cout<<(z1.*p)(z2)<<endl; // Множення
p=Complex::operator/;
cout<<(z1.*p)(z2)<<endl; // Ділення
```

Вправа 5.6

Запрограмуйте калькулятор для комплексних чисел. За введеними двома числами й операцією має бути виведено результат застосування цієї операції до заданих аргументів.

Тепер покажемо, як розрізняти функції за їхніми типами в разі одночасного використання однойменних сталих і несталих методів. Для прикладу візьмемо простий текстовий екран:

Лістинг 5.29

```
// Текстовий екран розміру не більшого,
// ніж maxHeigh x maxWidth
class Screen {
public:
// Конструктор: Екран буде заповнено рядком pc, якщо його задано,
// а ні - то наповнювачем _filler
   Screen(const size_t, const size_t , char* pc=0);
   ~Screen();// Деструктор
// Однойменні сталі та несталі методи
// Помістити курсор на початок екрана
```

```
   const Screen& home() const; Screen& home();
// Перемістити курсор на один символ праворуч
   const Screen& move() const; Screen& move();
// Перемістити курсор на один символ ліворуч
   const Screen& back() const; Screen& back();
// Показати вміст екрана
   const Screen& show() const; Screen& show();
// Перенести курсор на задану позицію
   const Screen& move(const size_t, const size_t) const;
   Screen& move(const size_t, const size_t);
// Очистити екран
   const Screen& clear()const; Screen& clear();
// Показати символ, перед яким міститься курсор
   const Screen& showCurrent() const; Screen& showCurrent();
// Селектор
   char get() const; char get();
// Модифікатор
   const Screen& set(char) const; Screen& set(char);
private:
   static const size_t  maxHeight; // Максимальна висота екрана
   static const size_t  maxWidth; // Максимальна ширина екрана
   static const char _filler; // Наповнювач
   size_t  _height; // Фактична висота екрана
   size_t  _width; // Фактична ширина екрана
   char* _wContent; // Наповнення екрана
   mutable size_t  _cursor; // Де міститься курсор
// Нереалізовані, а тому закриті копіювання:
// копіювання екранів не передбачено
   Screen(const Screen&); Screen& operator=(const Screen&);
};
```

Зверніть увагу на специфікатор **mutable** (змінюваний) при атрибуті курсора `_current`. При переміщенні курсора ноповнення екрану, але змінюється значення природно не змінюється атрибут `_current`. Для того щоб атрибув мав право змінюватися навіть у сталому об'єкті передбачено спеціальний тип змінюваних другорядних атрибутів. Спочатку задамо максимальні висоту й ширину, а також наповнювач (у деяких версіях системи програмування це можна зробити безпосередньо у визначенні класу):

Лістинг 5.30

```
const size_t  Screen::maxHeight=24;
const size_t  Screen::maxWidth=80; const char Screen::_filler='.';
```

Конструктор формує екран потрібного розміру та заповнює його заданим рядком, а залишок – наповнювачем, якщо довжина рядка менша за ємність екрана:

Лістинг 5.31

```
// Конструктор: Екран буде заповнено рядком pc,
// якщо його задано, а ні - то наповнювачем _filler
Screen::Screen(const size_t  m, const size_t  n, char* s):
// Ініціалізація довжини та ширини екрана з одночасною перевіркою
// виходу за його максимальні межі - у такому разі буде взято їх
   _width(m>maxWidth? maxWidth :m), _height(n>maxHeight? maxHeight :n),
   _wContent(new char[_height*_width+1]), // Образ екрана в пам’яті
   _cursor(0) {  // Курсор установлюється перед першим символом
// Указник може бути невизначеним, тоді вважаємо рядок порожнім, а ні -
// то беремо його довжину: if (s==0) lens=0; else lens = strlen(s);
   size_t lens= s==0?0:strlen(s);
// Якщо довжина рядка наповнення надто велика, урізаємо його
   size_t len=lens>_height*_width? _height*_width :lens;
// Його заповнення починаємо з кінця символом закінчення рядка
   _wContent[_height*_width]='\0'; size_t k=0;
   if(s!=0)
// Записуємо на початку екрана рядок, переданий як параметр
      for(; k<len; k++)
         *(_wContent+k)=*s++;
// Заповнюємо екран до кінця наповнювачем
   for(; k<_height*_width; k++)
      *(_wContent+k)=_filler;
}
```

Функція деструктора звичайна – вивільнення динамічної пам’яті:

Лістинг 5.32

```
// Деструктор
Screen::~Screen() {delete [] _wContent;}
```

Усі методи влаштовано так, щоб застосовувати їх транзитивно (якщо є така можливість). Тому кожен такий метод повертає як результат відсилку на екран, сталу або змінну, унаслідок чого довелося дублювати сталі методи аналогічними несталими, щоб їх можна було застосовувати в позиціях, де формально потрібні лівобічні значення. Спочатку розглянемо ці парні методи, а потім продемонструємо їх застосування.

Метод `home()` повертає курсор у початок екрана. Його сталий варіант можливий, бо курсор має особливий статус змінюваного атрибута:

Лістинг 5.33

```
// Розміщення курсору на початку екрана
const Screen& Screen::home() const {_cursor=0; return *this;}
Screen& Screen::home() {_cursor=0; return *this;}
```

Оскільки, як бачимо, реалізації методів у сталому та несталому варіантах однакові, то наведемо далі лише сталі їх версії.

Метод `move()` забезпечує зсув курсору на одну позицію праворуч із переходом на новий рядок у разі потреби. Після досягнення кінця екрана курсор переміщується в його початок:

Лістинг 5.34

```
// Переміщення курсору на один символ праворуч
const Screen& Screen::move() const {
   if((++_cursor)>=_width*_height) _cursor=0;
   return *this;
}
```

Метод `back()` переміщує курсор на одну позицію ліворуч:

Лістинг 5.35

```
// Переміщення курсору на один символ ліворуч
const Screen& Screen::back() const {
   if(--_cursor<0) _cursor=0;
   return *this;
}
```

Замість стандартної операції виведення в класі реалізовано метод `show()`, службовий за своїм призначенням. За його допомогою можна переглянути вміст екрана на будь-якому етапі роботи з ним. Функція – член класу в цьому разу має переважне право доступу до його закритої частини, зокрема до курсору. Перед початком виконання функції поточне значення курсору буде збережено та відновлено після виходу з неї:

Лістинг 5.36

```
// Відображення вмісту екрана
const Screen& Screen::show() const {
   size_t  cursor=_cursor; // Збереження значення курсору
   cout<<"cursor="<<_cursor<<endl; home();
// Цикл за рядками екрана
   for(size_t  i=0; i<_height; i++) {
// Формування одного рядка
      for(size_t  j=0; j<_width; j++) {
         cout<<get(); move();
      }
      cout<<endl;
   }
   _cursor=_cursor; // Відновлення значення курсору
   return *this;
}
```

Вправа 5.7

Узагальніть функцію show() для довільного вихідного потоку та реалізуйте на її основі утиліту виведення екрана.

Метод move(**const size_t, const size_t**) переміщує курсор у задану позицію:

Лістинг 5.37

```
// Перенесення курсор у задану позицію
const Screen& Screen::move(const size_t  i, const size_t  j) const {
// Якщо позиція некоректна, переносимо курсор у початок екрана
   if((i>=_height)||(j>=_width)) _cursor=0;
   else _cursor=_width*i+j;
   return *this;
}
```

Метод clear() очищає поточний вміст екрана та заповнює його наповнювачем:

Лістинг 5.38

```
// Очищення екрана
const Screen& Screen::clear() const {
   for(size_t  i=0; i<_height*_width; i++)
      *(_wContent+i)=_filler;
   _cursor=0;
   return *this;
}
```

Функцію селектора виконує метод get():

Лістинг 5.39

```
// Відображення символу, перед яким міститься курсор
char Screen::get() const {
   return *(_wContent+_cursor);
}
```

Модифікатор теж розрахований на транзитивне застосування:

Лістинг 5.40

```
const Screen& Screen::set(char a) const {
   *(_wContent+_cursor)=a;
   return *this;
}
```

Розглянемо тепер простий приклад використання екрана, зобразивши на екрані Андріївський хрест.

Лістинг 5.41

```
int main() {
   const size_t size=10; Screen demo(size, size);
```

```
   cout<<"Empty screen created"<<endl<<endl; demo.show();
   for(size_t  i=0; i<=size; i++)
      demo.set('*').move(i, i);
   cout<<"The screen filled"<<endl<<endl;
   for(i=0; i<=size; i++)
      demo.move(size-i-1, i).set('*');
   demo.show();
   return 0;
}
```

Результат виконання цієї програми зображено на рис. 5.2.

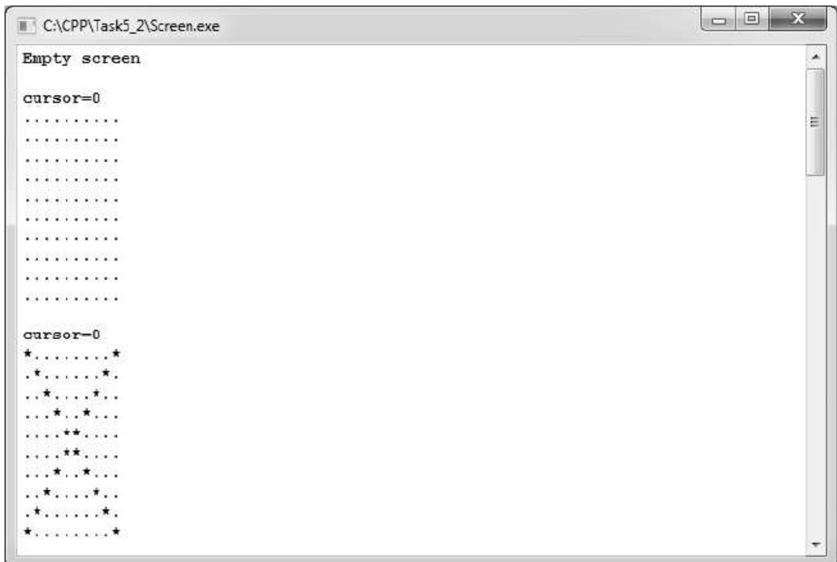


Рис. 5.2. Андріївський хрест

Вправа 5.8

Проаналізуйте й виконайте наведений далі приклад, який демонструє застосування транзитивності до сталих і несталих методів. У ньому **кожен** наступний метод щоразу застосовано до результату, створеного попереднім методом.

Лістинг 5.42

```
cout<<"1. Transitivity"<<endl;
Screen v(4, 4, "1111222233334444");
v.show().set('*').move(1, 1).set('*').move(2, 2).set('*').move(3, 3).
   set('*').show();
```

```
cout<<"2. Const transitivity"<<endl;
const Screen cv(5, 5, "aaaaabbbbbcccccdddddeeeee");
cv.show().move(1,1).showCurrent().move(2,2).showCurrent().move(3,3).
   showCurrent().show();
```

Тепер можна визначити два типи указників на методи для класу Screen: перші будуть оперувати зі сталими методами, другі – зі всіма іншими:

```
typedef const Screen& (Screen::*const ConstAction)() const;
typedef Screen&(Screen::*const NonConstAction)();
```

Розглянемо функції, які одержать указники наведених типів як параметри:

Лістинг 5.43

```
// Застосувати до екрана s сталий метод act n разів
void doActionConst(const Screen& s, const ConstAction act,
                   const size_t n) {
   for(size_t  i=0; i<n; i++) (s.*act)();
}
// Застосувати до екрана s метод act n разів
void doActionNonConst(const Screen& s, NonConstAction act,
                      const size_t n) {
   for(size_t  i=0; i<n; i++) (s.*act)();
}
```

Вправа 5.9

Проаналізуйте й виконайте наведений далі фрагмент програми, що демонструє приклад застосування цих функцій для реалізації простого меню.

Лістинг 5.44

```
const NonConstAction menuNonConst[]= {
   &Screen::home, &Screen::move, &Screen::back, &Screen::clear
};
const ConstAction menuConst[]= {
   &Screen::home, &Screen::move, &Screen::back, &Screen::show
};
size_t  k, n; char ch; cout<<"3. Menu Non Const"<<endl; v.show();
do {
   cout<<"your action?(0-home:1-move:2-back:3-clear)"; cin>>k;
   cout<<"multiplicity? "; cin>>n;
   doActionNonConst(v, menuNonConst[k], n);
   cout<<"Repeat(y/n)?"; cin>>ch;
} while(ch=='y');
v.show(); cout<<"4. Menu Const"<<endl; cv.show();
```

```
do {
   cout<<"your action?(0-home:1-move:2-back:3-show)"; cin>>k;
   cout<<"multiplicity? "; cin>>n;
   doActionConst(cv, menuConst[k], n);
   cout<<"Repeat(y/n)?"; cin>>ch;
} while(ch=='y');
cv.show();
```

Вправа 5.10

Доповніть меню обробки екрана з попередньої вправи такими командами:

- перемістити курсор у задану точку;
- модифікувати символ під курсором.

Вправа 5.11

Задайте сигнатуру та реалізацію утиліти `exe()`, здатної виконувати будь-яку операцію простого арифметичного калькулятора, фрагмент визначення якого наведено далі, якщо арифметичні операції виконуються над вмістом регістра акумулятора, як це показано на прикладі операції додавання. Інші операції визначте самостійно.

Лістинг 5.45

```
// Арифметичний калькулятор
struct Calc {
public:
   double _a; // Регістр акумулятора
   double _x; // Регістр x
// a = x + a
   double add() {return _a+=_x;}
// Інші операції
...............
};
```

Важлива особливість указників на методи в класах полягає в тому, що можна посилатися на метод, не фіксуючи об’єкт, до якого цей метод буде застосовано. Це дає змогу відокремити вибір конкретного методу від вибору об’єкта. Цю можливість було продемонстровано, наприклад, в лістингах 5.27, 5.28. Тепер додамо можливість параметризувати не лише метод, але й атрибут. З огляду на права доступу краще застосовувати типізацію до членів структур. Для прикладу візьмемо структуру матеріальних точок тривимірного простору:

```
struct MPoint {                                          Лістинг 5.46
   double _x, _y, _z; // Координати точки
   double _m; // Маса точки
};
```

Розглянемо задачу обчислення центра мас системи матеріальних точок

```
MPoint p[n];
```

Його координати обчислюють за формулами

$$x_c = \sum \frac{m_k x_k}{M}; y_c = \sum \frac{m_k y_k}{M}; z_c = \sum \frac{m_k z_k}{M},$$

де M – сума мас точок.

Для обчислення кожної з трьох координат доведеться побудувати окрему функцію; функції відрізнятимуться лише селектором відповідної координати:

```
// Обчислення координати x центра мас                   Лістинг 5.47
double centerx(const MPoint p[], const size_t  number) {
   double moment=0; double sum=0;
   for (size_t  i=0; i<number; ++i) {
      moment+=p[i]._m*p[i]._x; sum+=p[i]._m;
   }
   return moment/sum;
}
// Обчислення координати y центра мас
double centery(const MPoint p[], const size_t  number) {
   double moment=0; double sum=0;
   for (size_t  i=0; i<number; ++i) {
      moment+=p[i]._m*p[i]._y; sum+=p[i]._m;
   }
   return moment/sum;
}
// Обчислення координати z центра мас
double centerz(const MPoint p[], const size_t  number) {
   double moment=0; double sum=0;
   for (size_t  i=0; i<number; ++i) {
      moment+=p[i]._m*p[i]._z; sum+=p[i]._m;
   }
   return moment/sum;
}
```

Це рішення, попри його коректність, не видається досконалим, оскільки невиправдано продубльовано значну частину коду. Натяком на пошук у

правильному напрямі могла б бути порада змінити визначення матеріальної точки. Як уже було сказано в підрозділі 2.9.1, масиви відрізняються від структур способом доступу до своїх членів. Помістивши координати матеріальної точки до масиву, одержимо структуру такого вигляду:

Лістинг 5.48

```
// Матеріальна точка з пронумерованими координатами
struct MPoint3 {
   double _coord[3]; double _m;
};
```

Тепер номер координати можна передавати як параметр:

Лістинг 5.49

```
// Обчислення координати центра мас:
// використання масиву координат
double center3(const MPoint3 p[], const size_t  number,
               const size_t  index) {
   double moment=0; double sum=0;
   for (size_t  i=0; i<number; ++i) {
      moment+=p[i]._m*p[i]._coord[index]; sum+=p[i]._m;
   }
   return moment/sum;
}
```

Цей метод також незадовільний, принаймні для ситуацій, коли не можна змінити визначення структури. Проте останнє рішення підказує спосіб розв'язання задачі, який стає зрозумілішим, якщо замінити звертання до координати з масиву `_coord[index]` на указник `*(_coord+index)`:

Лістинг 5.50

```
// Обчислення координати центра мас:
// застосування адресної арифметики
double center3(const MPoint3 p[], const size_t  number, const size_t
index) {
   double moment=0; double sum=0;
   for (size_t  i=0; i<number; ++i) {
      moment+=p[i]._m*p[i].*(_coord+index); sum+=p[i]._m;
   }
   return moment/sum;
}
```

Тепер залишається лише замінити комбінацію указника й індексу `*(_coord+index)` на один указник, спрямований на потрібну координату. Для цього використовують тип **`double`** `MPoint::*pcoord` для позначення указника на атрибут дійсного типу в структурі `MPoint`:

```
// Координата центра мас:                                    Лістинг 5.51
// застосування указників на координати
double center(const MPoint p[], const size_t  number, double
MPoint::*pcoord) {
   double moment=0; double sum=0;
   for (size_t  i=0; i<number; ++i) {
      moment+=p[i]._m*p[i].*pcoord; sum+=p[i]._m;
   }
   return moment/sum;
}
```

Ось проста функція, яка демонструє використання указників на атрибути. Їх значеннями стають тепер адреси потрібних атрибутів – відповідно `&MPoint::_x`, `&MPoint::_y` або `&MPoint::_z`.

```
void showCenter(const MPoint& s, const size_t  n) {          Лістинг 5.52
// Обчислення центра мас: обчислюємо та виводимо координати x, y та z
   cout<<center(s, n, &MPoint::_x)<<','<<center(s, n, &MPoint::_y)<<','
       <<center(s, n, &MPoint::_z)<<endl;
}
```

Вправа 5.12

Означте для системи матеріальних точок функцію змінення знака довільної координати `x`, `y` або `z` на протилежний:

- водночас для всіх точок;
- для точки із заданим номером.

Як було зазначено в (підрозділі 3.2.4), статус статичних членів класу відрізняється від статусу звичайних членів. Так, статичні атрибути є спільними для всього класу, тоді як звичайні атрибути – складові частини своїх власних об'єктів. Так само статичні методи відрізняються від звичайних тим, що не мають неявного параметра **`this`**, який у нестатичному методі посилається на поточний об'єкт, до якого його застосовано.

Розглянемо особливості типізації статичних членів класів на прикладі спрощеної моделі банку, заданої класом `Bank`.

```
// Дуже спрощений банк                                       Лістинг 5.53
class Bank {
// Виняткові права має клас поточних рахунків
   friend class CurrentAccount;
private:
```

```
   const unsigned int _bankId; // Ідентифікаційний номер філії банку
   static double _normalInterestRate; // Запроваджена процентна ставка
   static double _advancedInterestRate; // Підвищена процентна ставка
   static double _goldInterestRate;     // «Золота» процентна ставка
public:
// Конструктор
   Bank(unsigned int id): _bankId(id) {}
// Підвищення процентної ставки для окремого рахунка
   static void raiseInterest(CurrentAccount& account);
// Перевірка стажу окремого рахунка
   static bool ageProof(const CurrentAccount&);
// Перевірка балансу на рахунку
   static bool balanceProof(const CurrentAccount&);
// Селектор-модифікатор
   const unsigned int& getId() const {return _bankId;}
};
```

Філії банку відповідає її банківський ідентифікаційний код `_bankId`, заданий під час створення банку; цей код незмінний. Три атрибути: `_normalInterestRate`, `_advancedInterestRate` і `_goldInterestRate` – задають процентні ставки відповідно для звичайних рахунків, рахунків постійних клієнтів після досягнення ними певного стажу перебування в банку та для заможних клієнтів, залишок на рахунках яких перевищує певний поріг. Оскільки ці норми спільні для всіх філій одного банку, кожній з яких відповідає окремий екземпляр класу, їх задано статичними атрибутами.

Статична функція `raiseInterest(CurrentAccount& account)` переводить рахунок, заданий її параметром, на іншу процентну ставку.

Розглянемо тепер, як можна організувати взаємодію зі статичними методами й атрибутами класу `Bank` у класі рахунків `CurrentAccount`.

Лістинг 5.54

```
// Клас поточних рахунків
class CurrentAccount {
private:
   const unsigned int _accountNo; // Номер рахунка
   static const _freeAccountNo;   // Поточний вільний номер рахунка
   const Date _openingDate;       // Дата створення рахунка
   int _balance;                  // Баланс на рахунку
   const unsigned int& _bankId;   // Ідентифікаційний номер філії банку
// Запроваджена процентна ставка для рахунка
   const double *_interestRatePtr;
// Указник на статичну функцію: підвищення процентної ставки рахунка
```

```
   void(*_raiseInterestPtr)(CurrentAccount&);
public:
// Конструктор
   CurrentAccount(const Bank& bank, int initBalance):
      accountNo(_freeAccountNo),
      _balance(initBalance), _bankId(bank.getId()),
      _interestRatePtr(&Bank::_normalInterestRate),
      _raiseInterestPtr(&Bank::raiseInterest) {}
// Селектори
   unsigned int getNo() const {return _accountNo;}
   int getBalance() const {return _balance;}
   const Date getOpening() const {return _openingDate;}
   unsigned int getId() const {return _bankId;}
   double getInterest() const {return *_interestRatePtr;}
// Модифікатор
   const double*& setInterestRatePtr() {return _interestRatePtr;}
};
```

Визначення цього класу містить досить велику стандартну частину, яка складається з набору зрозумілих атрибутів і їх селекторів і на якій ми не будемо особливо зупинятися. Розглянемо тільки елементи, що принципово важливі для розуміння нашої теми.

Клас `CurrentAccount` має два важливі атрибути, що відсилають до атрибутів класу `Bank`:

```
const unsigned int& _bankId; const double* _interestRatePtr;
```

Вони обидва сталі, але перший із них – відсилка, а другий – указник. Як відомо, відсилка жорстко пов'язана зі своїм об'єктом із моменту створення. Сталість відсилки `_bankId` жорстко пов'язує рахунок із банківською філією. Призначення атрибута зрозуміле: під час створення рахунка до його реквізитів долучають ідентифікаційний код банку. Ініціалізують цю відсилку за допомогою селектора `bank.getId()`.

Другий атрибут `_interestRatePtr` – указник на константу. Під час створення рахунка його теж ініціалізують відповідним атрибутом класу `Bank`, а саме процентною ставкою звичайних рахунків `_interestRatePtr (&Bank::_normalInterestRate)`, й ототожнюють із ним, проте лише на певний час. Зверніть увагу на тип ініціалізованого значення – це адреса атрибута в класі `Bank`. Оскільки атрибут статичний, то інформація про об'єкт не потрібна. Значення атрибута не переписано з класу банків до класу рахунків тому, що всі рахунки одержують значення процентної

ставки з одного місця. Це надає банку можливість у разі потреби залежно від ситуації централізовано змінювати її значення.

У процесі використання рахунка його процентний статус може змінитися. Тоді указник `_interestRatePtr` буде переспрямовано на інший атрибут банку. За це відповідає функція **`const double`**`& setInterestRatePtr()`, яка відіграє роль модифікатора, повертаючи відсилку на указник. Це дає змогу переспрямувати указник процентної ставки на потрібний статичний атрибут класу банків. Таке завдання розв'язує функція **`void`**`(*`**`const`** `_raiseInterestPtr)(CurrentAccount&)`, яку теж ініціалізують відповідною функцією класу `Bank` – `_raiseInterestPtr(&Bank::raiseInterest)`. Позаяк указник не перевстановлюватиметься, його визначено як сталий. Якби йшлося про атрибут, а не про функцію, то тут можна було б скористатися відсилкою.

Нарешті, наведемо спрощений варіант функції змінення процентної ставки:

Лістинг 5.55

```
// Підвищення процентної ставки окремого рахунка
void Bank::raiseInterest(CurrentAccount& account) {
   if(balanceProof(account)) {
      account.setInterestRatePtr()=&_goldInterestRate;
      return;
   }
   if(ageProof(account)) {
      account.setInterestRatePtr()=&_advancedInterestRate;
      return;
   }
   return;
}
```

Вправа 5.13

Проаналізуйте й виконайте наведену далі програму тестування класу `Bank`.

Лістинг 5.56

```
int main() {
   Bank myBank(1); CurrentAccount account1(myBank, 100); cout<<account1;
   Bank::raiseInterest(account1); cout<<account1;
   return 0;
}
```

Вправа 5.14

Доповніть клас `Bank` засобами ведення обліку рахунків.

5.3. Асоціювання об’єктів

Асоціації між об’єктами є одним із найважливіших інструментів об’єктно-орієнтованого програмування, призначених для подолання складності програм завдяки створенню ієрархій об’єктів. Повною мірою вивчати ієрархії ми будемо в розділі 7, але почнемо знайомитись із цим питанням вже тут. Точніше кажучи, спочатку підіб’ємо підсумки попередньої роботи, оскільки все, чим ми займалися досі, зводилося до створення одних об’єктів з інших. Так, клас точок площини складається з дійсних атрибутів, а з точок площини можна побудувати клас трикутників, які утворюють ієрархію другого рівня, узагалі кажучи, незалежну від способу визначення базового рівня. Розумно спроектовані ієрархії – визначальний критерій успішності архітектурних рішень кожного програмного проекту. Асоціація між об’єктами, за якої один являє собою атрибут іншого, називають *вкладенням* (layering). Залежно від типу атрибута вкладення поділяють на *композицію* (зазвичай це простий атрибут) і *агрегацію* (атрибут, приєднаний за допомогою указника чи відсилки).

5.3.1. Композиція об’єктів

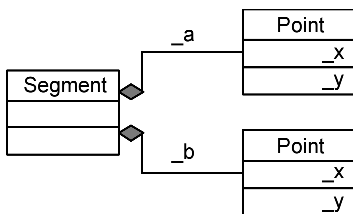


Рис. 5.3. Діаграма класів для відрізків

Знайомство з вкладеннями розпочнемо з композиції. Вона пов’язує об’єкти, які відіграють роль атрибутів, – їх зазвичай називають *компонентами* (component) – зі складеними з них об’єктами – *композитами* (composite object). Один із найпростіших прикладів композитів – відрізок на площині. Його атрибути – точки початку та кінця. Схематично структуру відрізка можна зобразити так (рис. 5.3).

Вивчаючи композицію, будемо водночас вести облік створюваних об’єктів. Тому крім змістовних атрибутів розглянуті класи міститимуть засоби обліку. Визначимо клас відрізків:

Лістинг 5.57

```
// Клас відрізків
class Segment {
private:
// Засоби обліку відрізків
   static unsigned int _freeID; unsigned int _myId;
// Точки початку та кінця відрізка
   Point _a,_b;
```

```
public:
// Конструктори відрізків:
// 1) за координатами їхніх кінців
   Segment(const double x1=0, const double y1=0, const double x2=1,
           const double y2=0);
// 2) за точками початку та кінця
   Segment(const Point& start, const Point& end);
// 3) копіювальний конструктор
   Segment(const Segment&);
   ~Segment();    // Деструктор
   Segment& operator=(const Segment&);    // Присвоєння
// Селектори
   const Point& start() const; const Point& end() const;
   double length() const;    // Обчислення довжини відрізка
// Обчислення відстані від відрізка до точки
   double distance(const Point&) const;
};
```

Важливо добре уявляти собі всі процеси, що проходять під час створення композита. Найважливіша особливість створення полягає у виклику конструкторів усіх компонентів.

Щодо прикладу з лістингу 5.57, бачимо, що компоненти-точки є частинами композита-відрізка, і тому в процесі створення композита буде створено й самі компоненти за допомогою звичайних або копіювальних конструкторів. Так, у першому конструкторі відрізка –

Лістинг 5.58

```
// Конструктор відрізка за координатами його кінців
Segment::Segment (const double x1, const double y1, const double x2,
                  const double y2):
// Ініціалізація вершин за допомогою конструкторів
      _a(x1, y1), _b(x2, y2),
// Ініціалізація ідентифікаційного номера
      _myId(++_freeID) {
// Кероване препроцесором виведення
#ifndef NDEBUG
   out<<myId<<": a Segment was created: "<<*this;
#endif
}
```

використано звичайний конструктор компонентів-точок за їхніми координатами, а в другому конструкторі –

Лістинг 5.59

```
// Конструктор відрізка за точками початку та кінця
Segment::Segment(const Point& a, const Point& b):
```

```
// Ініціалізація вершин за допомогою копіювальних конструкторів
      _a(a), _b(b),
// Ініціалізація ідентифікаційного номера
      _myId(++_freeID) {
// Кероване препроцесором виведення
#ifndef NDEBUG
   out<<myId<<": a Segment was created: "<<*this;
#endif
}
```

точки-компоненти відрізка створює копіювальний конструктор точок.

Вправа 5.15

Реалізуйте копіювальний конструктор відрізків, передбачивши їх обліковування та кероване препроцесором виведення.

Те, що об'єкт являє собою компонент іншого об'єкта – композита, не надає останньому якихось виняткових прав доступу до нього. Закрита частина компонента недоступна композиту, а компоненти, що містяться в закритій частині компонента (зазвичай саме так воно і є), недоступні ззовні клієнтам композита. Тому методи `start()` і `end()` виконують функції селекторів. Зверніть увагу на тип результату селектора: він свідчить про те, що копіювання не відбувається, і селектор указує саме на той екземпляр точки, який міститься всередині відрізка, проте не має права змінювати її.

Лістинг 5.60

```
// Селектори
const Point& Segment::start() const {return _a;}
const Point& Segment::end() const {return _b;}
```

Вправа 5.16

Порівняйте селектори лістингу 5.60 із копіювальними селекторами, наведеними в лістингу 5.61. Складіть тестову програму для підрахунку створених об'єктів у разі застосування селекторів кожного типу. Зробіть висновок про доцільність їх використання.

Лістинг 5.61

```
// Копіювальні селектори
const Point Segment::start() const {return _a;}
const Point Segment::end() const {return _b;}
```

Вправа 5.17

Продемонструйте некоректність поведінки незахищених копіювальних селекторів із лістингу 5.62. У чому полягає їх незахищеність?

```
// Незахищені копіювальні селектори                 Лістинг 5.62
Point Segment::start() const {return _a;}
Point Segment::end() const {return _b;}
```

Зрозуміло, що в разі застосування зазначеного способу створення компонентів тривалість їхнього життя не має перевищувати тривалості життя композита, до якого вони входять. За домовленістю створення та видалення кожного об'єкта має відбуватись у межах однієї програмної структури, у розглядуваному випадку – у межах одного класу. Тому деструктори реалізовано так, щоб видаленню композита передувало видалення всіх його компонентів.

Вправа 5.18

Реалізуйте деструктори відрізків і точок, передбачивши в них керовані препроцесором виведення, і перевірте, у якій послідовності викликаються деструктори компонентів під час видалення композита.

Вправа 5.19

Запрограмуйте функцію виведення відрізка за звичною сигнатурою

```
ostream& operator<<(ostream&, const Segment&);
```

Вправа 5.20

Проаналізуйте, скільки яких об'єктів створить наведена далі програма тестування відрізків. Зафіксуйте моменти створення та видалення кожного з об'єктів.

```
// Програма тестування відрізків                    Лістинг 5.63
int main() {
   Point u, v(0, 1), w(1, 1);
   cout<<"Segment processing================"<<endl;
   Segment s, s1(u, v), s2(v, u), s3(w, v), s4(u, w);
// Створено 5 нових точок і скопійовано 8
   cout<<s<<endl; cout<<"Length="<<s.length()<<endl;
   cout<<w<<endl; cout<<"Distance="<<s.distance(w)<<endl;
   return 0;
}
```

Вправа 5.21

Доповніть клас відрізків модифікаторами початку та кінця відрізка, а також окремо селекторами та модифікаторами їхніх абсцис і ординат, наприклад, за такими сигнатурами:

```
// Селектори-модифікатори точок                          Лістинг 5.64
Point& Segment::start(); Point& Segment::end();
// Селектори координат точок
const double& Segment::startX() const;
const double& Segment::startY() const;
const double& Segment::endX() const; const double& Segment::endY() const;
// Селектори-модифікатори координат точок
double& Segment::startX(); double& Segment::startY();
double& Segment::endX(); double& Segment::endY();
```

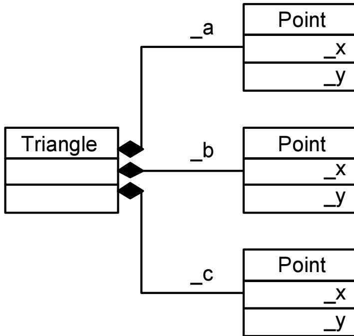


Рис. 5.4. Діаграма класів для трикутників

Розширимо набір геометричних об'єктів іще одним класом – трикутників. Залежно від способу їх визначення виникають різні типи відношень між точками, відрізками та трикутниками.

Найпростіший спосіб побудувати трикутник – це створити композицію трьох його вершин (рис. 5.4). Відрізки, що відповідають сторонам трикутника, явно не входитимуть до композиції. До них повернемося пізніше.

Ось визначення відповідного класу:

```
// Проста композиція                                     Лістинг 5.65
// Клас трикутників, заданих як композиція вершин
class Triangle {
private:
   Point _a, _b, _c; // Вершини трикутника
public:
// Конструктори трикутників:
// 1) за координатами їхніх вершин
   Triangle(const double x1=0, const double y1=0, const double x2=1,
      const double y2=0, const double x3=0, const double y3=1);
// 2) за їхніми вершинами
   Triangle(const Point& a, const Point& b, const Point& c);
// 3) копіювальний конструктор
   Triangle(const Triangle&);
// Деструктор
   ~Triangle();
// Присвоєння
   Triangle& operator=(const Triangle&);
```

```
// Селектори вершин
   const Point& apexA() const; const Point& apexB() const;
   const Point& apexC() const;
// Створення сторін
   const Segment side_a() const; const Segment side_b() const;
   const Segment side_c() const;
   double perimeter() const; // Обчислення периметра
   double area() const; // Обчислення площі
};
```

Конструктори, деструктор і оператор присвоєння реалізовано очевидним способом:

Лістинг 5.66

```
// Конструктор трикутника за координатами його вершин
Triangle::Triangle(const double x1, const double y1, const double x2,
   const double y2, const double x3, const double y3):
// Ініціалізація вершин за допомогою конструкторів
      _a(x1, y1), _b(x2, y2), _c(x3, y3) {}
// Конструктор трикутника за його вершинами
Triangle::Triangle(const Point& a, const Point& b, const Point& c):
// Ініціалізація вершин за допомогою копіювальних конструкторів
      _a(a), _b(b), _c(c) {}
// Копіювальний конструктор
Triangle::Triangle(const Triangle& t):
// Ініціалізація вершин за допомогою копіювальних конструкторів
      _a(t._a), _b(t._b), _c(t._c) {}
// Деструктор
Triangle::~Triangle() {}
// Присвоєння
Triangle& Triangle::operator=(const Triangle& t) {
   _a=t._a; _b=t._b; _c=t._c;
   return *this;
}
```

Селектори-модифікатори `apexA()`, `apexB()` й `apexC()` класу трикутників 5.66 надають доступ до вершин трикутника за тією самою схемою, що й у класі відрізків. Наведемо реалізацію одного з них:

Лістинг 5.67

```
// Селектор вершини A
const Point& Triangle::apexA() const {return _a;}
```

Природно, що сторони трикутника – відрізки, однак у нашій версії трикутника сторони не будуть його компонентами. Проте їх можна побудувати методами `side_a()`, `side_b()` та `side_c()`.

```
// Створення сторони a                                    Лістинг 5.68
const Segment Triangle::side_a() const {return Segment(_b, _c);}
```

Якщо сторони побудовано описаним вище способом, то зв’язок між трикутником і відрізками – його сторонами – уже не буде виражатись у термінах композиції. Тоді говорять про інший тип асоціації між об’єктами, який буде розглянуто в наступному підрозділі.

5.3.2. Залежності між об’єктами

Залежності виникають тоді, коли об’єкти одного класу «знають» про існування іншого класу, але не перебувають у прямих відношеннях вкладення. Про таку «обізнаність» може свідчити наявність імен класів у сигнатурі однієї з функцій – членів класу (байдуже, серед параметрів чи при результаті), а також наявність локальних змінних класового типу в методах іншого класу. Клас, методи якого використовують інші класи у сигнатурі чи реалізації, завдяки цьому теж «обізнаний» про ці класи. Між об’єктами «обізнаних» один про одного класів виникають відношення, які називають залежністю. Так, клас трикутників (лістинг 5.65) є композицією вершин, але його методи створення сторін залежать від класу відрізків. Залежності бувають несиметричними. Наприклад, трикутник «обізнаний» зі своїми сторонами, а відрізкам, якими ці сторони виражено, нічого «не відомо» про існування трикутників.

На відміну від композиції, яка припиняє існування компонентів одночасно з припиненням існування композиту, залежність не забороняє об’єктам мати власний життєвий цикл, що не обмежується рамками життєвого циклу композита. Так, скажімо, видалення трикутника не призводить до автоматичного видалення створених у ньому сторін. В кожному конкретному випадку цю задачу доводиться розглядати окремо.

Вправа 5.22

Складіть тестову програму для визначення тривалості життя відрізка, створеного як сторона заданого трикутника з лістингу 5.65. Як видалити залежний об’єкт?

Розглянемо невеликий приклад. Нехай задано якийсь трикутник `ABC`, до якого буде застосовано метод `side_a()`. Це зумовлює виникнення асоціації між трикутником `ABC` та його стороною `a`, як показано на рис. 5.5, на якому зображено трикутник `ABC` та позначено сторону `BC` з її власними компонентами.

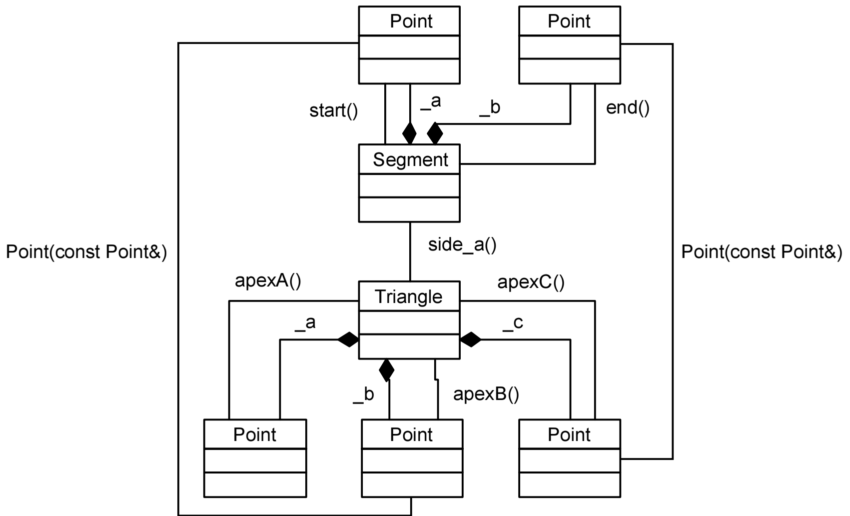


Рис. 5.5. Діаграма асоціаціювання класу трикутників з класами відрізків і точок площини

Аналізуючи сигнатуру методів трикутника, можна помітити інші асоціації, зокрема породжені методами доступу до вершин `apexA()`, `apexB()` й `apexC()`. У розглядуваному випадку асоційовані об'єкти виявляються компонентами, але далі перейдемо до інших способів визначення трикутників, за яких зміниться характер зв'язку трикутника з його вершинами. Так само кінці відрізка, з одного боку, являють собою його компоненти, з іншого – утворюють асоціацію з ним за допомогою методів `start()` і `end()`. Самі ж кінці відрізка `BC` асоційовано з їх оригіналами – вершинами `B` та `C` трикутника `ABC` – за допомогою відповідних копіювальних конструкторів.

Вправа 5.23

Доповніть трикутник із лістингу 5.65 конструкторами:

- за стороною та протилежною вершиною, користуючись сигнатурою

```
Triangle::Triangle (const Segment&, const Point&);
```

- за стороною та прилеглими кутами, користуючись сигнатурою

```
Triangle::Triangle (const Segment&, double, double);
```

Методи `perimeter()` і `area()` обчислюють відповідно периметр і площу трикутника.

```
// Обчислення периметра                                    Лістинг 5.69
double Triangle::perimeter() const {
   return side_a().length()+side_b().length()+side_c().length();
}
```

Для обчислення площі трикутника використано відому шкільну формулу:

```
// Обчислення площі                                        Лістинг 5.70
double Triangle::area() const {
   return 0.5*side_a().length()*side_a().distance(_a);
}
```

Вправа 5.24

Реалізуйте облік допоміжних об’єктів – точок і відрізків, створених для обчислення периметра та площі трикутника з лістингу 5.65. Запропонуйте власні ефективні рішення.

Вправа 5.25

Доповніть клас трикутників самостійними селекторами і модифікаторами його вершин за зразком лістингу 4.41, відмінними від селекторів, суміщених з модифікаторами (лістинг 5.65). Запропонуйте відповідні сигнатури.

Доповнимо клас трикутників з лістингу 5.65 методом візуалізації – наприклад, із такою реалізацією:

```
// Візуалізація трикутника                                 Лістинг 5.71
ostream& Triangle::show(ostream& os) const {
// Виведення вершин
   os<<"Triangle: "<<apexA()<<apexB()<<apexC()<<endl;
// Виведення сторін
   os<<side_a(); os<<side_b(); os<<side_c();
// Виведення числових характеристик
   os<<"perimeter "<<perimeter()<<endl; os<<"area "<<area()<<endl<<endl;
   return os;
}
```

Вправа 5.26

Запрограмуйте у вигляді утиліти класу операцію виведення трикутників з лістингу 5.65, використовуючи запропонований метод (лістинг 5.71). Реалізуйте облік допоміжних об’єктів – точок і відрізків, які буде створено під час виведення. Запропонуйте власні ефективні рішення.

У лістингу 5.72 наведено тестову програму обробки трикутників.

```
// Тестова програма для класу трикутників
int main() {
// Створення трикутника за заданим набором вершин
   Point u, v(0, 3), w(4);
   cout<<"Triangle composing apices======"<<endl;
   Triangle ABC(u, v, w); cout<<"ABC "<<ABC;
// Створення трикутника за замовчуванням
   Triangle a; cout<<"a "<<a;
// Створення трикутника за координатами вершин
   Triangle a, b(2, 0, 2, 2, 0, 2), c(b); cout<<"b "<<b;
// Копіювання трикутника
   Triangle a, b(2, 0, 2, 2, 0, 2), c(b); cout<<"c "<<c;
// Перевірка присвоєння
   c=ABC; cout<<"new value of c "<<c;
   return 0;
}
```

Лістинг 5.72

Результат виконання програми буде таким:

```
Triangle composing apices======
ABC Triangle: (0,0)(0,3)(4,0)
[(0,3),(4,0)]
[(4,0),(0,0)]
[(0,0),(0,3)]
perimeter 12
area 6
a Triangle: (0,0)(1,0)(0,1)
[(1,0),(0,1)]
[(0,1),(0,0)]
[(0,0),(1,0)]
perimeter 3.41421
area 0.5
b Triangle: (2,0)(2,2)(0,2)
[(2,2),(0,2)]
[(0,2),(2,0)]
[(2,0),(2,2)]
perimeter 6.82843
area 2
c Triangle: (2,0)(2,2)(0,2)
[(2,2),(0,2)]
[(0,2),(2,0)]
[(2,0),(2,2)]
perimeter 6.82843
```

Лістинг 5.73

```
area 2
new value of c Triangle: (0,0)(0,3)(4,0)
[(0,3),(4,0)]
[(4,0),(0,0)]
[(0,0),(0,3)]
perimeter 12
area 6
```

Вправа 5.27

Виконайте облік допоміжних об'єктів – точок і відрізків, створених наведеною в лістингу 5.72 тестовою програмою. Локалізуйте в програмі місця їх створення та видалення.

Вправа 5.28

З'ясуйте, за яких умов застосування формули обчислення площі з лістингу 5.70 впливає на кількість створюваних і копійованих точок і сторін. Порівняйте кількість об'єктів, які буде створено тестовою програмою з попередньої вправи, для різних реалізацій функції обчислення площі. Результати поясніть.

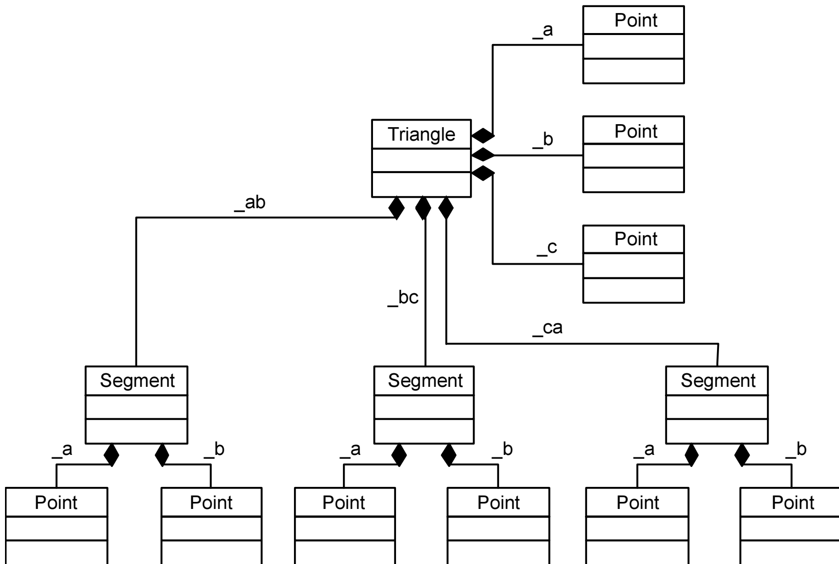


Рис. 5.6. Діаграма класів для трикутників, заданих композицією вершин і сторін

Тепер розглянемо інший спосіб компонування трикутників, узявши сторони як додаткові компоненти. Якщо при цьому залишити серед компонентів і вершини, то виникне варіант такої собі надлишкової композиції (рис. 5.6).

Зверніть увагу на те, що кожну точку буде представлено в такому трикутнику тричі: як його власну вершину, початок однієї сторони та кінець іншої. Визначення класу набуде такого вигляду:

Лістинг 5.74

```
// Подвійна композиція
// Клас трикутників, заданих як композиція вершин і сторін
class Triangle {
private:
   Point _a, _b, _c;      // Вершини трикутника
   Segment _ab, _bc, _ca; // Сторони трикутника
public:
// Конструктори трикутника:
// 1) за координатами його вершин
   Triangle(const double x1=0, const double y1=0,
      const double x2=1, const double y2=0,
      const double x3=0, const double y3=1);
// 2) за його вершинами
   Triangle(const Point& a, const Point& b, const Point& c);
// 3) копіювальний конструктор
   Triangle(const Triangle&);
   ~Triangle();   // Деструктор
   Triangle& operator=(const Triangle&);   // Присвоєння
// Селектори вершин
   const Point& apexA() const; const Point& apexB() const;
   const Point& apexC() const;
// Селектори сторін
   const Segment& side_a() const; const Segment& side_b() const;
   const Segment& side_c() const;
   double perimeter() const;         // Обчислення периметра
   double area() const;              // Обчислення площі
   ostream& show(ostream& os) const; // Візуалізація трикутника
};
// Виведення трикутника
ostream& operator<<(ostream&, const Triangle&);
```

Відповідно скоректуємо конструктори й оператор присвоєння:

Лістинг 5.75

```
// Конструктор трикутника за координатами його вершин
Triangle::Triangle(const double x1, const double y1, const double x2,
```

```
   const double y2, const double x3, const double y3):
// Ініціалізація вершин за допомогою конструкторів
      _a(x1, y1), _b(x2, y2), _c(x3, y3),
// Ініціалізація сторін за допомогою конструкторів
      _ab(_a, _b),_bc(_b, _c), _ca(_c, _a) {}
// Конструктор трикутника за його вершинами
Triangle::Triangle(const Point& a, const Point& b, const Point& c):
// Ініціалізація вершин за допомогою копіювальних конструкторів
      _a(a), _b(b), _c(c),
// Ініціалізація сторін за допомогою конструкторів
      _ab(a, b),_bc(b, c), _ca(c, a) {}
// Копіювальний конструктор
Triangle::Triangle(const Triangle& t):
// Ініціалізація вершин за допомогою копіювальних конструкторів
      _a(t._a), _b(t._b), _c(t._c),
// Ініціалізація сторін за допомогою копіювальних конструкторів
      _ab(t._ab), _bc(t._bc), _ca(t._ca) {}
// Присвоєння
Triangle& Triangle::operator=(const Triangle& t) {
   _a=t._a; _b=t._b; _c=t._c;
   _ab=t._ab; _bc=t._bc; _ca=t._ca;
   return *this;
}
```

Зверніть увагу на те, що функції доступу до сторін перетворилися на селектори: сторони тепер не потрібно створювати, бо вони є в трикутнику як компоненти. Відповідно зміниться реалізація:

Лістинг 5.76

```
// Селектор сторони BC
const Segment& Triangle::side_a() const {
// У попередній версії - Segment(_b, _c);
   return _bc;
}
```

На перший погляд може здатися, що надлишкова композиція неефективна, але, знову запустивши ту саму тестову програму з лістингу 5.72, можна пересвідчитися, що кількість створюваних і скопійованих об’єктів може виявитися значно меншою.

Вправа 5.29

Виконайте облік допоміжних об’єктів – точок і відрізків, створених наведеною в лістингу 5.72 тестовою програмою, яку застосовано до класу трикутників подвійної композиції з лістингу 5.74. Порівняйте отримані

дані з результатами роботи програми з вправи 5.27. З'ясуйте, створення яких саме об'єктів удається уникнути, і чому.

Правда, надлишкова композиція ускладнює модифікацію як вершин, так і сторін. Змінюючи вершину, потрібно потурбуватися про модифікацію прилеглих до неї сторін, а змінюючи сторону – відповідно про вершини й інші сторони.

Вправа 5.30

Доповніть клас трикутників подвійної композиції з лістингу 5.74 модифікаторами вершин за зразком вправи 5.25. Визначте і реалізуйте також модифікатори сторін. Поясніть відсутність у визначенні розглянутого класу трикутників селекторів, суміщених з модифікаторами. Зверніть увагу на необхідність узгодження модифікацій сторін і вершин одне з одним.

Важливий спосіб організації взаємодії між композитом і компонентом – делегування, уже згадане в підрозділі 5.1. Воно полягає в тому, що композит бере на себе відповідальність за певні дії, повноваження на виконання яких передано компоненту. Так компонент, недоступний ззовні композиту, може замість композита відповідати на запити, подані тому його клієнтами.

Лістинг 5.77

```
// Подвійна композиція, доповнена делегуванням
// Клас трикутників, заданих як композиція вершин і сторін
class Triangle {
private:
   Point _a, _b, _c; // Вершини трикутника
   Segment _ab, _bc, _ca; // Сторони трикутника
public:
// Конструктори трикутника:
// 1) за координатами його вершин
   Triangle(const double x1=0, const double y1=0, const double x2=1,
      const double y2=0, const double x3=0, const double y3=1);
// 2) за його вершинами
   Triangle(const Point& a, const Point& b, const Point& c);
// 3) копіювальний конструктор
   Triangle(const Triangle&);
   ~Triangle();   // Деструктор
   Triangle& operator=(const Triangle&);   // Присвоєння
// Селектори вершин
   const Point& apexA() const; const Point& apexB() const;
   const Point& apexC() const;
```

```
// Селектори сторін
   const Segment& side_a() const; const Segment& side_b() const;
   const Segment& side_c() const;
// Модифікатори вершин
   Triangle& setApexA (const Point&);
   Triangle& setApexB (const Point&);
   Triangle& setApexC (const Point&);
// Модифікатори сторін
   Triangle& setSide_a (const Segment&);
   Triangle& setSide_b (const Segment&);
   Triangle& setSide_c (const Segment&);
   double perimeter() const;   // Обчислення периметра
   double area() const;   // Обчислення площі
   ostream& show(ostream& os) const;   // Візуалізація трикутника
// Делегування: обчислення довжин сторін
   double length_a() const; double length_b() const;
   double length_c() const;
};
ostream& operator<<(ostream&, const Triangle&); // Виведення трикутника
```

Останні три методи класу `Triangle` (наведемо лише один з них) делегують обчислення довжини сторони відповідним методам класу `Segment`, об'єкти якого – компоненти трикутника.

Лістинг 5.78

```
// Делегування класу відрізків:
// обчислення довжини сторони _a
double Triangle::length_a() const {return _bc.length();}
```

Завдяки тому, що в композиті є делеговані методи, у реалізаціях інших методів не потрібно прямо звертатися до компонентів, залишивши цю функцію за делегатами. Перепишемо метод обчислення периметра, використовуючи делегування замість прямого виклику методів класу сегментів:

```
// Обчислення периметра трикутника
double Triangle::perimeter() const {
   return length_a()+length_b()+length_c();
}
```

Тому вираз

```
length_a()+length_b()+length_c();
```

кращий, ніж, скажімо, вираз

```
_bc.length()+_ca.length()+_ab.length();
```

оскільки в ньому менше зв'язків між композитом і компонентом. Порівнюючи ці два рішення, варто брати до уваги не лише ефективність реалізації, але й питання її політики. Наприклад, якщо серед методів класу є певний метод (тут – метод обчислення довжини сторони), то в реалізаціях усіх інших методів класу має бути саме використано його, а не метод компонента, хоча він доступний. Справді, закрита частина класу, у якій, зокрема, містяться визначення компонентів, може змінитися в процесі розробки класу. При цьому, як побачимо далі, самі компоненти можуть бути видалені й замінені іншими. Згідно з загальними правилами розробки програмних систем не варто модифікувати відкриту частину класу, оскільки вона стосується його клієнтів. Реалізація композита буде стійкішою до впливу змін, якщо скрізь, де це можливо, вона надаватиме перевагу використанню членів його відкритої, а не закритої частини.

Вправа 5.31

Перевірте, чи вплинуло делегування обчислення довжин сторін на ефективність виконання тестової програми з лістингу 5.72.

Застосовуючи делегування до функції обчислення площі, доцільно було б розширити набір методів класу трикутників методами обчислення довжин висот.

Вправа 5.32

Доповніть клас трикутників з лістингу 5.77 методами обчислення довжин висот за такими сигнатурами:

Лістинг 5.79

```
// Делегування: обчислення довжин висот
double Triangle::length_ha() const; double Triangle::length_hb() const;
double Triangle::length_hc() const;
```

Тоді реалізація методу обчислення площі набуде вигляду

Лістинг 5.80

```
// Площа трикутника
double Triangle::area() const {return 0.5*length_a()*length_ha();}
```

Як уже було зазначено, класу трикутників подвійної композиції, визначеному в лістингу 5.74, властива певна надлишковість. Вона полягає в дублюванні інформації про вершини трикутника, які входять до нього як компоненти на додаток до того, що кожна вершина – водночас компонент двох прилеглих до неї сторін. Побудований таким способом трикутник тричі містить кожну зі своїх вершин. З іншого боку, хоча наявність у трикутнику

його сторін як компонентів і спричиняє надлишковість, це дає змогу уникнути неефективності першого подання трикутника як простої композиції вершин (лістинг 5.65). Джерело такої неефективності – потреба повторно утворювати сторони для кожного обчислення з їх використанням. У конструкції трикутника не передбачено можливості їх зберігання, а тому доводиться заново будувати сторону для кожного наступного запиту.

Тепер поставимо собі запитання: наскільки важливою була наявність вершин безпосередньо як компонентів трикутника? Позаяк усі вершини – компоненти прилеглих до них сторін, розглянемо третій варіант композиції в класі трикутників – за трьома сторонами (рис. 5.7).

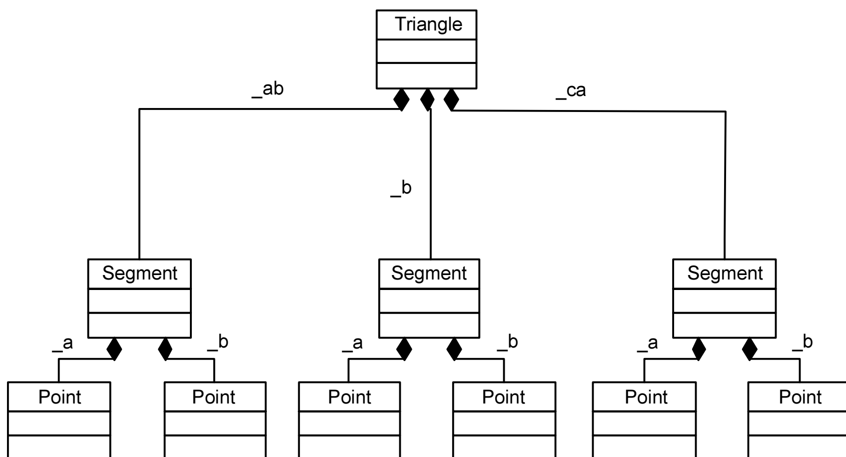


Рис. 5.7. Діаграма класу трикутників, заданих композицією сторін

Ця композиція також надлишкова з точки зору дублювання вершин суміжних сторін, однак до неї кожна вершина входить не тричі, як раніше у класі трикутників подвійної композиції з лістингу 5.74, а двічі. Наведемо визначення цієї версії класу трикутників:

Лістинг 5.81

```
// Усе ще надлишкова композиція
// Клас трикутників, заданих як композиція сторін
class Triangle {
private:
// Сторони трикутника
   Segment _ab, _bc, _ca;
public:
// Конструктори трикутника:
```

```
// 1) за координатами його вершин
   Triangle(const double x1=0, const double y1=0, const double x2=1,
      const double y2=0, const double x3=0, const double y3=1);
// 2) за його вершинами
   Triangle(const Point& a, const Point& b, const Point& c);
// 3) копіювальний конструктор
   Triangle(const Triangle&);
   ~Triangle();   // Деструктор
   Triangle& operator=(const Triangle&);   // Присвоєння
// Обчислення вершин
   const Point apexA() const; const Point apexB() const;
   const Point apexC() const;
// Селектори сторін
   const Segment& side_a() const; const Segment& side_b() const;
   const Segment& side_c() const;
   double perimeter() const;   // Обчислення периметра
   double area() const;   // Обчислення площі
   ostream& show(ostream& os) const;   // Візуалізація трикутника
// Делегування: обчислення довжин сторін
   double length_a() const; double length_b() const;
   double length_c() const;
// Обчислення довжин висот
   double length_ha() const; double length_hb() const;
   double length_hc() const;
};
// Виведення трикутника
ostream& operator<<(ostream&, const Triangle&);
```

Тепер трохи зміняться реалізації конструкторів, а також оператора присвоєння:

Лістинг 5.82

```
// Конструктор трикутника за координатами його вершин
Triangle::Triangle(const double x1, const double y1, const double x2,
   const double y2, const double x3, const double y3):
// Ініціалізація сторін за допомогою конструкторів
      _ab(Segment(x1, y1, x2, y2)), _bc(Segment(x2, y2, x3, y3)),
      _ca(Segment(x3, y3, x1, y1)) {}
// Конструктор трикутника за його вершинами
Triangle::Triangle(const Point& a, const Point& b, const Point& c):
// Ініціалізація сторін за допомогою конструкторів
      _ab(Segment(a, b)), _bc(Segment(b, c)), _ca(Segment(c, a)) {}
// Копіювальний конструктор
Triangle::Triangle(const Triangle& t):
```

```
// Ініціалізація сторін за допомогою копіювальних конструкторів
      _ab(t._ab), _bc(t._bc), _ca(t._ca) {}
// Присвоєння
Triangle& Triangle::operator=(const Triangle& t) {
   _ab=t._ab; _bc=t._bc; _ca=t._ca;
   return *this;
}
```

Вправа 5.33

Визначте і реалізуйте модифікатори сторін для класу трикутників з лістингу 5.81. Порівняйте з реалізаціями модифікаторів сторін в класі трикутників подвійної композиції (вправа 5.30).

Вправа 5.34

Визначте і реалізуйте методи обчислення вершин трикутника для класу трикутників з лістингу 5.81.

Цікаво порівняти одержані три версії класу трикутників із погляду ефективності використання. Відповідь може дати така вправа.

Вправа 5.35

Виконайте облік допоміжних об'єктів – точок і відрізків, створених наведеною в лістингу 5.72 тестовою програмою, яку застосовано до класу трикутників, заданих як композиція їхніх сторін, із лістингу 5.81. Порівняйте з результатами випробувань для класу трикутників, заданих композицією вершин, і класу трикутників подвійної композиції.

Підбиваючи попередні підсумки, можна сказати, що перевага подвійної композиції над звичайною в задачах обробки трикутників і відрізків полягає у постійній наявності сторін як допоміжних компонентів, для побудови яких не потрібні додаткові виклики. При цьому, щоправда, виникає дублювання вершин і ускладнюється їх модифікація. Розглянемо інший спосіб вкладання об'єктів, який дає можливість поєднувати переваги подвійної композиції з простотою доступу звичайної.

5.3.3. Агрегати об'єктів

Поряд із композицією широко застосовують іще один спосіб вкладання одного об'єкта в інший, який називають *агрегацією* (aggregation). На відміну від компонент, які з'являються разом із композитом і наявні в ньому аж до його видалення, *агреговані об'єкти* (aggregated instance, чи aggregee) можуть бути створені в *агрегаті* (aggregate) лише в разі потреби чи існувати поза ним.

Агреговані атрибути можна поділити на два типи залежно від відповідальності за їх створення та видалення. До першого типу відносять атрибути, які безпосередньо належать об'єктові-агрегату. Він їх створює та несе відповідальність за їх видалення. Такі атрибути можна назвати *необов'язковими* (optional). Необов'язковий атрибут пов'язують з агрегатом за допомогою указника. Як відомо, указники можуть бути невизначеними, тому звертатися до такого атрибута слід завжди обережно, спочатку перевіряючи його визначеність.

Зауваження. *До наведених раніше аргументів на користь закритих атрибутів (нагадуємо, це була потреба їх захисту від несанкціонованих змін) можна додати ще один: потреба захисту атрибутів не тільки від модифікації, але й від простого читання їх значень. Справді, за прямого доступу до необов'язкового атрибута перевіряти його визначеність має клієнт класу, тоді як доступ за допомогою селектора відіграє роль запобіжника від спроби вжити невизначений атрибут. Отже, можливість гарантувати попередню перевірку в разі звертання до атрибута – іще одна причина, з якої класи закривають клієнтам прямий доступ до своїх атрибутів.*

Другий тип атрибутів – це відсилки до об'єктів, що містяться за межами агрегату. Його буде розглянуто трохи пізніше.

Отже, агрегацією називають вкладання об'єкта за допомогою указника чи відсилки. Розглянемо ці два різновиди агрегації детальніше.

Агрегація указниками

Розглянемо ще одну версію класу `Triangle`. Тепер вершини залишаться компонентами трикутника, тоді як сторони агреговані в нього.

Лістинг 5.83

```
// Агрегація за допомогою указників
// Клас трикутників, визначених композицією вершин з агрегацією сторін
class Triangle {
// Визначаємо тип указника SidePtr на сторону трикутника
   typedef Segment* Triangle :: *SidePtr;
private:
   Point _a, _b, _c;                    // Вершини трикутника
   mutable Segment *_ab, *_bc, *_ca; // Указники на сторони трикутника
public:
// Конструктори трикутника:
// 1) за координатами його вершин
   Triangle(const double x1=0, const double y1=0, const double x2=1,
      const double y2=0, const double x3=0, const double y3=1);
// 2) за його вершинами
   Triangle(const Point& a, const Point& b, const Point& c);
```

```
// 3) копіювальний конструктор
   Triangle(const Triangle&);
   ~Triangle();   // Деструктор
   Triangle& operator=(const Triangle&);   // Скорочене присвоєння
// Обчислення вершин
   const Point apexA() const; const Point apexB() const;
   const Point apexC() const;
// Селектори сторін
   const Segment& side_a() const; const Segment& side_b() const;
   const Segment& side_c() const;
// Селектори-модифікатори сторін
   Segment& side_a(); Segment& side_b(); Segment& side_c();
   double perimeter() const;   // Обчислення периметра
   double area() const;   // Обчислення площі
   ostream& show(ostream& os) const;   // Візуалізація трикутника
// Делегування: обчислення довжин сторін
   double length_a() const; double length_b() const;
   double length_c() const;
// Обчислення довжин висот
   double length_ha() const; double length_hb() const;
   double length_hc() const;
};
ostream& operator<<(ostream&, const Triangle&); // Виведення трикутника
```

Особливість агрегації полягає в тому, що необов'язковий атрибут може залишатися невизначеним, доки він не знадобиться. Його визначення ініціюється першим запитом до нього. Один раз обчислений атрибут буде запам'ятовано, і наступні запити до нього не потребуватимуть повторних обчислень. Водночас частина атрибутів, до яких не відбулося запитів під час виконання програми, так і залишаться невизначеними. Такий підхід до обчислень набув особливої популярності в розподілених системах, де він одержав назву принципу *керованих запитами*, або *замовних*, *обчислень* (demand-driven calculation, calculation on demand).

Тепер конструктори набудуть такого вигляду:

Лістинг 5.84

```
// Конструктор трикутника за координатами його вершин
Triangle::Triangle(const double x1, const double y1, const double x2,
      const double y2, const double x3, const double y3):
// Ініціалізація вершин за допомогою конструкторів
      _a(x1, y1), _b(x2, y2), _c(x3, y3),
// Ініціалізація невизначених сторін
      _ab(0), _bc(0), _ca(0) {}
```

```
// Конструктор трикутника за його вершинами
Triangle::Triangle(const Point& a, const Point& b, const Point& c):
// Ініціалізація вершин за допомогою копіювальних конструкторів
      _a(a), _b(b), _c(c),
// Ініціалізація невизначених сторін
      _ab(0), _bc(0), _ca(0) {}
```

Достатньо один раз звернутися до котроїсь зі сторін, як її буде створено та запам'ятовано. Наведемо приклад обчислення сторони _bc:

Лістинг 5.85

```
// Обчислення та запам'ятовування сторони _bc
const Segment& Triangle::side_a() const {
   if(_bc==0) _bc=new Segment(_b, _c);
   return *_bc;
}
```

Така конструкція класу трикутників впливає на концепцію копіювання за допомогою копіювального конструктора та копіювального присвоєння. Вони виконують не повне, а лише часткове копіювання трикутника. Виходячи з концепції замовних обчислень, повне копіювання потрібно програмувати особливо, про що йтиметься пізніше. Наведемо текст реалізації копіювального конструктора:

Лістинг 5.86

```
// Копіювальний конструктор
Triangle::Triangle(const Triangle& t):
// Ініціалізація вершин за допомогою копіювальних конструкторів
      _a(t._a), _b(t._b), _c(t._c),
// Ініціалізація невизначених сторін
      _ab(0), _bc(0), _ca(0) {
   return;
}
```

та присвоєння:

Лістинг 5.87

```
// Скорочене присвоєння вершин (без сторін)
Triangle& Triangle::operator=(const Triangle& t) {
// Копіювання вершин
   _a=t._a; _b=t._b; _c=t._c;
// Видалення старих сторін
   delete _ab; delete _bc; delete _ca;
   _ab=_bc=_ca=0;
   return *this;
}
```

Позаяк виникла динамічна пам'ять, потрібно задати нетривіальний деструктор:

```
// Деструктор                                              Лістинг 5.88
Triangle::~Triangle() {
   delete _ab; delete _bc; delete _ca;
   _ab=_bc=_ca=0;
   return;
}
```

Вправа 5.36

Виконайте тестову програму з лістингу 5.72 і проведіть обліковування об'єктів для класу трикутників з агрегованими за допомогою указників сторонами (з лістингу 5.84). Результати порівняйте з даними попередніх обліків (вправи 5.27 , 5.29 і 5.35).

Вправа 5.37

Реалізуйте селектори та модифікатори вершин і сторін трикутника з агрегованими за допомогою указників сторонами (з лістингу 5.83).

Нарешті, доповнимо клас трикутників засобами повного копіювання на замовлення.

```
// Агрегація за допомогою указників                        Лістинг 5.89
// Клас трикутників, визначених композицією вершин з агрегацією сторін
class Triangle {
// Визначаємо тип указника SidePtr на сторону трикутника
   typedef Segment* Triangle::* SidePtr;
private:
   Point _a, _b, _c; // Вершини трикутника
   mutable Segment *_ab, *_bc, *_ca; // Указники на сторони трикутника
// Функція підтримки копіювання
   void assignSegment (const SidePtr, const Triangle&);
public:
// Конструктори трикутника:
// 1) за координатами його вершин
   Triangle(const double x1=0, const double y1=0, const double x2=1,
      const double y2=0, const double x3=0, const double y3=1);
// 2) за його вершинами
   Triangle(const Point& a, const Point& b, const Point& c);
// 3) Повний копіювальний конструктор
   Triangle(const Triangle&, const bool full=false);
   ~Triangle();   // Деструктор
```

```
// Скорочене присвоєння вершин (без сторін)
   Triangle& operator=(const Triangle&);
// Повне присвоєння вершин і сторін
   Triangle& assign(const Triangle&);
// Обчислення вершин
   const Point apexA() const; const Point apexB() const;
   const Point apexC() const;
// Селектори сторін
   const Segment& side_a() const; const Segment& side_b() const;
   const Segment& side_c() const;
// Селектори-модифікатори сторін
   Segment& side_a(); Segment& side_b(); Segment& side_c();
   double perimeter() const;   // Обчислення периметра
   double area() const;   // Обчислення площі
   ostream& show(ostream& os) const;   // Візуалізація трикутника
// Делегування: обчислення довжин сторін
   double length_a() const; double length_b() const;
   double length_c() const;
// Обчислення довжин висот
   double length_ha() const; double length_hb() const;
   double length_hc() const;
};
// Виведення трикутника
ostream& operator<<(ostream&, const Triangle&);
```

У копіювальному конструкторі є додатковий заданий за замовчуванням параметр, від якого залежить, чи потрібно повністю копіювати як вершини, так і наявні сторони.

Зауваження. *За способом застосування неповний копіювальний конструктор трикутників – певний аналог конструктора мультикопіювання рядків із лістингу 4.157 (поясніть, чому).*

Лістинг 5.90

```
// Повний копіювальний конструктор
Triangle::Triangle(const Triangle& t,const bool full):
// Ініціалізація вершин за допомогою копіювальних конструкторів
   _a(t._a), _b(t._b), _c(t._c),
// Ініціалізація невизначених сторін
   _ab(0), _bc(0), _ca(0) {
   if(full) {
       assignSegment(&Triangle::_ab, t);
       assignSegment(&Triangle::_bc, t);
       assignSegment(&Triangle::_ca, t);
    };
```

```
   return;
}
```

Власне для копіювання використано службову функцію копіювання сторони трикутника `assignSegment`, яку розглянемо пізніше.

Копіювальний конструктор керує глибиною копіювання за допомогою додаткового параметра, а стосовно оператора присвоєння доведеться шукати інше рішення. Керуючись принципом обчислень на замовлення, залишимо за оператором присвоєння просту функцію копіювання вершин.

Повне присвоєння виконуватиме спеціальна функція `assign`:

Лістинг 5.91

```
// Повне присвоєння вершин і сторін
Triangle& Triangle::assign(const Triangle&) {
// Копіювання вершин
   _a=t._a; _b=t._b; _c=t._c;
// Копіювання сторін
   assignSegment(&Triangle::_ab, t); assignSegment(&Triangle::_bc, t);
   assignSegment(&Triangle::_ca, t);
   return *this;
}
```

У копіювальному конструкторі та функції присвоєння застосовано службову функцію підтримки копіювання `assignSegment`. Її влаштовано так, щоб вона підтримувала копіювання як у присвоєнні, так і в копіювальному конструкторі.

Лістинг 5.92

```
// Функція підтримки копіювання
// Копіює одну сторону side з об'єкта s до цільового об'єкта this
void Triangle::assignSegment (const SidePtr side, const Triangle& s) {
   if((*this).*side==0) {
// 1) Сторону цілі не задано
// (випадок присвоєння чи конструктора)
      if(s.*side==0)
// 1.1) сторону джерела не задано: не робимо нічого
         ;
      else
// 1.2) сторону джерела задано: створюємо нову сторону та додаємо її
// до цільового об'єкта
         (*this).*side=new Segment(*(s.*side));
   }
   else {
// 2) Сторону цілі задано (тільки в разі присвоєння)
      if(s.*side==0) {
// 2.1) сторону джерела не задано: видаляємо сторону в цільовому об'єкті
```

```
        delete((*this).*side);
        (*this).*side=0;
      }
      else
// 2.2) сторону джерела задано: копіюємо джерело до цілі
        *((*this).*side)=*(s.*side);
   }
   return;
}
```

Нагадаємо, що в контексті класу трикутників діє визначення типу `SidePtr` указника на його сторону

```
typedef Segment* Triangle:: *SidePtr;
```

Застосування техніки типізації дає змогу визначити сигнатуру одного методу для присвоєння довільної сторони в довільному трикутнику

```
void assignSegment (const SidePtr side, const Triangle& t);
```

Тут перший параметр слід інтерпретувати як «указник у класі `Triangle` на атрибут, виражений указником на відрізок `Segment`», або повністю

```
void assignSegment (const Segment* Triangle:: *side, const Triangle& t);
```

без визначення типу. Важко цілком однозначно стверджувати, що визначення типу краще, ніж повний запис, оскільки з останнього запису точно видно, що параметр передається указником, до того ж другого рівня.

Без застосування типізації атрибутів довелося б виписувати три окремі функції для кожної зі сторін. Використання параметра, типізованого відповідним атрибутом, дає можливість застосовувати одну ту саму функцію для присвоєння кожної зі сторін, переданої як параметр:

Лістинг 5.93

```
assignSegment(&Triangle::_ab, t);
assignSegment(&Triangle::_bc, t); assignSegment(&Triangle::_ca, t);
```

Тут важливо, що один і той самий параметр `side` типу `SidePtr` відсилає до потрібного атрибута як у цільовому об'єкті **`this`**, так і в джерелі копіювання, вираженому параметром `t`.

Вправа 5.38

Виходячи з концепції замовних обчислень, доповніть клас трикутників методами побудови:

- висот;
- медіан;

* бісектрис.

Виявіть методи, реалізація яких потребує уточнень, і модифікуйте її.

Завершуючи порівняння різних способів вкладання об'єктів, повернімося до методу створення сторони в останньому класі трикутників (лістинг 5.89). Кожну зі сторін створює конструктор відрізка, наприклад `Segment(_a, _b)`, який копіює атрибути об'єкта класу трикутників у атрибути об'єкта класу відрізків. Спробуємо з'ясувати, чи потрібне таке копіювання. Візьмемо довільний трикутник `t` з вершинами `t._a`, `t._b`, `t._c`. Під час створення сторони `*_ab` трикутника `t`, на яку вказує його атрибут – указник `t._ab`, у ній буде створено точки `t._ab->_a` й `t._ab->_b`, що стануть кінцями відповідного відрізка. Тривалість життя відрізка `*t._ab` не може перевищувати тривалості життя самого трикутника `t`, оскільки в разі видалення трикутника буде видалено і його сторони. Звідси випливає надлишковість копіювання вершин до відрізків. Натомість сторони трикутника можуть посилатися на відповідні його вершини.

Агрегація відсилками

Тепер займемося другим типом вкладених атрибутів – приєднаних до агрегату відсилкою. Цей тип агрегації називатимемо *агрегацією за допомогою відсилок*. Доведеться визначити новий клас відрізків, призначених для випростання як сторони трикутників. Його визначаємо як клас, вкладений до класу трикутників.

Лістинг 5.94

```
#include "Triangle.h"
// Клас відрізків, вкладених як сторони трикутників
class Triangle::Segment {
private:
// Засоби обліковування відрізків
   static unsigned int _freeID; unsigned int _myId;
// Відсилка до вершини трикутника, що являє собою точку початку відрізка
   const Point& _a;
// Відсилка до вершини трикутника, що являє собою точку кінця відрізка
   const Point& _b;
// Не реалізовані, а тому закриті:
// 1) копіювальний конструктор
   Segment(const Segment&);
// 2) присвоєння
   Segment& operator=(const Segment&);
public:
// Конструктор відрізків за точками початку та кінця
   Segment(const Point& start, const Point& end);
```

```
// Деструктор
   ~Segment();
// Селектори
   const Point& start() const; const Point& end() const;
// Обчислення довжини відрізка
   double length() const;
// Обчислення відстані від відрізка до точки
   double distance(const Point&) const;
};
```

Головна особливість класу відрізків випливає з його заголовка `class Triangle::Segment`. Він перестав бути самостійним класом, а тому не має сенсу поза класом трикутників. Цього статусу йому надає сам клас трикутників, приєднання заголовного файлу якого до заголовного файлу класу відрізків стає неодмінною умовою його компіляції. Він відрізняється від попереднього типами атрибутів і конструкторами, а також присвоєнням. Сталість атрибутів унеможливлює присвоєння, а унікальність сторін у трикутнику робить зайвим навіть звичайне копіювання.

Атрибути перетворилися на сталі відсилки, жорстко пов'язані з вершинами трикутника. Для забезпечення цілісності трикутника заборонено змінювати його вершини через сторони. Зверніть увагу на те, що в класі відрізків залишився лише один конструктор за точками початку та кінця відрізка – вершинами трикутника. Тепер не може бути конструктора за координатами, бо псевдоніми мають відсилати до наявних точок.

Лістинг 5.95

```
// Конструктор відрізків
Segment::Segment(const Point& a, const Point& b):
      _a(a), _b(b) {}   // Ініціалізація відсилок
```

Слід переглянути й саме визначення класу трикутників, переважно тому, що потрібно локалізувати клас відрізків.

Лістинг 5.96

```
// Клас трикутників із вкладеними відрізками:
// агрегація сторін за допомогою указників, агрегація їх вершин
// за допомогою відсилок
class Triangle {
public:
// Попереднє визначення вбудованого класу відрізків
   class Segment;
// Означаємо тип указника SidePtr на сторону трикутника
   typedef Segment* Triangle::* SidePtr;
// Конструктори трикутника:
// 1) за координатами його вершин
```

```
   Triangle(const double x1=0, const double y1=0, const double x2=1,
      const double y2=0, const double x3=0, const double y3=1);
// 2) за його вершинами
   Triangle(const Point& a, const Point& b, const Point& c);
// 3) копіювальний конструктор
   Triangle(const Triangle&);
   ~Triangle();   // Деструктор
// Скорочене присвоєння вершин (без сторін)
   Triangle& operator=(const Triangle&);
// Обчислення вершин
   const Point apexA() const; const Point apexB() const;
   const Point apexC() const;
// Селектори сторін
   const Segment& side_a() const; const Segment& side_b() const;
   const Segment& side_c() const;
// Селектори-модифікатори сторін
   Segment& side_a(); Segment& side_b(); Segment& side_c();
   double perimeter() const;   // Обчислення периметра
   double area() const;   // Обчислення площі
   ostream& show(ostream& os) const;   // Візуалізація трикутника
// Делегування: обчислення довжин сторін
   double length_a() const; double length_b() const;
   double length_c() const;
// Обчислення довжин висот
   double length_ha() const; double length_hb() const;
   double length_hc() const;
private:
   Point _a, _b, _c; // Вершини трикутника
   mutable Segment *_ab, *_bc, *_ca; // Указники на сторони трикутника
};
ostream& operator<<(ostream&, const Triangle&); // Виведення трикутника
```

Зверніть увагу на те, що модифікувати потрібно й реалізацію класу трикутників. Функції, що використовують відрізки як параметри чи результат, мають відповідно кваліфікувати їх тип.

Лістинг 5.97

```
// Обчислення та запам'ятовування сторони _bc
const Triangle::Segment& Triangle::side_a() const {
   if(_bc==0) _bc=new Segment(_b, _c);
   return *_bc;
}
```

Діаграму цього класу трикутників подано на рис. 5.8.

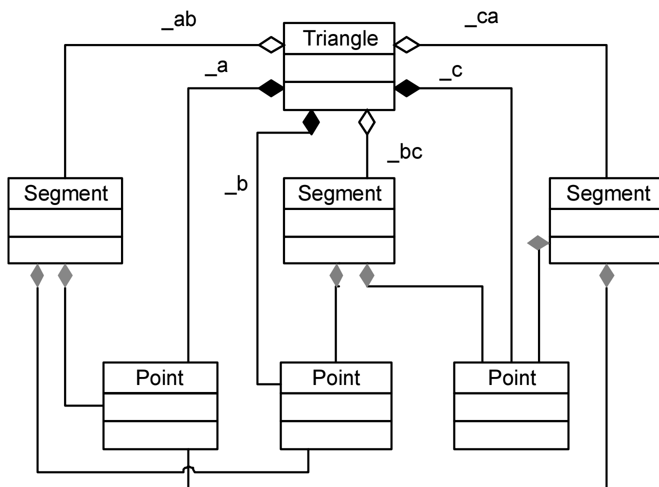


Рис. 5.8. Діаграма класу трикутників із вкладеними вершинами і агрегованими сторонами

Кожен об'єкт, укладений до трикутника, є в ньому не більше ніж в одному примірнику. Передусім це вершини трикутника, що виникають під час його створення як його компоненти. На рис. 5.8 цей тип укладень позначено чорним ромбом. Сторони відіграють роль необов'язкових атрибутів, приєднаних за допомогою указників. Їх буде створено на замовлення; сторони позначено стрілками з білими ромбами. Вершини – зовнішні щодо сторін атрибути; вони приєднані до них відсилками, а на рисунку їм відповідають з'єднання із сірими ромбами.

Вправа 5.39

Виконайте тестову програму з лістингу 5.72 та виконайте обліковування об'єктів для класу трикутників із вкладеним класом відрізків (лістинг 5.96). Результати порівняйте з результатами попередніх досліджень (вправи 5.27 , 5.29, 5.35 і 5.36).

Вправа 5.40

У табл. 5.2 наведено результати випробувань розглянутих версій класу трикутників за тестовою програмою з лістингу 5.72, виконаних у системі Microsoft Visual Studio 2008. Доповніть таблицю результатами випробувань, виконаних вами, і порівняйте їх.

Таблиця 5.2. Результати випробувань класу трикутників

Номер випробу-вання	Тип трикутника	Кількість вершин		Кількість сторін	
1	Композиція вершин	107		46	
2	Композиція вершин і сторін	39		12	
3	Композиція сторін	27		12	
4	Композиція вершин, агрегація сторін за допомогою указників	45		15	
5	Клас трикутників із вкладеними відрізками	15		15	

Підрахуйте також кількість викликів методів кожного з класів, виконану під час роботи тестової програми з лістингу 5.72, і для порівняння заповніть табл. 5.3.

Таблиця 5.3. Кількість викликів методів у класах

Номер випробу-вання	Тип трикутника	Клас		
		Point	Segment	Triangle
1	Композиція вершин			
2	Композиція вершин і сторін			
3	Композиція сторін			
4	Композиція вершин, агрегація сторін за допомогою указників			
5	Клас трикутників із вкладеними відрізками			

Результати розгляду видів відношень між об'єктами в цьому підрозділі свідчать, що правильний вибір агрегації, композиції чи асоціації може суттєво впливати на ефективність програми. Водночас розуміння суті кожного типу відношень дає змогу уникнути серйозних помилок у керуванні об'єктами, зокрема під час їх створення та видалення.

Запропонована класифікація вкладень дає змогу регламентувати відповідальність за створення та видалення об'єктів. Компоненти автоматично видаляє деструктор композита. Агреговані атрибути можуть бути приєднані за допомогою указників (тоді відповідальність за їх видалення несе деструктор агрегату) чи відсилок (у цьому разі агрегат використовує зовнішні відносно нього об'єкти, розміщенням і видаленням яких керує їх власник). Агрегуючи зовнішні об'єкти, потрібно бути впевненим у їх доступності впродовж усього терміну життя агрегату.

5.4. Примітивні контейнери

Узагальнивши композити, розглянуті в попередньому підрозділі, із трикутника на багатокутник, одержимо приклад особливо важливого композита, відомого під назвою *контейнер* (container). Справді, багатокутник можна розглядати як ємність, наповнену вершинами, для яких задано певний порядок обходу.

Отже, контейнери – це класи, об'єкти яких можуть містити інші вкладені в них об'єкти, зазвичай у довільній кількості. Розміщені в контейнері об'єкти в сукупності називають його *вмістом* (content), а кожен окремий об'єкт – *одиницею вмісту* (contained item або containee).

Насправді ми вже стикалися з контейнерами. Це передусім масиви та списки. Символьний рядок теж можна розглядати як впорядкований контейнер символів. Стандартна бібліотека C++ пропонує багато різних типів стандартних контейнерів, з якими варто навчитися працювати. Однак не менш важливо опанувати методи програмування, застосовувані для створення і обробки контейнерів. Тому, як і раніше, діє домовленість про позначення програмованих нами контейнерів іменами з великої літери. Так, ми будемо розрізняти класи `string` і `String`, `list` і `List` тощо.

Є різні способи доступу до вмісту контейнерів. Один із найпоширеніших – доступ до одиниці вмісту за її номером, як у масиві. Для реалізації цього способу використовують стандартну *операцію індексування* (subscript operator), яку вже було розглянуто під час вивчення масивів і рядків. Тепер визначимо операцію індексування для контейнерів довільного типу.

5.4.1. Клас масивів

Програмований масив із фіксованою розмірністю – найпростіший тип контейнера, який ми розглянемо. Як і раніше, тип елемента масиву задано (поки що) визначенням типу. Розмір `_size` масиву сталий, його не можна змінити після створення масиву. Указник `_allocator` до вмісту масиву також сталий, його не можна переспрямувати в інше місце.

```
typedef int Elem;                                          Лістинг 5.98
// Простий масив елементів типу Elem
class Array {
public:
   class BadArray; // Вбудований клас для обробки помилкових ситуацій
   explicit Array(size_t); // Конструктор без можливості конвертування типів
   ~Array();// Деструктор
   Elem& operator[](const size_t index); // Селектор-модифікатор елемента
   const Elem& operator[](const size_t index) const; // Селектор елемента
   size_t size() const; // Розмірність масиву (селектор)
private:
   const size_t _size; // Розмірність масиву
   Elem* const _allocator; // Уміст масиву
// Операції та функції, заборонені для використання в масивах
   Array(const Array&); Array& operator=(const Array&);
};
// Вбудований клас для обробки помилкових ситуацій
class Array::BadArray {
private:
   const string& _reason; // Причина, що зумовила аварійну ситуацію
   size_t _index; // Індекс елемента, під час доступу до якого
                  // виникла аварійна ситуація
// Заборонені функції й операції
   BadArray (const BadArray&); BadArray& operator=(const BadArray&);
public:
// Конструктор
   BadArray(string reason="", const size_t  index=0);
// Деструктор
   ~BadArray();
// Візуалізація діагностики
   ostream& diagnose(ostream&) const;
};
// Виведення діагностичного повідомлення
ostream& operator<<(ostream&, const Array::BadArray&);
```

Зауваження. *Зверніть увагу на те, що засоби копіювання контейнерів заборонено використовувати, бо вони містяться в закритій частині, а тому можуть залишатися нереалізованими. Такі операції зазвичай не передбачені для контейнерів. Закривати їх доводиться для того, щоб уникнути створення системою програмування заданих за замовчуванням версій.*

Перше запитання, яке виникає щодо масиву, пов'язане з типом вкладання вмісту до контейнера. Ним мала би бути композиція, але оскільки контейнер зазвичай використовують для розміщення довільної наперед

невідомої кількості одиниць вмісту, проте він також може виявитися порожнім, то застосуємо агрегацію за допомогою указника.

Для масиву агрегацію атрибута `_allocator` за допомогою указника застосовують як доступний спосіб визначення масиву, розміри якого невідомі до виконання. Реалізації методів для класу масивів фактично не відрізняються від реалізацій відповідних методів для рядків. Наведемо деякі з них.

Лістинг 5.99

```
// Конструктор без можливості конвертування типів
Array::Array(const size_t sz):
// Ініціалізація атрибутів
      _size(sz), _allocator(new Elem[_size]) {}
```

Створений таким способом масив наповнено сміттям. На цьому етапі вивчення його можна було б ініціалізувати нейтральним значенням, скажімо нулем, – наприклад, так:

```
for (size_t i=0; i<_size; ++i) Elem[_size]=0;
```

Така ініціалізація була б доречною в разі використання масиву для цілих або інших арифметичних типів – наприклад, якщо тип елемента визначено як

```
typedef double Elem;
```

У разі ж програмованих типів, скажімо

```
typedef Complex Elem;
```

така ініціалізація виявилася б повторною (поясніть, чому).

Детальніше цю проблему розглядатимемо далі, а тепер повернімося до реалізації інших методів класу масивів. Ось іще два з них:

Лістинг 5.100

```
// Деструктор
Array::~Array() {delete [] _allocator; _allocator=0; return;}
// Селектор-модифікатор елемента масиву
Elem& Array::operator[](const size_t index) {
   if(index>=_size) throw BadArray("Bad index: ", index);
   return _allocator[index];
}
```

Вправа 5.41

Реалізуйте решту методів класу масивів і класу обробки його аварійних ситуацій з лістингу 5.98. Складіть і виконайте кілька тестових програм для різних типів елементів масиву.

Вправа 5.42

Складіть програму, яка б видавала статистику звертань до кожного з елементів визначеного в ній масиву.

Чим клас програмованих масивів відрізняється від масиву на рівні мови C? Передусім зазначимо, що реалізація програмованого масиву сама ґрунтується на використанні масиву мови C, але приховує його від безпосереднього доступу. Двоступеневий доступ за допомогою методу програмованого масиву за ефективністю, звісно, поступається прямому доступу з використанням адресної арифметики. Але компенсацією за ці накладні витрати стає можливість обліковувати звертання до кожного з елементів масиву (див. вправу 5.42), а також контролювати, чи не вийшли індекси за допустимі межі.

Хибою масивів – як програмованих, так і стандартних – можна вважати вимогу задавати кількість їх елементів уже на момент створення. При цьому всі елементи вважають рівнозначними, навіть ті з них, що містять сміття чи були заповнені нейтральними елементами під час створення масиву. Щоб програмування стало надійним, потрібно більш дисципліновано працювати з контейнерами, зокрема дотримувати складніших вимог щодо їх наповнення та доступу до них. Масиви ж буде використано як базу для реалізації складніших контейнерів.

5.4.2. Клас обмежених послідовностей

Використаємо клас масивів для реалізації послідовного контейнера, який так і називатимемо – *послідовністю* (sequence). Певною мірою послідовність нагадує контейнерний клас `vector` зі стандартної бібліотеки, але, на нашу думку, назва «послідовність» краще віддзеркалює сутність цього класу.

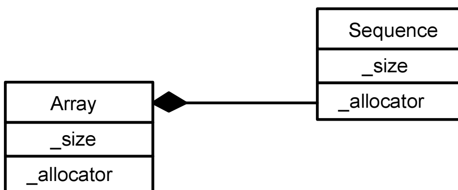


Рис. 5.9. Діаграма класу обмежених послідовностей

Діаграму класу послідовностей зображено на рис. 5.9.

Із рис. 5.9 видно, що послідовність `Sequence` містить два атрибути, перший з яких – `_allocator` – об'єкт класу масивів `Array`, що розподіляє пам'ять для послідовності, а другий – `_size` – задає розмір послідовності.

Визначення класу `Sequence` запишемо так:

```
// Обмежений послідовний контейнер елементів типу Elem      Лістинг 5.101
class Sequence {
public:
// Обробник помилкових ситуацій
   class BadSeq;
// Конструктор без можливості конвертування типів
   explicit Sequence(const size_t capacity=_default);
   ~Sequence();
   size_t capacity() const; // Видати ємність контейнера
   size_t size() const; // Видати поточний розмір
   bool empty() const; // Чи не порожня послідовність?
   bool full() const; // Чи не переповнений контейнер?
   Sequence& clear(); // Очистити послідовність
// Селектор елемента послідовності
   const Elem& operator[](const size_t index) const;
// Селектор-модифікатор елемента послідовності
   Elem& operator[](const size_t index);
   Sequence& add(const Elem& elem); // Долучити до послідовності
// Уставити всередину послідовності
   Sequence& insert(const Elem& elem, const size_t  index);
   Sequence& cut();// Укоротити послідовність на один елемент
   Sequence& remove(const size_t index); // Видалити елемент за його
номером
private:
   size_t _size; // Розмір послідовності
   static const size_t _default; // Стандартний розмір контейнера
   Array _allocator; // Масив для розміщення послідовності
// Службові функції, використовувані для реалізації відкритих функцій
   Sequence& doinsert(Elem elem, const size_t  index);
   Sequence& doremove(const size_t index);
// Заборонені функції й операції
   Sequence(const Sequence&); Sequence& operator=(const Sequence&);
};
ostream& operator<<(ostream&os, const Sequence& seq);
```

Послідовність відрізняється від масиву тим, що вона впродовж свого існування може збільшуватися чи зменшуватися. Справді, розмір масиву задано в момент його створення, після чого його вважають повністю заповненим, а всі елементи – однаково доступними для використання, навіть якщо їх не було ініціалізовано. Ця властивість не видається надто вдалою. Проаналізуємо з цього погляду клас послідовностей.

Є кілька варіантів визначення цього контейнера. Спочатку розглянемо обмежену послідовність на базі масиву. Як це видно із сигнатури конс-

труктора, під час створення послідовності буде задано її стандартну ємність _default. Початкову ємність контейнера можна перевищувати, але це досить витратна дія. Тому доводиться шукати компроміс між можливо зайвою пам'яттю та витратами на збільшення виділеної пам'яті в разі економного її замовлення.

Саму послідовність безпосередньо після її створення, поки до неї не додано жодного елемента, уважають порожньою. Слід розрізняти ємність і поточний розмір послідовності. Для цього використовують дві функції, одна з яких видає ємність, а друга – поточний розмір – відповідно capacity() та size(). Отже, якщо для створення послідовності s ємністю capacity було замовлено масив s._allocator розміром capacity, то після її створення функція s.size() поверне значення 0, а функції s.capacity() й s._allocator.size() у цей момент і надалі даватимуть значення capacity.

Статичний атрибут _default дає змогу задавати в конструкторі стандартне значення для ємності обмеженої послідовності. Його визначення, наприклад

```
const size_t Sequence::_default=100;
```

може міститись у файлі реалізації.

Сам конструктор делегує основну роботу конструктору масивів.

Лістинг 5.102

```
// Конструктор послідовності
Sequence::Sequence(const size_t capacity):
// Ініціалізація поточного розміру послідовності
        _size(0),
// Ініціалізація масиву за допомогою його конструктора
        _allocator(capacity) {}
```

Оскільки виділення та вивільнення пам'яті зосереджено в класі масивів, деструктор послідовностей тривіальний:

Лістинг 5.103

```
// Деструктор
Sequence::~Sequence(){}
```

Далі розглянемо методи, що стосуються розмірів послідовності. Метод capacity() визначення ємності послідовності делегує селекторові розміру масиву _allocator, який, зауважимо, не розрізняє ємність і розмір.

Лістинг 5.104

```
// Видати ємність контейнера
size_t Sequence::capacity() const {
// Делегування
   return _allocator.size();
}
```

Зауваження. *Може виникнути бажання доповнити послідовність іще одним атрибутом, у якому зберігатиметься її ємність. Попри очевидну економність такого рішення відкидаємо його як нестійке щодо можливих змін у концепції розміщення послідовності, – скажімо, використання масиву зі змінними розмірами.*

Реалізація методу обчислення розміру послідовності зводиться до виклику селектора атрибута `_size`. Слід виходити з того, що методи додання та видалення елементів послідовності оновлюють значення цього атрибута.

Лістинг 5.105

```
// Видати поточний розмір
size_t Sequence::size() const {return _size;}
```

Наступні дві функції перевіряють, чи не є послідовність порожньою або переповненою:

Лістинг 5.106

```
// Чи не порожня послідовність?
bool Sequence::empty() const {return (size()==0);}
// Чи не переповнений контейнер?
bool Sequence::full() const {return size()==capacity();}
```

Очищення ж послідовності методом `clear()` природно зводиться до обнуління цього самого атрибута:

Лістинг 5.107

```
// Очистити послідовність
Sequence& Sequence::clear() {_size=0; return *this;}
```

Модифікатор і селектор елемента послідовності визначено як операції індексування, які після певних перевірок делегують свої функції іншим операціям індексування, а саме методу доступу за індексом у розподілювачі пам'яті, у розглядуваному випадку – масиву. Зауважимо, до речі, що розподіляти пам'ять міг би й інший клас, який відповідав би використовуваному в послідовності інтерфейсу: мав би конструктор з одним параметром і функцію, що задає його розмір, а також операцію індексування.

Є два способи доступу до елемента послідовності: за допомогою селектора та селектора-модифікатора. Нагадаємо, що система програмування ставить у виклику перший із них тоді, коли йдеться про сталу послідовність, наприклад передану за допомогою параметра, котрий задекларовано як сталу відсилку (до речі, подумайте, чому немає сенсу від самого початку визначати послідовність як сталу).

Лістинг 5.108

```
// Селектор елемента послідовності
const Elem& Sequence::operator[](const size_t index) const {
// Перевірка, чи не порожня послідовність
```

```
   if(empty())
     throw BadSeq("Try to manipulate the empty sequence");
// Перевірка виходу за межі послідовності
   if(index>_size)
     throw BadSeq("Try to access a non existing element of the sequence");
// Делегування розподілювачу пам'яті
   return _allocator[index];
}
```

Селектор-модифікатор не відрізняється від селектора нічим, окрім своєї сигнатури. Його реалізацію наведено тут власне для підтвердження цього факту, а також для того, щоб ви спробували сконструювати осібні визначення селекторів і модифікаторів.

Лістинг 5.109

```
// Селектор-модифікатор елемента послідовності
Elem& Sequence::operator[](const size_t index) {
// Перевірка, чи не порожня послідовність
   if(empty())
     throw BadSeq("Try to manipulate the empty sequence");
// Перевірка виходу за межі послідовності
   if(index>_size)
     throw BadSeq("Try to access a non existing element of the sequence");
// Делегування розподілювачу пам'яті
   return _allocator[index];
}
```

Зауваження. *Операція індексування в масивах виникла в програмуванні тому, що на апаратному рівні вона доволі легко реалізується через операцію модифікації адреси виконуваної команди за допомогою вмісту індексного регістру. Ми поширили операцію індексування на програмовані типи даних. Реалізація її стала не такою простотою і ефективною, як на базі адресної арифметики, але з точки зору наступності системи позначень використання квадратних дужок з одночасним суміщенням функцій селектора і модифікатора видається досить слушним. Узагалі кажучи, суміщення кількох функцій (тут – перегляду й заміни члена послідовності) у межах однієї операції радше вважають вадою, ніж перевагою. Тому наявність оператора індексування в контейнерах – лише данина традиції. Така традиційність була б зрозумілішою, якби компілятор, розпізнаючи призначення виклику, завжди застосовував як селектор сталу операцію індексування. Однак, на жаль, це не просто. Ми ще повернемося до цієї проблеми в підрозділі 5.8 і запропонуємо спосіб розрізнення читання та запису, який технічно надто складний, щоб застосувати його тепер.*

Перейдемо до реалізації центральних методів додання та видалення елементів послідовності. Як видно з визначення класу, за ці дії відпові-

дають як відкриті, так і закриті функції. Це такий собі приклад інкапсуляції в дії. Відкрита функція задає точку входу – інтерфейс – користувача класу, виходячи з призначення якого визначається її сигнатура. Користувачеві недоступна реалізація інтерфейсної функції, а тому вона може спиратися на закриті від користувача класу функції, від яких залежать технічні особливості реалізації. Причин приховувати особливості реалізації може бути кілька, але найважливішою залишається можливість унесення змін до неї, не впливаючи на спосіб використання класу.

Певною мірою ми вже використовували цей прийом, реалізуючи конструктор класу дат у підрозділі 4.8. Тепер розглянемо його трохи докладніше.

На рис. 5.10 зображено масив `a` на `m` місць, у якому розміщено послідовність `s` із `n` елементів. Домовимося нумерувати члени послідовності, починаючи з першого номера.

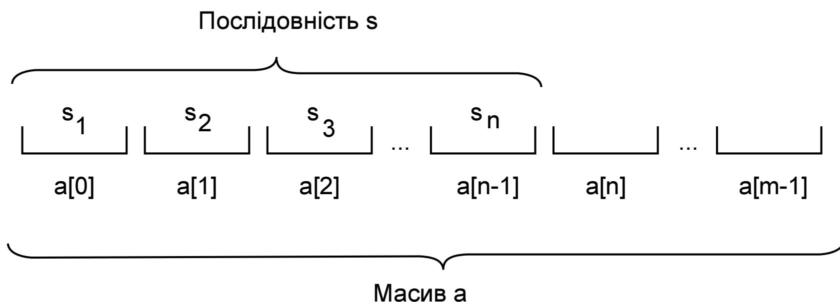


Рис. 5.10. Схема розміщення членів послідовності на базі масиву

У визначенні класу є два методи додання елемента до послідовності залежно від того, куди буде вставлено елемент: у її кінець чи в середину. У першому випадку новий член ставиться на вільне місце, не зачіпаючи інших, а в другому частина послідовності має зсунутися праворуч, щоб звільнити місце для нового елемента.

Лістинг 5.110

```
// Дописати до послідовності
Sequence& Sequence::add(const Elem& elem) {
   return doinsert(elem, _size); // Помістити в кінці послідовності
}
// Уставити всередину послідовності
Sequence& Sequence::insert(const Elem& elem, const size_t  index) {
// Перевірити допустимість індексу
   if(_size<index) throw BadSeq(
```

```
      "Try to insert after a non existing element of the sequence");
// Помістити елемент усередині послідовності
   return doinsert(elem, index);
}
```

Обидва методи реалізовано за допомогою спільного закритого методу. У ньому спочатку перевіряється, чи є вільне місце, і якщо воно є, то розмір послідовності збільшується на одиницю. Усі її члени, що містяться справа від нового елемента, зсуваються праворуч на одну позицію (у разі додання в кінець послідовності цей цикл порожній). Нарешті, новий елемент ставиться на звільнене місце.

Лістинг 5.111

```
// Додавання елемента до послідовності
Sequence& Sequence::doinsert(const Elem& elem, const size_t  index) {
   assert(index<=_size);
// Якщо контейнер переповнено
   if(_size+1>capacity())
      throw BadSeq("The bounded sequence is full, no more space");
// збільшити розмір послідовності
   ++_size;
// Зсунути залишок послідовності праворуч
   for (size_t i=_size-1; i>index; --i)
      _allocator[i]= _allocator[i-1];
// Уставити новий елемент у масив
   _allocator[index]=elem;
   return *this;
}
```

У разі видалення члена послідовності теж розрізняють два випадки: потрібно видалити або останній елемент (тоді його номер не потрібен), або той, що міститься всередині послідовності за його номером. Знову маємо дві інтерфейсні функції:

Лістинг 5.112

```
// Укоротити послідовність на один елемент
Sequence& Sequence::cut() {
   return doremove(_size); // Видалити елемент із кінця
}
// Видалити елемент за його номером
Sequence& Sequence::remove(const size_t index) {
// Перевірити допустимість індексу
   if((_size<index)||(index==0))
     throw BadSeq("Try to delete a non existing element of the sequence");
   return doremove(index); // Видалити елемент за індексом
}
```

І тут є спільний службовий метод для реалізації видалення елемента. Після перевірки непорожності послідовності всі елементи, що містяться справа від видаленого, зсуваються ліворуч. У разі вкорочення послідовності цей цикл знову стає порожнім.

Лістинг 5.113

```
// Видалити елемент за його номером
Sequence& Sequence::doremove(const size_t index) {
   assert(index<=_size);
// Перевірити непорожність послідовності
   if(empty())
      throw BadSeq("Try to manipulate the empty sequence");
// Зменшити розмір послідовності
   --_size;
// Зсунути залишок послідовності ліворуч
   for (size_t i=index; (i<_size)&&(i>0); ++i)
      _allocator[i-1]= _allocator[i];
   return *this;
}
```

Зверніть увагу на те, що в обох службових методах використано припущення про коректність номера члена послідовності; це гарантують їм інтерфейсні функції.

Вправа 5.43

Доповніть клас обмежених послідовностей з лістингу 5.101 методом доступу до останнього елемента за сигнатурою

```
const Elem& Sequence::last() const;
```

Вправа 5.44

Доповніть клас обмежених послідовностей з лістингу 5.101 методом видалення кількох елементів, задавши номер першого і останнього з них,

```
Sequence& Sequence::remove(const size_t size fist, const size_t last);
```

Проаналізуємо одержані результати. На відміну від масиву, який має довільний, але фіксований на момент створення розмір, розміри обмеженої послідовності змінюються в процесі її функціонування, хоча й не як завгодно, а лише в межах заданої під час створення ємності. Якщо, створюючи обмежену послідовність, задати надто велику ємність, то є ризик того, що значну частину пам'яті просто не буде використано. Коли ж виділено мало місця, то в разі його заповнення під час виконання виникне особлива ситуація, обробка якої покладається на користувача класу.

Використання обмежених контейнерів доцільне в ситуаціях, коли наперед відомий максимальний обсяг даних. Тоді можна почати роботу з резервування потрібного обсягу пам'яті у вигляді масиву. Далі цей масив буде використано для розміщення членів послідовності.

5.4.3. Клас необмежених послідовностей

Спробуємо змінити структуру послідовності, передбачивши в ній можливість заміни в разі потреби меншої ємності на більшу. Зображена на рис. 5.11 схема відрізняється від попередньої тим, що масив `_allocator` агрегується за допомогою указника.

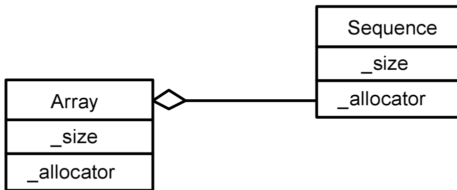


Рис. 5.11. Діаграма класу необмежених послідовностей

Відмінності від попереднього визначення послідовності незначні. Зверніть увагу на те, що зміни стосуються лише закритої частини класу. Це означає, що програми, що використовують обмежений контейнер, можна перекомпілювати під використання необмеженого контейнера, не вносячи жодних змін до їхніх текстів.

Лістинг 5.114

```
// Необмежений послідовний контейнер елементів типу Elem
class Sequence {
public:
   class BadSeq; // Обробник помилкових ситуацій
// Конструктор, закритий від конвертування типів
   explicit Sequence(const size_t capacity=_default);
   ~Sequence();// Деструктор
   size_t capacity() const; // Видати ємність контейнера
   size_t size() const; // Видати поточний розмір
   bool empty() const; // Чи не порожня послідовність?
   bool full() const; // Чи не переповнений контейнер?
   Sequence& clear();// Очистити послідовність
// Селектор елемента послідовності
   const Elem& operator[](const size_t index) const;
// Селектор-модифікатор елемента послідовності
   Elem& operator[](const size_t index);
   Sequence& add(const Elem& elem); // Дописати до послідовності
// Уставити всередину послідовності
   Sequence& insert(const Elem& elem, const size_t  index);
```

```
   Sequence& cut(); // Укоротити послідовність на один елемент
// Видалити елемент за його номером
   Sequence& remove(const size_t index);
private:
   size_t _size; // Розмір послідовності
   static const size_t _default; // Стандартний розмір блока контейнера
   Array* _allocator; // Масив для розміщення послідовності
   void enlarge(const size_t times=2); // Збільшити ємність
// Додати елемент до послідовності
   Sequence& doinsert(const Elem elem, const size_t  index);
// Видалити елемент за його номером
   Sequence& doremove(const size_t index);
// Невикористовувані функції й операції
   Sequence(const Sequence&); Sequence& operator=(const Sequence&);
};
ostream& operator<<(ostream&os, const Sequence& seq);
```

Модифікація закритої частини класу спричиняє зміни в реалізації методів. Вони стосуються передусім конструктора та деструктора. Тепер конструктор має явно замовити створення масиву:

Лістинг 5.115

```
// Конструктор, закритий від конвертування типів
Sequence::Sequence(const size_t size):
// Ініціалізація довжини порожньої послідовності
      _size(0),
// Виділення пам'яті для послідовності
      _allocator(new Array(size)) {}
```

Деструктор нині відповідає за видалення масиву:

Лістинг 5.116

```
// Деструктор
Sequence::~Sequence() {delete _allocator; return;}
```

Відповідно змінюється механізм делегування:

Лістинг 5.117

```
// Видати ємність контейнера
size_t Sequence::capacity() const {
// Делегування
   return _allocator->size();
}
```

Індексування також інакше звертається до елемента масиву. Наведемо лише текст селектора. Таку саму зміну потрібно внести й до селектора-модифікатора.

```
// Селектор елемента послідовності                    Лістинг 5.118
const Elem& Sequence::operator[](const size_t index) const {
// Перевірка порожності послідовності
   if(empty())
     throw BadSeq("Try to manipulate the empty sequence");
// Перевірка виходу за межі послідовності
   if(index>_size)
     throw BadSeq("Try to access a non existing element of the sequence");
// Делегування розподілювачу пам'яті за допомогою його указника
   return (*_allocator)[index];
}
```

Тут важливо розуміти, що індексування в масиві `_allocator[index]` вже потребує попереднього розіменування `(*_allocator)[index]`: адже `_allocator` – не ім'я, а указник на масив, у якому виконується індексування.

Найбільших змін зазнає функція додання нових елементів. По-перше, іншою стане реакція на переповнення масиву – його ємність буде збільшено. По-друге, доведеться змінити спосіб доступу до елемента масиву, оскільки його тепер іменовано указником `_allocator`.

```
// Додавання елемента до послідовності                Лістинг 5.119
Sequence& Sequence::doinsert(const Elem& elem, const size_t  index) {
// У разі потреби збільшити ємність
   if(_size+1>capacity()) enlarge(); ++_size;
// Зсунути залишок послідовності праворуч
   for(size_t i=_size-1; i>index; i--)
      (*_allocator)[i]=(*_allocator)[i-1];
// Уставити новий елемент у масив
   (*_allocator)[index]=elem;
   return *this;
}
```

З'являється нова функція `enlarge()`, завдання якої – переписати масив на нове місце, замовивши більший у `times` разів обсяг пам'яті:

```
 Збільшення контейнера
void Sequence::enlarge(const size_t times) {
// Дати запит на новий масив
   Array* newArray= new Array(times*_default+capacity()+1);
// Переписати до нього вміст старого масиву
   for (size_t i=0; i<_size; i++)
      (*newArray)[i]=(*_allocator)[i];
// Видалити старий масив
```

```
    delete _allocator;
// Переспрямувати розподільник пам'яті на новий масив
    _allocator=newArray;
    return;
}
```

Оскільки пам'ять виділяється блоками, що мають певний розмір, передбачено функцію змінення розміру блока.

Вправа 5.45

На якомусь етапі використання послідовності може виявитися, що її ємність доцільно зменшити. Реалізуйте функцію `reduce` зменшення ємності на задану кількість блоків, симетричну до `enlarge`, за сигнатурою

```
void reduce(const size_t times=1);
```

Завершуючи вдосконалення класу послідовностей, варто усвідомити суть процесу, який ми виконали. На зорі програмування вважали, що програма проходить два основні етапи свого життєвого циклу: створення й експлуатацію. Створенням програм займалися програмісти-розробники, а експлуатацію супроводжували програмісти супроводу. Тому зазвичай ці етапи називали створенням і супроводом. У процесі супроводу виявляли та виправляли помилки в програмі, а також удосконалювали її. Із часом центр ваги супроводу було перенесено з виправлення помилок до вдосконалень так суттєво, що процес супроводу програмної системи почали називати продовженням її розроблення. Досвідчені програмісти вважають, що програми не пишуть, а переписують з однієї версії до іншої. Тому, на думку розробників програмних систем, доопрацювання програмного коду – основний метод його розроблення, призначений для того, щоб поліпшити код і пристосувати його до нових умов використання. Тут ідеться не про виправлення помилок, а про процес, мета якого – удосконалити програми, щоб спростити подальшу підтримку та супровід коду. Сам процес доопрацювання одержав назву *рефакторингу коду* (code refactoring), під яким розуміють модифікацію коду програмної системи без змінення її поведінки.

Порівнюючи дві версії класу послідовностей, можна зазначити кілька можливих причин для рефакторингу. Перша – усунути залежність від способу розміщення масиву. Справді, якщо в реалізації функції `doinsert()` не використовувати напряму атрибут `_allocator`, а користуватись операцією індексування, то цей фрагмент коду не доведеться змінювати в разі переходу до необмеженої послідовності:

```
// Додавання елемента до послідовності.                    Лістинг 5.121
// Перший рефакторинг: модифікація оператора індексування
Sequence& Sequence::doinsert(const Elem& elem, const size_t index) {
   assert(index<=_size);
// Якщо контейнер переповнено
   if(_size+1>capacity())
      throw BadSeq("The bounded sequence is full, no more space");
   ++_size;   // Збільшити розмір
// Зсунути залишок послідовності праворуч
   for (size_t i=_size-1; i>index; --i)
      (*this)[i]=(*this)[i-1];
   (*this)[index]=elem;    // Уставити новий елемент у масив
   return *this;
}
```

Наведений прийом ілюструє так званий *принцип першості інтерфейсу*, згідно з яким слід програмувати, спираючись на інтерфейс, а не на реалізацію. Пояснимо застосування цього принципу на щойно наведеному коді з лістингу 5.121, порівнюючи його з попередньою реалізацією функції `doinsert()` із лістингу 5.111. У цьому коді явно використано реалізацію класу, оскільки є пряме звернення до його атрибута `_allocator`. Справді, `_allocator[i]` – це виклик оператора індексування масиву, застосованого до закритого атрибута послідовності.

Чому не варто спиратися на реалізацію? Тому що зміна, яку внесено до реалізації (у нашому випадку – заміна типу атрибуту `_allocator`), зумовлює потребу в модифікації всіх частин коду класу, де використано цей атрибут. Насправді не варто було напряму звертатися до реалізації, оскільки клас послідовностей сам містить інтерфейсну функцію індексування. Читаючи код лістингу 5.121, важливо зрозуміти, що `(*this) [i]` – виклик власного методу класу послідовностей, застосованого до поточного об'єкта. Запис `(*this) [i]` можна було б подати і як `this->operator[](i)` чи скорочено `operator[](i)` (поясніть, чому). У разі виклику методу індексування послідовності, як бачимо з його реалізації в лістингу 5.118, буде використано атрибут `_allocator` і відповідно реалізацію класу саме тому, що цей метод служить реалізацією доступу до елемента послідовності. Усім іншим методам краще звертатися до них за допомогою інтерфейсу.

Принцип першості інтерфейсу перед реалізацією відіграє чи не найважливішу роль у сучасному програмуванні. Невдалі, не кажучи вже про некоректні, інтерфейси можуть звести нанівець усю розробку.

Зверніть увагу на те, що в реалізаціях функцій перевірки порожності чи переповнення послідовності в лістингу 5.106 також використано інтерфейсні функції доступу без прямого звернення до атрибутів `_size` і `_capacity`. На перший погляд здається, що пряме порівняння атрибута `_size` із нулем

```
_size == 0;
```

виконується швидше, ніж виклик інтерфейсної функції з подальшим порівнянням

```
size() == 0;
```

а тому безпосереднє використання реалізації, а не наявного інтерфейсу часто обґрунтовують вимогами ефективності. Таке обґрунтування може виявитися безпідставним.

По-перше, застосування реалізації замість інтерфейсу неодмінно призводить до зниження якості розробки через ускладнення супроводу та подальших рефакторингів. По-друге, застосування інтерфейсу може виявитися не менш ефективним, аніж пряме звертання до реалізації, якщо використовувати вбудовані (inline) функції.

Вправа 5.46

Розробіть програмний проект, у якому методи

Лістинг 5.122

```
size_t capacity() const; // Видати ємність послідовності
size_t size() const;     // Видати поточний розмір
bool empty() const;      // Чи не порожня послідовність?
bool full() const;       // Чи не переповнена послідовність?
Sequence& clear();       // Очистити послідовність
```

реалізовано відкритим способом за допомогою вбудованих функцій.

До службових методів доцільно додати метод перевірки переповнення масиву:

Лістинг 5.123

```
// Перевірка переповнення контейнера
void Sequence::check() {
   if(_size+1>capacity())
// Контейнер переповнено
      throw BadSeq("The bounded sequence is full, no more space");
}
```

після чого метод вставлення нового елемента всередину послідовності перестане залежати від способу її реалізації.

Остаточно результат рефакторингу функції можна подати так:

Лістинг 5.124

```
// Додавання елемента до послідовності.
// Другий рефакторинг: перевірка переповнення контейнера
Sequence& Sequence::doinsert(const Elem& elem, const size_t  index) {
   assert(index<=_size);
   check();// Перевірка переповнення контейнера
   ++_size; // Збільшити розмір
// Зсунути залишок послідовності праворуч
   for (size_t i =_size-1; i>index; i--)
      operator[](i)=operator[](i-1);
   operator[](index)=elem;  // Уставити новий елемент у масив
   return *this;
}
```

Вправа 5.47

Завершіть реалізацію необмеженої послідовності, відслідковуючи зміни внесені до реалізації методів обмеженої послідовності.

Вправа 5.48

Доповніть клас необмежених послідовностей засобами звільнення пам'яті, що перестала використовуватися після видалення певної кількості членів послідовності.

Проблему обмеженості послідовності розв'язано, утім залишилася ще проблема неефективності вставлення та видалення елементів усередині послідовності. Перш ніж перейти до контейнерів, які вирішують і цю проблему (наприклад, списків), розглянемо ще кілька важливих контейнерів, у яких жорстко регламентовано способи додання та вилучення елементів.

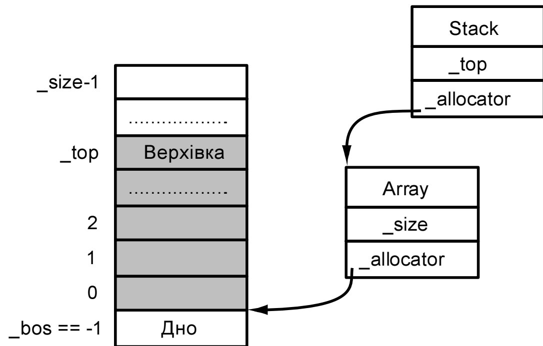


Рис. 5.12. Діаграма обмеженого стеку на базі масиву

5.4.4. Клас стеків на базі масиву

Почнемо з класу стеків на базі масиву. Не будемо концентрувати увагу на питанні обмеженості чи необмеженості стека, уважаючи, що читач самостійно доповнить клас визначенням відповідної функції збільшення контейнера.

Діаграму стека наведено на рис. 5.12.

Визначення класу наведено в лістингу 5.125. Стек розміщено в масиві `_allocator` від його початку (нульовий індекс) до вершини `_top`. Для зручності програмування стек обмежено дном, значення якого задає статична стала `_bos`. Доречно нагадати, що іменованим сталим ми надаємо перевагу перед константами препроцесора. До того ж надання сталій статусу статичного атрибута розв'язує проблему її локалізації.

Лістинг 5.125

```
// Стек на базі масиву
class Stack {
public:
   class BadStack; // Обробник помилкових ситуацій
// Конструктор, закритий від конвертування типів
   explicit Stack(const size_t size=_default);
   ~Stack(); // Деструктор
   bool empty() const; // Чи порожній стек?
   bool full() const; // Чи не переповнений контейнер?
   size_t size() const; // Видати поточний розмір
   size_t capacity() const; // Видати ємність контейнера
// Керування стеком:
   const Elem& top() const; // Видати вершину стека
   void pop(); // Виштовхнути вершину стека
   void push(const Elem& value); // Помістити до стека
private:
   static const size_t _default; // Стандартний розмір блока контейнера
   static const size_t _bos; // Дно стека: Bottom of the Stack
   size_t _top; // Вершина стека
   Array* _allocator; // Масив для розміщення стека
   void enlarge(const size_t times=2); // Збільшити ємність
// Невикористовувані функції й операції
   Stack(const& Stack); Stack& operator=(const Stack&);
};
```

Прокоментуємо визначення класу. Почнемо з призначення атрибутів. Статичні атрибути `_bos` і `_default` задають відповідно дно стека та його стандартну ємність:

```
const size_t Stack::_bos=-1;
const size_t Stack::_default=100;
```
Лістинг 5.126

Масив _allocator призначено для розміщення елементів стека, а атрибут _top фіксує індекс його вершини. Дисципліна вставлення та видалення елементів стека робить зайвим окремий атрибут, відповідальний за його розмір.

Конструктор копіювання та копіювальне присвоєння закриті зі зрозумілих причин (поясніть, яких). Клас обробки помилкових ситуацій зазвичай визначають окремо. Рекомендуємо читачеві зробити це самостійно. Конструктор стека зрозумілий. Вершина набуває значення дна стеку, що свідчить про порожність стека. Розподілювач пам'яті замовляє новий масив:

Лістинг 5.127
```
// Конструктор, закритий від конвертування типів
Stack::Stack(const size_t size):
      _top(_bos),   // Ініціалізація вершини порожнього стека
      _allocator(new Array(size)) {}  // Виділення пам'яті для стека
```

Деструктор вивільняє виділену пам'ять:

Лістинг 5.128
```
// Деструктор
Stack::~Stack() {delete _allocator; return;}
```

Порожність або переповнення стека виявляють, виходячи зі значення вершини:

Лістинг 5.129
```
// Чи не порожній стек?
bool Stack::empty() const {return _top==_bos;}
// Чи не переповнений контейнер?
bool Stack::full() const {return _top==capacity()-1;}
```

Визначення ємності стека зазвичай делегують його масиву:

Лістинг 5.130
```
// Видати ємність контейнера
size_t Stack::capacity() const {
   return _allocator->size();   // Делегування
}
```

Розмір стека обчислюють, виходячи із значення його вершини:

Лістинг 5.131
```
// Видати поточний розмір
size_t Stack::size() const {return _top+1;}
```

Стек – це особливий тип контейнера, у якому безпосередньо доступний лише один його елемент, що міститься у верхівці. Уміст вершини непорожнього стека повертає стала функція її перегляду `top()`. Зверніть увагу: результат функції не скопійовано, він залишається на своєму місці, захищений від змін.

Лістинг 5.132

```
// Видати вершину стека
const Elem& Stack::top() const {
   assert(!empty()); // Функція не застосовна до порожнього стека
   return (*_allocator)[_top]; // Операція індексування в масиві
}
```

Варто також домовитися про статус аварійної ситуації, що виникає в разі спроби звернутися до порожнього стека. Ця ситуація подібна до ділення на нуль або видобування кореня з від'ємного числа. Вважатимемо, що відповідальність за застосовність функції несе клієнт класу, що її викликав. Це свого роду *принцип захисного програмування функцій*, згідно з яким функції застосовні лише за виконання певних умов (непорожній стек, ненульовий дільник, невід'ємний підкореневий вираз тощо). Істинність передумови гарантує клієнт класу, тому функція не виконує додаткової (власне кажучи, зайвої) перевірки. Розробник класу гарантує коректність результату функції за умови її застосовності. Однак на етапі відлагодження розробником доцільно помістити в код функції макрос перевірки суджень для виявлення можливого неправильного її використання клієнтом.

Наповнення стека можна змінити за допомогою двох функції, які зазвичай називають *проштовхуванням* (push) до стека та *виштовхуванням* (pop) із нього. Переглядаючи код функції, можна помітити, що насправді ніякого виштовхування не відбувається, просто вершину пересувають на одну позицію донизу, що зробить попередній елемент надалі недоступним. Зауважимо, що функція виштовхування не застосовна до порожнього стека.

Лістинг 5.133

```
// Виштовхування вмісту вершини стека
void Stack::pop() {
   assert(!empty()); // Функція не застосовна до порожнього стека
   _top--; // Просування вершини вниз
}
```

Зауваження. *Застерігаємо від привабливого на перший погляд, але по суті помилкового рішення щодо суміщення в одному методі функцій доступу до вершини з її видаленням. У деяких публікаціях саме так розуміють функцію* `top()`. *Проаналізуємо цей підхід, але дамо методу іншу назву* `top_pop()`.

Перша вада наведеного далі методу полягає в тому, що він завжди виконує дві дії, навіть коли одна з них зайва.

Лістинг 5.134

```
// Приклад поганого проектування:
// один метод виконує дві функції
const Elem& Stack::top_pop() {
   assert(!empty()); // Функція не застосовна до порожнього стека
// Виштовхування та просування вершини вниз
   return (*_allocator)[_top--];
}
```

Поділ показу та виштовхування вершини стека на дві окремі функції ілюструє важливий принцип розробки програмних систем, який назвемо *принципом звуження методів*: кожен метод має виконувати одну й лише одну функцію. Є багато аргументів на користь принципу звуження методів. До них належать спрощення налагодження та супроводу, розширення сфери застосування, але чи не найважливіший полягає в усуненні проблеми обробки помилкових ситуацій, які в разі недотримання зазначеного принципу можуть виникнути саме в разі завершення однієї функції та неповного завершення іншої.

Покласти елемент у вершину стека просто:

Лістинг 5.135

```
void Stack::push(const Elem& value) {
   if(full()) enlarge();
   (*_allocator)[++_top]=value;
}
```

Вправа 5.49

Реалізуйте функцію збільшення ємності контейнера.

Вправа 5.50

Реалізуйте операцію виведення вмісту стека. Проаналізуйте, якими допоміжними структурами даних потрібно скористатися.

5.4.5. Клас черг на основі масиву

Як відомо, стек працює за принципом LIFO: «останнім прийшов, першим пішов» («last in, first out»). У черзі інша дисципліна обслуговування – за принципом FIFO: «першим прийшов, першим пішов» («first in, first out»). Розглянемо клас черг на базі масиву.

Лістинг 5.136

```
// Черга на базі масиву
class Queue {
```

```
public:
   class BadQueue;           // Обробник помилкових ситуацій
// Конструктор, закритий від конвертування типів
   explicit Queue(const size_t size=_default);
   ~Queue();                 // Деструктор
   bool empty() const;       // Чи не порожня черга?
   bool full() const;        // Чи не переповнений контейнер?
   size_t capacity() const;  // Видати ємність контейнера
   size_t size() const;      // Видати поточний розмір
   const Elem& front() const; // Керування чергою: видати початок черги
   void get();                  // Забрати з черги
   void put(const Elem& value);  // Поставити в чергу
private:
   static const size_t _default; // Стандартний розмір блока контейнера
   static const size_t _boq;     // Дно черги: Bottom Of the Queue
   size_t _size;                 // Довжина черги
   size_t _front;                // Початок черги в масиві
   size_t _back;                 // Кінець черги в масиві
   Array* _allocator;            // Масив для розміщення стека
// Службові функції:
// Збільшення індексу циклічного масиву на одиницю
   size_t plus1(const size_t x);
   void enlarge(const size_t times=2); // Збільшення ємності
// Невикористовувані функції й операції
   Queue(const& Queue); Queue& operator=(const Queue&);
};
```

Прокоментуємо визначення класу. Почнемо з призначення атрибутів. Зміст статичних атрибутів залишається таким самим, як у стеку. Статичні атрибути `_boq` і `_default` задають відповідно дно черги та її стандартну ємність:

Лістинг 5.137

```
const size_t Queue::_boq=-1;
const size_t Queue::_default=100;
```

Власні атрибути мають таке призначення: `_front` – індекс початку черги, `_back` – індекс її кінця, `_size` задає довжину черги.

Конструктор містить ініціалізацію атрибутів:

Лістинг 5.138

```
// Конструктор, закритий від конвертування типів
Queue::Queue(const size_t size):
// Ініціалізація розміру порожньої черги
      _size(0),
```

```
// У порожній черзі кінець міститься перед початком
      _front(0), _back(_boq),
// Виділення пам'яті для черги
      _allocator(new Array(size)) {}
```

Тривіальний деструктор вивільняє виділену пам'ять:

Лістинг 5.139

```
// Деструктор
Queue::~Queue() {delete _allocator;return;}
```

Про наповненість або порожність черги свідчить значення атрибута її розміру; переповнену чергу виявляємо, порівнюючи ємність із розміром:

Лістинг 5.140

```
// Чи не порожня черга?
bool Queue::empty() const {return _size==0;}
// Чи не переповнений контейнер?
bool Queue::full() const {return _size==capacity();}
```

Визначення ємності, як завжди, делеговано масиву:

Лістинг 5.141

```
// Видати ємність контейнера
size_t Queue::capacity() const {
   return _allocator->size();   // Делегування
}
```

Визначення ж розміру черги реалізовано простим селектором:

Лістинг 5.142

```
// Видати поточний розмір
size_t Stack::size() const {return _size;}
```

Як і в разі стека, перегляд початку черги не впливає на її вміст:

Лістинг 5.143

```
// Видати початок черги
const Elem& Queue::front() const {
// До порожньої черги функція не застосовна
   assert(!empty());
   return (*_allocator) [_front];
}
```

Перш ніж перейти до функцій додання до черги та видалення елементів з неї, розглянемо концепцію циклічного масиву, яка дає змогу гнучко й ефективно розміщувати елементи обмеженої черги. Її реалізує закритий метод визначення наступної позиції в масиві `size_t plus1(const size_t)`, який проілюстровано на рис. 5.13. Звичайний порядок слідування

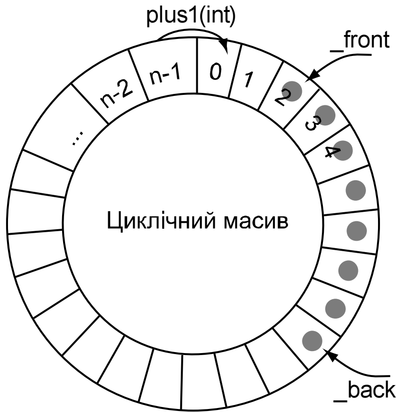


Рис. 5.13. Закритий метод визначення наступної позиції в масиві

індексів доповнено циклічним замиканням масиву, відповідно до якого наступним за останнім елементом масиву буде перший його елемент.

Якщо для розміщення черги скористатися звичайним масивом, як для стека, то після видалення елемента з черги з'явиться додаткова вільна ділянка на початку масиву. Зайнявши всю вільну пам'ять у кінці масиву, доцільно було б використати вільне місце на його початку. Цього можна досягти, або перемістивши всю чергу вздовж масиву (що неефективно), або замкнувши масив у кільце, як це зроблено в наведеній нижче реалізації.

Збільшення індексу масиву на одиницю тепер виконуватиме вже згадана функція:

Лістинг 5.144

```
// Збільшення індексу циклічного масиву на одиницю
size_t Queue::plus1(const size_t i) {
   return i<capacity()-1? i+1 :0;
}
```

Розглянемо тепер методи керування чергою. Для переміщення індексу в методі вилучення елемента з початку черги `get()` використано циклічність масиву. Елемент не видаляють із черги, лише відповідно переміщують індекс її початку:

Лістинг 5.145

```
// Вилучення з черги
void Queue::get() {
   assert(!empty()); // Функція не застосовна до порожньої черги
   _size--;
   _front=plus1(_front); // Вилучення полягає в збільшенні індексу
}
```

Циклічність використано й в операції додавання в кінець черги `put()`:

Лістинг 5.146

```
// Постановка в чергу
void Queue::put(const Elem& value) {
   if(full()) enlarge();
```

```
   _size++;
   _back=plus1(_back); // Збільшити індекс
   (*_allocator)[_back]=value; // Поставити новий елемент на вільне місце
}
```

Вправа 5.51

Завершіть реалізацію класу черг, доповніть його оператором виведення та виконайте тестування. Обґрунтуйте можливість виведення без застосування допоміжних структур даних.

5.4.6. Клас двобічних черг на базі масиву

Останній із контейнерів цього підрозділу – клас двобічних черг Deque (Double Ended Queue) – наводимо без коментарів.

Лістинг 5.147

```
// Двобічна черга на базі масиву
class Deque {
public:
   class BadDeque; // Обробник помилкових ситуацій
// Конструктор, закритий від конвертування типів
   explicit Deque (const size_t size=_default);
   ~Deque ();// Деструктор
   bool empty() const; // Чи не порожня черга?
   bool full() const; // Чи не переповнений контейнер?
   size_t capacity() const; // Видати ємність контейнера
   size_t size() const; // Видати поточний розмір
   const Elem& front() const; // Керування чергою: видати початок черги
   const Elem& back() const; // видати кінець черги
   void get(); // вилучити з початку черги
   void getBack();// вилучити з кінця черги
   void put(const Elem& value); // поставити в кінець черги
   void putFront(const Elem& value); // поставити в початок черги
private:
   static const size_t _default; // Стандартний розмір блока контейнера
   static const size_t _bod; // Дно черги: Bottom Of the Deque
   size_t _size; // Довжина черги
   size_t _front; // Початок черги в масиві
   size_t _back; // Кінець черги в масиві
   Array* _allocator; // Масив для розміщення стека
// Службові функції:
// Збільшення індексу циклічного масиву на одиницю
   size_t plus1(size_t);
// Зменшення індексу циклічного масиву на одиницю
   size_t minus1(size_t);
```

```
   void enlarge(const size_t times=2); // Збільшити ємність
// Невикористовувані функції й операції
   Deque (const& Deque); Deque& operator=(const Deque&);
};
```

Вправа 5.52

Реалізуйте клас двобічних черг на базі масиву, доповніть його оператором виведення та виконайте тестування.

5.5. Інтелектуальні указники

Вживання указників належить до сильних сторін C++, успадкованих від мови програмування C. Їх сила полягає у наданні розробнику прямого впливу на процеси розподілу пам'яті в програмі. На думку Д. Кнута, спосіб оперування указниками в мові C був блискучою інновацією, він вирішив багато проблем, які до цього були в структуруванні даних, і забезпечив гарний вигляд програмам на майбутнє. Разом з тим застосування указників створює певні проблеми, які можуть істотно вплинути на надійність програми. Таких проблем кілька: по-перше, це спроба використати неініціалізований указник; по-друге – поява завислих указників після некоректного звільнення пам'яті; по-третє – неможливість відрізнити указник від масиву. Тому спробуємо серйозніше проаналізувати указники.

Ось приклад некоректного використання неініціалізованого указника:

Лістинг 5.148

```
// Неініціалізований указник
Point* p; cout<<*p;
```

Невизначеним було названо указник, ініціалізований нулем. Ним не можна скористатися для розіменування, але невизначеність указника, на відміну від неініціалізованості, легко перевірити:

Лістинг 5.149

```
// Невизначений указник
Point* p=0; if(p==0) cout<<"Undefined pointer";
```

Компілятор може допомогти у виявленні неініціалізованих указників та інших неініціалізованих змінних таким попередженням:

```
warning C4700: local variable 'p' used without having been initialized
```

Використання указників, які показують на область пам'яті, якої немає, узагалі не буде діагностовано. Ось приклад:

Лістинг 5.150

```
// Завислий указник
Time* p=new Time(0, 0, 1); // Рис. 5.14
```

```
Time* p1=p; // Рис. 5.15
delete p1; p1=0; cout<<*p; // Рис. 5.16
```

Користуючись указником p, виділяємо область пам'яті, де розміщуємо екземпляр класу Time (рис. 5.14).

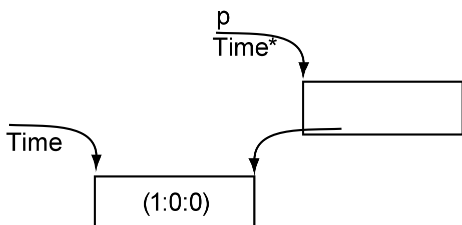


Рис. 5.14. Ініціалізація указника

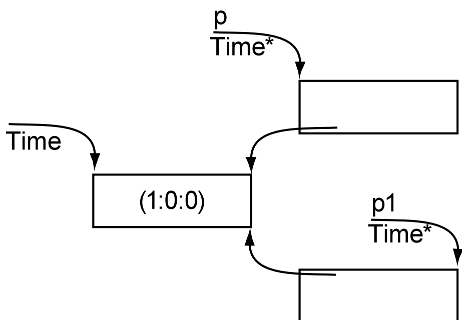


Рис. 5.15. Спрямування указника p1 на значення указника p

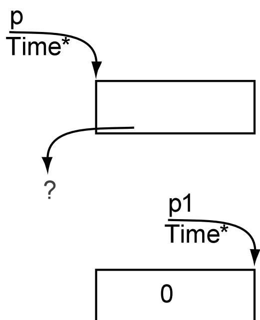


Рис. 5.16. Нейтралізований указник p1 і завислий указник p

Спрямовуємо туди інший указник p1 (рис. 5.15).

Користуючись щойно встановленим указником, звільняємо виділену пам'ять і нейтралізуємо указник p1. Указник p після цього зависає, хоча його значення не змінилося (рис. 5.16).

Так само не діагностовна спроба використати указник як масив, коли не було виділено пам'ять для масиву.

Лістинг 5.151

```
// Звертання до
// масиву, якого не існує
Time* p=new Time;
(p+5)->x()=1; p[10].y()=2;
```

Подолати ці вади мають *інтелектуальні указники* (smart pointer). Вони не просто вказують на об'єкт, але й контролюють його життєвий цикл.

5.5.1. Клас монолітних маніпуляторів

Інтелектуальний указник визначають як окремий клас. Такі класи, екземпляри яких відсилають до об'єктів інших класів, зазвичай називають *маніпуляторами* (handler). Розглянемо, наприклад, маніпулятори об'єктів якогось типу Pointee, який визначено окре-

мо. Маніпулятор поводиться так, як поводився б указник. Його можна розіменувати, діставшись до об'єкта маніпулювання, чи застосувати до нього операцію «стрілка» – доступу до члена класу за указником.

Почнемо з маніпуляторів, які ми назвемо *монолітними*.

Лістинг 5.152

```
// Тип для маніпулювання
typedef Time Pointee;
// Монолітний маніпулятор об'єктами Pointee. Повне копіювання
// об'єкта маніпулювання
class SmartPtr {
private:
   Pointee* _pointee; // Об'єкт маніпулювання
public:
   class BadPtr; // Обробник помилкових ситуацій
// Конструктор, закритий від конвертування типів
   explicit SmartPtr(Pointee* p=0);
   SmartPtr(const SmartPtr&); // Копіювальний конструктор
   ~SmartPtr();// Деструктор
   SmartPtr& operator=(const SmartPtr&); // Копіювальне присвоєння
   bool defined() const; // Перевірка визначеності маніпулятора
// Специфічні операції над указниками
   Pointee& operator*() const; // Операція розіменування маніпулятора
   Pointee* operator->() const; // Операція доступу за указником
};
// Виведення об'єкта маніпулювання
ostream& operator<<(ostream&, const SmartPtr&);
```

Маніпульований тип поки що задано просто оператором визначення типів `typedef`. Пізніше будуть розглянуті параметризовані маніпулятори.

Як завжди, до основного класу додано клас обробки аварійних ситуацій, деталі якого випустимо.

Передбачено також оператор виведення, який виводить значення об'єкта маніпулювання. Зверніть увагу, що в наведеній далі реалізації використано не стандартну операцію розіменування, а програмовану операцію маніпулятора. У разі невизначеного маніпулятора не буде виведено нічого.

Лістинг 5.153

```
// Виведення об'єкта маніпулювання
ostream& operator<<(ostream& os, const SmartPtr& sptr) {
   if(sptr.defined()) os<<*sptr;
}
```

Проаналізуємо склад інтелектуального указника. Його атрибут `_pointee` – це звичайний указник, за допомогою якого, власне, відбувається непрямий доступ до значення.

Реалізація перевірки визначеності маніпулятора фактично зводиться до перевірки значення `_pointee` на рівність нулю.

```
// Перевірка визначеності маніпулятора                    Лістинг 5.154
bool SmartPtr::defined() const {return _pointee!=0;}
```

Маніпулятори не виникають просто так унаслідок неявних перетворень типів, тому конструктор має кваліфікатор **`explicit`** і відповідно дозволяє лише явні виклики. Розглянемо реалізацію конструктора:

```
// Конструктор, закритий від конвертування типів          Лістинг 5.155
SmartPtr::SmartPtr(Pointee* p):
   _pointee(p) {}   // Ініціалізація маніпулятора указником
```

Розглянемо поведінку маніпулятора під час його створення, коли немає об'єкта маніпулювання. Якщо конструктор викликано без параметра, наприклад так:

```
SmartPtr pTime0;
```

то інтелектуальний указник стане невизначеним, а його атрибут `_pointee` гарантовано набуде значення 0. Важливо, що при цьому він не може виявитися неініціалізованим, що було однією з проблем у визначенні звичайних указників. Порівняйте з визначенням звичайного неініціалізованого указника (рис. 5.17).

```
Time* pt;
```

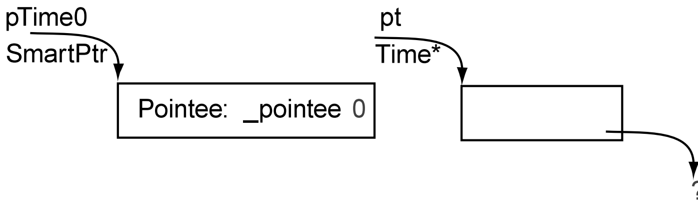


Рис. 5.17. Невизначені маніпулятор і указник

Із цього погляду використовувати інтелектуальний указник безпечніше.

Зауваження. *Зрозуміло, що немає захисту від застосування конструктора до параметра, що являє собою неініціалізований указник, тому варто бути обережними, наприклад не використовувати указники, а передавати як фактичний параметр адресу самого об'єкта, а не указник на нього.*

Обережним можна вважати, наприклад, такий спосіб створення інтелектуального указника. Маніпульований об'єкт створюємо безпосередньо в конструкторі маніпулятора. Розглянемо типовий прийом – за допомогою команди **new** (рис. 5.18):

```
SmartPtr pTime1(new Time(45, 23, 1));
```

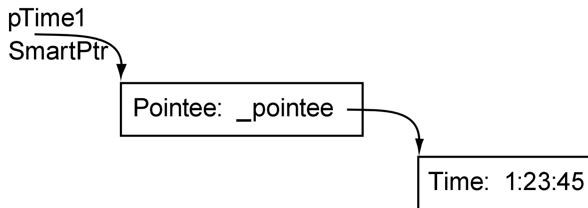


Рис. 5.18. Ініціалізація маніпулятора новоствореним об'єктом

Розглянемо копіювальний конструктор:

Лістинг 5.156

```
// Копіювальний конструктор
SmartPtr::SmartPtr(const SmartPtr& source):
// Якщо маніпулятор оригіналу визначено, то копіюємо оригінал
   _pointee(source._pointee? new Pointee(*source._pointee): 0) {}
```

Проаналізуємо поведінку копіювального конструктора. Під час створення маніпулятора за допомогою копіювального конструктора

```
SmartPtr pTime2(pTime1);
```

копіюється й сам маніпульований об'єкт (рис. 5.19).

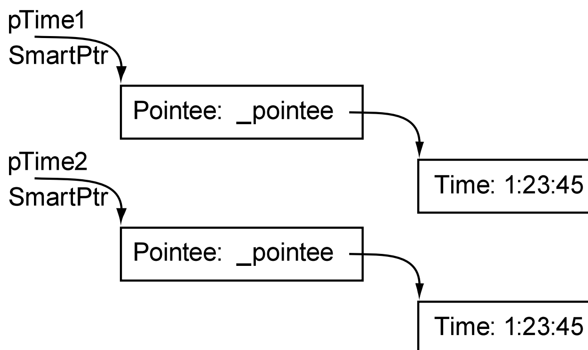


Рис. 5.19. Застосування копіювального конструктора

Тепер займемося деструктором. Видалення об'єкта маніпулювання за допомогою деструктора маніпулятора безпечніше, ніж його видалення за указником. Маніпулятор точно знає, що він має справу з об'єктом маніпулювання, а не з масивом. Тому немає ризику переплутати **delete []_pointee** з **delete _pointee**.

Лістинг 5.157

```
// Деструктор
SmartPtr::~SmartPtr() {delete _pointee; return;}
```

Зверніть увагу на те, що деструктор не перевіряє, чи визначено указник. Це коректно, бо стандарт С++ дозволяє застосувати видалення до невизначеного (нульового) указника.

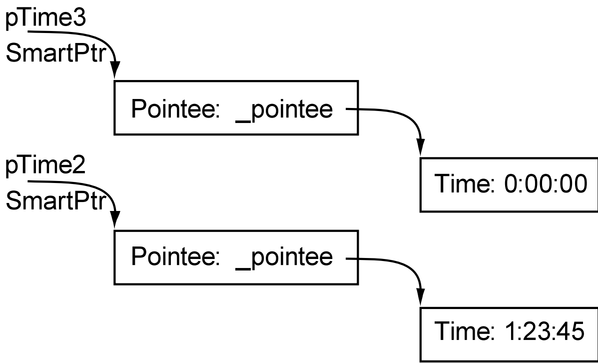


Рис. 5.20. Незалежні маніпулятори

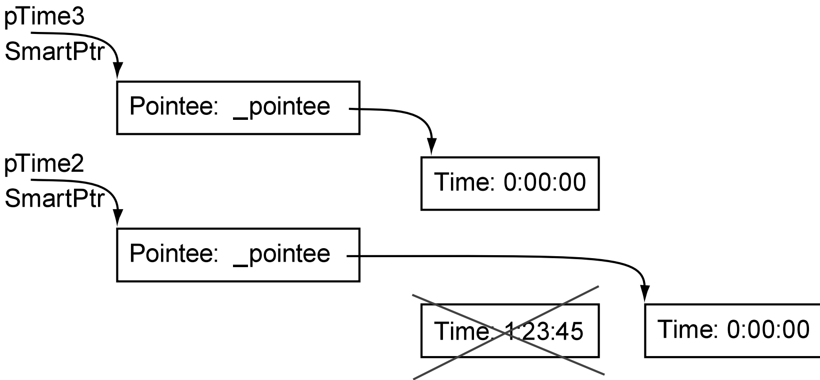


Рис. 5.21. Копіювальне присвоєння маніпуляторів

Як і копіювальний конструктор, присвоєння також супроводжується копіюванням. Спочатку розглянемо приклад. Візьмемо два незалежних маніпулятори (рис. 5.20):

Лістинг 5.158

```
SmartPtr pTime2(new Time(45,23, 1));
SmartPtr pTime3(new Time); // Рис. 5.21
```

Виконаємо присвоєння:

```
pTime2=pTime3;
```

Результат зображено на рис. 5.21: старе значення маніпулятора з лівої частини присвоєння видалено, натомість туди скопійовано значення правої частини.

Реалізацію присвоєння запишемо так:

Лістинг 5.159

```
// Копіювальне присвоєння
SmartPtr& SmartPtr::operator=(const SmartPtr& source) {
   if(&source!=this) {
// Видаляємо старе значення
      delete _pointee;
// Якщо маніпулятор оригіналу визначено,
      if(source._pointee)
// викликаємо копіювальний конструктор об'єкта маніпулювання
         _pointee=new Pointee(*source._pointee);
      else _pointee=0;
   }
   return *this;
}
```

Найважливіші оператори інтелектуального указника – специфічні операції над указниками. Почнемо з оператора *розіменування*, чи *непрямого доступу* (indirection operator). Передумова його застосування – визначеність маніпулятора, в іншому разі результат операції не визначено. Спробу розіменування невизначеного указника розцінюватимемо як помилку його клієнта, а не самого оператора розіменування. Тому в операторі не передбачено перевірки визначеності, але для полегшення налагодження програми, у якій його викликано, застосовуємо відповідний макрос суджень (а не умовний оператор). Нагадуємо, що макрос суджень діє лише в процесі налагодження програми й автоматично вимикається в її робочій версії.

Лістинг 5.160

```
// Операція розіменування маніпулятора
Pointee& SmartPtr::operator*() const {
```

```
   assert(defined());
   return *_pointee;
}
```

Зверніть увагу, що цей і наступний розглядуваний оператори сталі. Їхня особливість порівняно зі сталими операторами, які ми досі аналізували, полягає в тому, що вони повертають адресу, а не сталу адресу, як, наприклад, селектори. Сталість тут означає сталість маніпулятора, тобто згадані оператори не дозволять розірвати зв'язок інтелектуального указника з об'єктом. При цьому сталість не поширюється на значення самого об'єкта маніпулювання, стан якого може змінюватися. Тому оператор непрямого доступу може доставляти як лівобічне, так і правобічне значення.

Лістинг 5.161

```
// Розіменування: лівобічне значення
*pTime2=Time(1, 2, 3);
// Розіменування: правобічне значення
cout<<*pTime2<<endl;
```

Якщо оператор непрямого доступу дає значення, на яке показує інтелектуальний указник, то найважливіший оператор – *оператор доступу до члена класу* (class member access operator) – повертає саме значення указника (воно може бути нульовим):

Лістинг 5.162

```
// Операція доступу за указником Pointee
Pointee* SmartPtr::operator->() const {return _pointee;}
```

Оператор доступу до члена класу може поводитись і як бінарний, і як унарний. Застосування бінарного оператора зрозуміле й порівняно безпечне (якщо не брати до уваги можливості появи завислого указника). Потрібно звернутися до члена класу за маніпулятором його об'єкта, як зі звичайними указниками:

Лістинг 5.163

```
SmartPtr pTime0(new Time(0,0,1));
// Звертаємося до члена класу за указником:
pTime0->hours();     // Викликаємо селектор годин
pTime0->setHours(5); // Викликаємо модифікатор годин
```

Унарний оператор повертає сам указник. Він, як побачимо далі, доволі небезпечний. Правда, його достатньо складно записати, наприклад:

```
// Беремо значення самого указника
pTime0.operator->();
```

Це вже саме собою може бути певним захистом від некваліфікованого вживання.

Проаналізуємо безпечність монолітного маніпулятора стосовно втручання до нього ззовні. Виділимо дві причини можливих проблем:

- видалення об'єкта поза маніпулятором;
- неузгоджене керування кількома маніпуляторами одним об'єктом.

Розглянемо першу причину. Знову ж можливі два випадки. Навіть у разі осібного використання об'єкта одним маніпулятором аварійну ситуацію може спричинити видалення об'єкта за його указником, виконане відповідно до правила гарного тону оперування указниками (підрозділ 2.4.2). Так, завдяки наведеному в лістингу 5.164 фрагментові коду указник pt має доступ до об'єкта маніпулювання (рис. 5.22), оскільки не можна передавати права володіння об'єктом від указника до маніпулятора.

Лістинг 5.164

```
// Об'єкт у динамічній пам'яті (Heap)
Time* pt=new Time(0, 0, 1); SmartPtr pTimeDirect(pt);
```

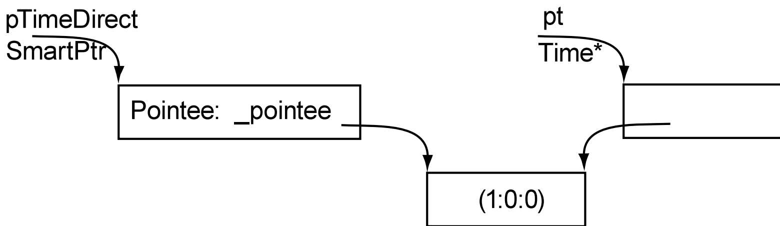


Рис. 5.22. Ініціалізація маніпулятора указником

Справді, ініціалізація маніпулятора указником pt залишає за ним повний доступ до об'єкта маніпулювання, зокрема право на його видалення (рис. 5.23).

```
// Небезпечна операція
delete pt;
```

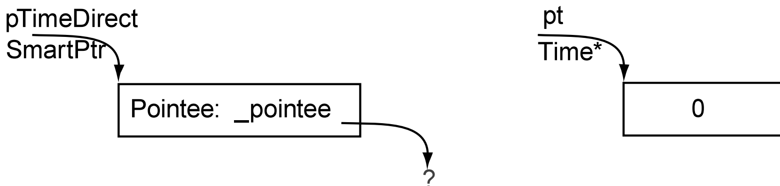


Рис. 5.23. Порушення цілісності маніпулятора

Після такого видалення рано чи пізно виникнуть неприємності: наприклад, коли маніпулятор спробує звернутися до цього об'єкта або просто виконати власний деструктор.

Залишається дотримувати правила хорошого тону в оперуванні маніпуляторами, яке полягає в тому, що указник відмовляється від прав на об'єкт, як у цьому фрагменті коду:

Лістинг 5.165

```
Time* pt=new Time(0, 0, 1);
SmartPtr pTimeDirect(pt); pt=0;
```

Якщо в попередньому прикладі помилки можна уникнути, додержуючи правила хорошого тону, то спроба маніпулювати звичайним іменованим об'єктом (рис. 5.24) узагалі не залишає жодного шансу на успіх. Програма, що міститиме наведений далі код, завершиться аварійно.

Лістинг 5.166

```
// Локальний об'єкт (Auto)
Time t(0, 1, 2); SmartPtr pTimeDirect(&t);
```

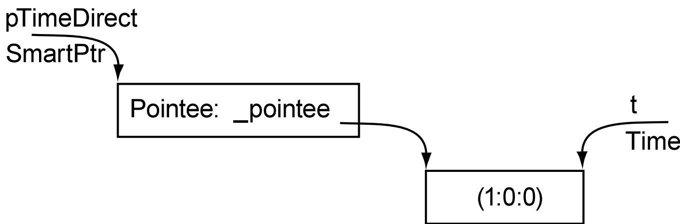


Рис. 5.24. Ініціалізація маніпулятора локальним іменованим об'єктом

З'явиться оператор видалення, який не відповідає жодному оператору виділення пам'яті. Справді, під час завершення блока, де визначено маніпулятор, буде викликано його деструктор, якому доведеться видаляти локальний об'єкт, який не було створено динамічно. Тому його слід видаляти не програмовано, а лише автоматично після виходу з відповідного блока (він може виявитися ширшим, аніж блок маніпулятора).

Тепер проаналізуємо другу причину небезпечної поведінки маніпуляторів, не розглядаючи більше об'єктів, створених поза маніпулятором. Однак навіть працюючи з об'єктом, створеним безпосередньо маніпулятором, той не може гарантовано осібно користуватися ним, а тому керування об'єктом може бути неузгоджене. У наведеному далі коді об'єкт маніпулювання передано іншому маніпулятору за допомогою оператора доступу.

Лістинг 5.167

```
// Позичений об'єкт (Borrowed)
SmartPtr pTime1(new Time(52,41,14));
SmartPtr pTime2(pTime1.operator->());
```

Програма, що міститиме такі команди, завершиться аварійно, тому що деструктор першого маніпулятора видалить об'єкт маніпулювання, а деструкторові другого дістанеться завислий указник.

Із тієї самої причини проблем повторного видалення не уникнути, якщо буде спроба спільного використання одного й того самого стороннього стосовно маніпулятора об'єкта.

Лістинг 5.168

```
// Спільний об'єкт (Common)
Time* pt=new Time(52,41,14);
SmartPtr pTime1(pt); SmartPtr pTime2(pt);
```

Вправа 5.53

Виконайте комп'ютерний експеримент із відтворенням ситуацій `Heap`, `Auto`, `Borrowed` і `Common`.

5.5.2. Клас гнучких маніпуляторів

Щоб уникнути аварійних завершень через спроби повторного видалення, доповнимо маніпулятор засобом від'єднання об'єкта маніпуляції, а для симетричності запропонуємо також метод приєднання об'єкта до маніпулятора. Одержимо *гнучкий маніпулятор*. Тепер, закінчивши обробку об'єкта за допомогою маніпулятора, можна роз'єднати їх, а в разі потреби приєднати до маніпулятора інший об'єкт. Відповідальність за подальше керування об'єктом буде покладено на клієнта маніпулятора.

Лістинг 5.169

```
// Тип для маніпулювання
typedef Time Pointee;
// Гнучкий маніпулятор об'єктами Pointee: програмовані приєднання
// та від'єднання, повне копіювання об'єкта маніпулювання
class SmartPtr {
private:
   Pointee* _pointee; // Об'єкт маніпулювання
public:
   class BadPtr; // Обробник помилкових ситуацій
// Конструктор, закритий від конвертування типів
   explicit SmartPtr(Pointee* p=0);
   SmartPtr(const SmartPtr&); // Копіювальний конструктор
   ~SmartPtr();// Деструктор
   SmartPtr& operator=(const SmartPtr&); // Копіювальне присвоєння
   bool defined() const; // Перевірка визначеності маніпулятора
// Специфічні операції над указниками
   Pointee& operator*() const; // Операція розіменування маніпулятора
```

```
   Pointee* operator->() const; // Операція доступу за указником
   SmartPtr& reset(Pointee&); // Приєднання об'єкта
   Pointee& release();// Від'єднання об'єкта
};
// Виведення об'єкта маніпулювання
ostream& operator<<(ostream&, const SmartPtr&);
```

Реалізувати нові методи не складно:

Лістинг 5.170

```
// Приєднання об'єкта
SmartPtr& SmartPtr::reset(Pointee& pt) {
   _pointee=&pt;
   return *this;
}
// Від'єднання об'єкта
Pointee& SmartPtr::release() {
   Pointee& _remember=*_pointee; _pointee=0;
   return _remember;
}
```

Від'єднанням об'єкта маніпулювання можна доповнити правила гарного тону, що ілюструє такий програмний код:

Лістинг 5.171

```
// Відновлення указника
Time* pt=new Time(0, 0, 1); SmartPtr pTimeDirect1(pt);
pt=0; pt=pTimeDirect1.release(); cout<<pt<<endl;
// Можна обробляти й видаляти без проблем
delete pt; pt=0;
```

Повернувши звичайному указникові позичені на певний час права (рис. 5.25), можна знову повноцінно оперувати ним.

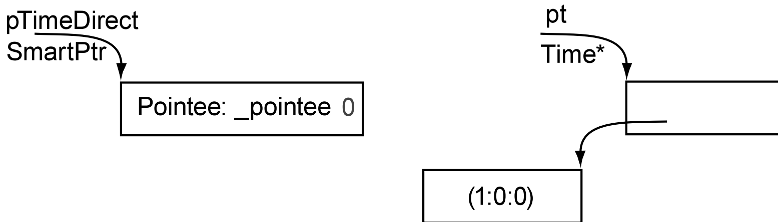


Рис. 5.25. Від'єднання об'єкту маніпулювання

Тепер видалення об'єкта через указник більше не вплине на маніпулятор (рис. 5.26).

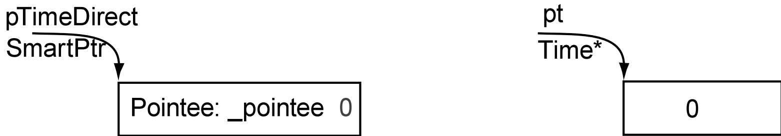


Рис. 5.26. Видалення від'єднаного об'єкта

Так само буде розв'язано проблему передавання маніпулятору іменованого об'єкта:

Лістинг 5.172

```
// Звільнення іменованого об'єкта
Time t(0, 1, 2); SmartPtr pTimeDirect2(&t); pTimeDirect2.release();
```

Після від'єднання об'єкта маніпулювання більше не виникне суперечностей (рис. 5.27).

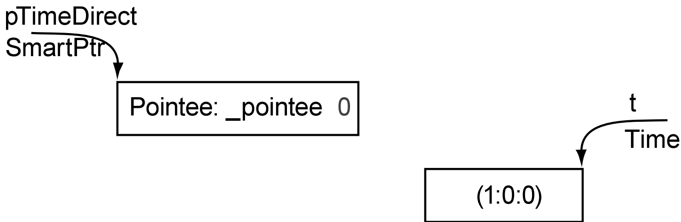


Рис. 5.27. Відєднання локального об'єкта маніпулювання

Вправа 5.54

Виконайте комп'ютерний експеримент і проаналізуйте поведінку гнучкого маніпулятора.

Застосування гнучкого маніпулятора потребує певної акуратності стосовно від'єднаного об'єкта. Від'єднання без запам'ятовування об'єкта маніпуляції може призвести до засмічення пам'яті.

Вправа 5.55

Проаналізуйте й виправте наведений далі код.

Лістинг 5.173

```
SmartPtr pTime1(new Time(1,2,3));
SmartPtr pTime2(pTime1); pTime1.release(); pTime2.release();
```

Застосування гнучких маніпуляторів дає змогу регламентувати передавання об'єкта маніпулювання від одного маніпулятора до іншого.

```
// Передавання об'єкта                                        Лістинг 5.174
Time* pt=new Time(0, 1, 2); SmartPtr pTimeDirect1(pt);
SmartPtr pTimeDirect2(pTimeDirect1); pTimeDirect1.release();
```

5.5.3. Клас осібних маніпуляторів

Проаналізуємо концепцію маніпулятора, який під час кожного копіювання не копіює об'єкт маніпуляції, а автоматично від'єднує його від оригіналу. Назвемо його *осібним маніпулятором* із винятковими правами на об'єкт маніпулювання. Через копіювання осібного маніпулятора оригінал утрачає повноваження на об'єкт маніпулювання, і виняткові права на нього передаються копії маніпулятора.

До речі, для розглянутих досі маніпуляторів у класах об'єктів маніпулювання мало бути присвоєння. Спроба скопіювати маніпулятор, застосований до класів без присвоєння, наприклад стека, черги чи іншого контейнера, спричинятиме синтаксичні помилки. Осібний маніпулятор виявляється стійким до того, що в об'єктах маніпуляції немає присвоєнь. Копіювальний конструктор і копіювальне присвоєння в ньому будуть полягати у від'єднанні об'єкта маніпулювання від одного маніпулятора та передання його іншому.

Визначення класу при цьому не зміниться, зміниться лише реалізація:

```
// Тип для маніпулювання                                      Лістинг 5.175
typedef Time Pointee;
// Осібний маніпулятор об'єктами Pointee: виключна власність
// на об'єкт маніпулювання
class SmartPtr {
private:
   Pointee* _pointee; // Об'єкт маніпулювання
public:
   class BadPtr; // Обробник помилкових ситуацій
// Конструктор, закритий від конвертування типів
   explicit SmartPtr(Pointee* p=0);
   SmartPtr(SmartPtr&); // (Не)копіювальний конструктор
   ~SmartPtr();// Деструктор
   SmartPtr& operator=(SmartPtr&); // (Не)копіювальне присвоєння
   bool defined() const; // Перевірка визначеності маніпулятора
// Специфічні операції над указниками
   Pointee& operator*() const; // Операція розіменування маніпулятора
   Pointee* operator->() const; // Операція доступу за указником
   SmartPtr& reset(Pointee&); // Приєднання об'єкта
   Pointee& release(); // Від'єднання об'єкта
};
```

```
// Виведення об’єкта маніпуляції
ostream& operator<<(ostream&, const SmartPtr&);
```

Тепер діятиме особлива концепція копіювального конструктора та присвоєння: маніпулятор-оригінал передає своє право власності на об’єкт маніпулювання маніпулятору-копії. При цьому сам оригінал перетворюється на невизначений маніпулятор. Зверніть увагу на тип параметра для копіювання: він не сталий, оскільки сам оригінал змінюється під час копіювання.

Лістинг 5.176

```
// (Не)копіювальний конструктор
SmartPtr::SmartPtr(SmartPtr& source):
// Ініціалізація копії маніпулятора
      _pointee(source._pointee) {
   source._pointee=0; // Оригінал утратив об’єкт маніпулювання
   return;
}
```

За об’єкт маніпулювання тепер відповідає новостворений маніпулятор. Ось так стає неможливим спільне використання одного й того самого ресурсу кількома маніпуляторами, а також унеможливлюється несанкціоноване видалення об’єкта відразу кількома маніпуляторами.

Розглянемо приклад використання копіювального конструктора. Створимо маніпулятор

```
SmartPtr pTime1(new Time(45, 23, 1));
```

Результат відображено на рис. 5.28.

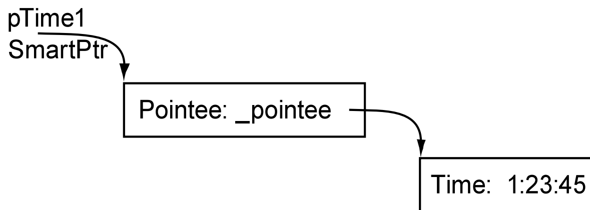


Рис. 5.28. Ініціалізація маніпулятора

Тепер застосуємо до нього копіювальний конструктор

```
SmartPtr pTime2(pTime1);
```

дію якого продемонстровано на рис. 5.29. Бачимо, що після копіювання оригінал став невизначеним. Утім такий стан маніпулятора не аварійний.

Із цього погляду операція присвоєння як передавання повноважень безпечніша, ніж присвоєння монолітних або гнучких указників, тому що вона не перешкоджає видаленню об'єкта маніпуляції.

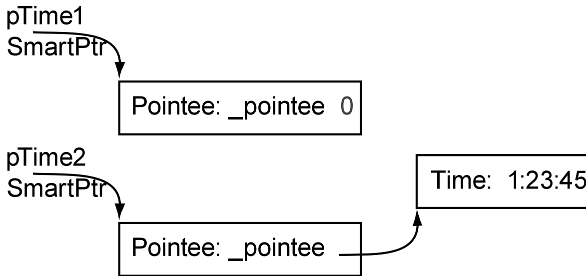


Рис. 5.29. Передача повноважень щодо об'єкту маніпулювання

Так само діє оператор присвоєння: маніпулятор правої частини втрачає повноваження над об'єктом, який передано у виключне володіння маніпулятору лівої частини присвоєння. Відмінність від конструктора копіювання полягає у видаленні попереднього значення лівої частини, щоб воно не залишилося сміттям.

Лістинг 5.177

```
// (Не)копіювальне присвоєння
SmartPtr& SmartPtr::operator=(SmartPtr& source) {
   if(&source!=this) {
      delete _pointee; // Видаляємо старий об'єкт маніпулювання
      _pointee=source._pointee; // Приєднуємо новий
      source._pointee=0; // Від'єднуємо об'єкт від оригіналу маніпулятора
   }
   return *this;
}
```

Ось приклади застосування присвоєння в контексті попереднього прикладу (рис. 5.30).

Лістинг 5.178

```
SmartPtr pTime1(new Time(45, 23, 1));
SmartPtr pTime2(pTime1); SmartPtr pTime3(new Time);
```

Виконаємо присвоєння

```
pTime2=pTime3;
```

Результат відображено на рис. 5.31.

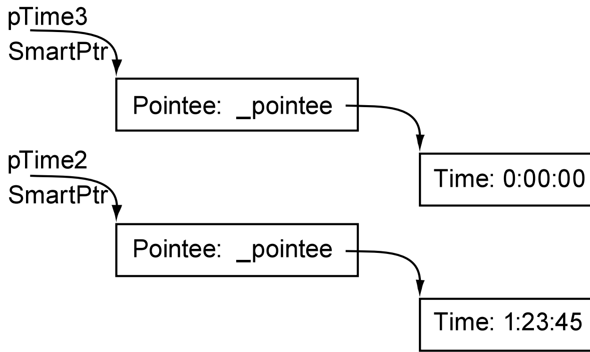


Рис. 5.30. Ініціалізація незалежних маніпуляторів

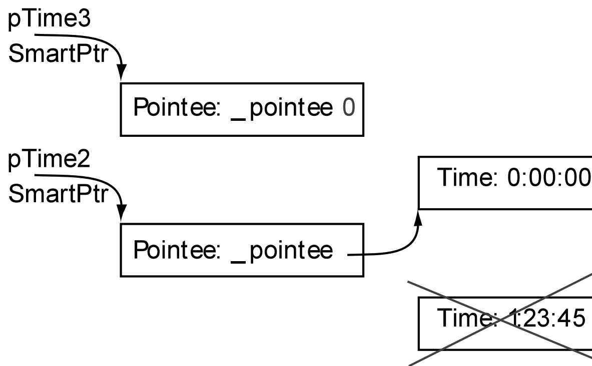


Рис. 5.31. Передача повноважень копіювальним присвоєнням

Як бачимо, попереднє значення `pTime2` видалено, об'єкт передано від `pTime3` до `pTime2`, `pTime3` стає невизначеним.

Унаслідок особливої концепції конструктора копіювання осібним маніпуляторам властивий особливий спосіб передавання параметрів за допомогою значення: фактичний параметр утрачає права на об'єкт маніпулювання. Спочатку визначимо функцію, яка через параметр одержує значення маніпулятора:

Лістинг 5.179

```
// Тепер передавання параметра за допомогою значення
// не оберігає фактичний параметр від змін. Функція, яка
// нічого не робить, а лише показує прийнятий нею параметр
void parameterEater(SmartPtr toBePassed) {
```

```
    cout<<toBePassed<<endl;
    return;
}
```

Наведемо приклад її застосування:

Лістинг 5.180

```
// Створюємо осібний маніпулятор
SmartPtr toBeEaten(new Time(1, 1, 1));
// Виводимо його. Результат відображено на рис. 5.32
cout<<toBeEaten<<endl;
// Викликаємо функцію. Результат відображено на рис. 5.33
parameterEater(toBeEaten);
// Виводимо фактичний параметр. Із рис. 5.34 видно, що він став невизначеним
cout<<toBeEaten<<endl;
```

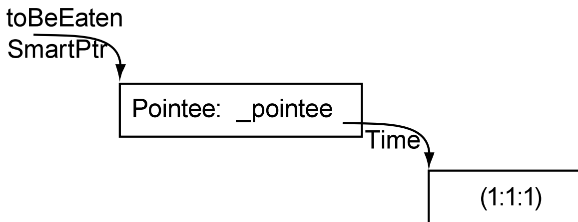


Рис. 5.32. Ініціалізація маніпулятора

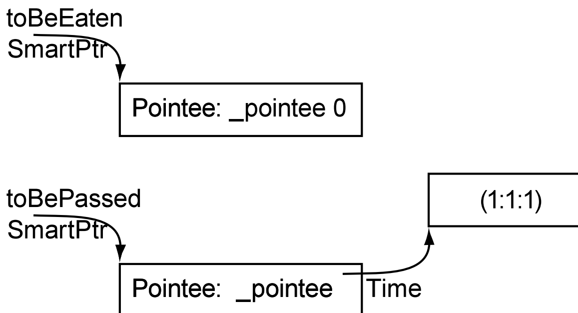


Рис. 5.33. Передача фактичного параметра

Таке передавання, що призводить до руйнування фактичного параметра, потребує серйозних застережень, адже воно принципово не відповідає концепції передавання параметрів за допомогою значення. Тому доходимо висновку, що спроектований нами клас маніпуляторів іще можна вдосконалювати, а поки що для того, щоб не спотворювати семантики передавання параметрів за допомогою значення, у наступній версії класу

Рис. 5.34. Невизначеність фактичного параметра після виходу із функції

маніпуляторів закриємо доступ до копіювального конструктора. Тепер можливі два варіанти. У першому залишаємо реалізацію, але закриваємо копіювальний конструктор. Тоді він стає недоступним для всіх зовнішніх клієнтів класу, але залишається доступним як для методів класу, так і для програмних конструкцій, наділених особливими правами доступу (якщо такі з'являться):

Лістинг 5.181

```
typedef Time Pointee; // Тип для маніпулювання
// Осібний маніпулятор об'єктами Pointee: виключна власність
// над об'єктом маніпулювання, закритий копіювальний конструктор
class SmartPtr {
private:
   Pointee* _pointee; // Об'єкт маніпулювання
   SmartPtr(const SmartPtr&); // Закритий (не)копіювальний конструктор
public:
   class BadPtr; // Обробник помилкових ситуацій
// Конструктор, закритий від конвертування типів
   explicit SmartPtr(Pointee* p=0);
   ~SmartPtr(); // Деструктор
   SmartPtr& operator=(const SmartPtr&); // (Не)копіювальне присвоєння
   bool defined() const; // Перевірка визначеності маніпулятора
// Специфічні операції над указниками:
   Pointee& operator*() const; // операція розіменування маніпулятора
   Pointee* operator->() const; // операція доступу за указником
   SmartPtr& reset(Pointee&); // Приєднання об'єкта
   Pointee& release(); // Від'єднання об'єкта
};
// Виведення маніпульованого об'єкта
ostream& operator<<(ostream&, const SmartPtr&);
```

У другому варіанті залишаємо копіювальний конструктор без реалізації. Тепер його неможливо викликати навіть ізсередини класу (правда, із діагностуванням помилки компонувальника):

Лістинг 5.182

```
typedef Time Pointee; // Тип для маніпулювання
// Осібний маніпулятор об'єктами Pointee: виключна власність над об'єктом
// маніпулювання, закритий нереалізований копіювальний конструктор
class SmartPtr {
private:
   Pointee* _pointee; // Об'єкт маніпулювання
```

```
   SmartPtr(const SmartPtr&); // Невикористовувана функція
public:
   class BadPtr; // Обробник помилкових ситуацій
// Конструктор, закритий від конвертування типів
   explicit SmartPtr(Pointee* p=0);
   ~SmartPtr();// Деструктор
   SmartPtr& operator=(const SmartPtr&); // (Не)копіювальне присвоєння
   bool defined() const; // Перевірка визначеності маніпулятора
// Специфічні операції над указниками:
   Pointee& operator*() const; // операція розіменування маніпулятора
   Pointee* operator->() const; // операція доступу за указником
   SmartPtr& reset(Pointee&); // Приєднання об'єкта
   Pointee& release(); // Від'єднання об'єкта
};
// Виведення маніпульованого об'єкта
ostream& operator<<(ostream&, const SmartPtr&);
```

В обох варіантах передавання параметрів за допомогою значення стане неможливим для клієнтів класу, а тому в ситуаціях, коли за звичайних умов таке передавання могло б статися, тепер доведеться передавати параметри як сталі відсилки. Це повністю усуває проблему їх спотворення. Перепишемо функцію з лістингу 5.179:

Лістинг 5.183

```
// Безпечна щодо руйнування параметра функція
void parameterEater(const SmartPtr& toBePassed) {
   cout<<"Pased into the function: "; cout<<*toBePassed<<endl;
   return;
}
```

Знову розглянемо її попередній виклик:

Лістинг 5.184

```
Створимо осібний маніпулятор
SmartPtr toBeEaten(new Time(1, 1, 1)); //
// Виведемо його. Результат такий самий, як на рис. 5.32
cout<<toBeEaten<<endl;
// Викличемо функцію. Результат побачимо на рис. 5.35
parameterEater(toBeEaten);
cout<<toBeEaten<<endl; // Виведемо фактичний параметр. Він не змінився
```

Вправа 5.56

Проаналізуйте безпечність передавання параметра за допомогою значення для кожного з розглянутих раніше варіантів маніпуляторів – монолітного та гнучкого. Чи можна твердити, що стан об'єкта маніпуляції після виходу з кожної функції, що приймає параметр за допомогою значення, залишиться незмінним?

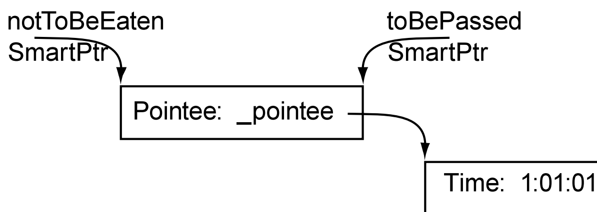


Рис. 5.35. Передача маніпулятора відсилкою

Вправа 5.57

Дайте відповідь на запитання попередньої вправи в разі передавання параметра за допомогою сталої відсилки для маніпулятора із закритим копіювальним конструктором.

5.5.4. Порівняння маніпуляторів з указниками

Порівняємо можливості створених нами маніпуляторів із можливостями звичайних указників. Як видно з підрозділу 5.5.1, крім розіменування, доступу та присвоєння, визначено ще два види операцій над указниками: булеві операції порівняння й операції адресної арифметики – збільшення та зменшення.

Розглянемо спочатку порівняння. Можливість його виконання свідчить про те, що два маніпулятори володіють одним і тим самим об'єктом. Позитивний результат порівняння для осібного маніпулятора буде розцінено швидше як програмну помилку, але зазначену операцію можна цілком коректно використовувати, скажімо, у судженнях `assert`.

Вправа 5.58

Доповніть клас осібних маніпуляторів утилітами перевірки на рівність

```
bool operator==(const SmartPtr&, const SmartPtr&);
```

і нерівність

```
bool operator!=(const SmartPtr&, const SmartPtr&);
```

Іще одна широковживана операція крім порівняння указників один з одним – порівняння указника з нулем для перевірки його визначеності. Для звичайного указника це має такий вигляд:

Лістинг 5.185

```
Time* pt=new Time(1, 2, 3);
if(pt==0) cout<<"Pointer defined";
if(pt!=0) cout<<"Pointer undefined";
```

Спроба порівняти з нулем маніпулятор завершиться синтаксичною помилкою. Клас містить особливу функцію `defined()` для перевірки на визначеність; нею й доведеться користуватися. Так само невдалими виявляться «хакерські» скорочення на зразок `if(pt)`… або `if(!pt)`….

На перший погляд, проблемі можна зарадити, доповнивши клас маніпуляторів оператором перетворення типів. Його можна викликати неявно (тоді він не матиме позначення) чи явно за допомогою операції перетворення типів (наприклад, `static_cast<Pointee*>` у розглядуваному випадку), якщо тип маніпулятора перетворювати до типу об'єкта маніпуляції:

Лістинг 5.186

```
// Оператор перетворення SmartPtr до Pointee*
SmartPtr::operator Pointee*() {return _pointee;}
```

На жаль, таке рішення майже перекреслює переваги, одержані завдяки маніпуляторам: знову не можна відрізнити маніпулятор від масиву маніпуляторів.

Вправа 5.59

Проаналізуйте поведінку описаного далі класу маніпуляторів, наділеного конвертором типів:

Лістинг 5.187

```
// Клас маніпуляторів із некоректною поведінкою
class SmartPtr {
private:
   Pointee* _pointee; // Об'єкт маніпулювання
   SmartPtr(const SmartPtr&); // Невикористовувана функція
public:
   class BadPtr; // Обробник помилкових ситуацій
// Конструктор, закритий від конвертування типів
   explicit SmartPtr(Pointee* p=0);
   ~SmartPtr();// Деструктор
   SmartPtr& operator=(const SmartPtr&); // (Не)копіювальне присвоєння
   bool defined() const; // Перевірка визначеності маніпулятора
// Специфічні операції над указниками:
   Pointee& operator*() const; // операція розіменування маніпулятора
   Pointee* operator->() const; // операція доступу за указником
   SmartPtr& reset(Pointee&); // Приєднання об'єкта
   Pointee& release(); // Від'єднання об'єкта
   SmartPtr::operator Pointee*(); // Конвертор типів
};
// Виведення маніпульованого об'єкта
ostream& operator<<(ostream&, const SmartPtr&);
```

Поясніть результати виконання наведеної далі тестової програми:

Лістинг 5.188

```
int main() {
   SmartPtr pTime(new Time(0, 0, 1));
   cout<<pTime+1<<endl; cout<<pTime[1]<<endl;
   Time* pt=&pTime[2]; cout<<*pt<<endl;
   return 0;
}
```

5.5.5. Клас вкладених маніпуляторів

Маніпулятори мають одну цікаву особливість, яка дає змогу будувати ланцюжки маніпулювань. Спочатку згадаймо застосування операцій доступу до указників:

```
// Указник першого рівня: указник на час
Time* pt=new Time(11, 11, 11);
```

Якщо оператор доступу `->` застосовано до указника, то той має бути указником на структуру чи клас. Тоді оператор уважають стандартним бінарним, а його другий аргумент має бути іменем члена класу.

Лістинг 5.189

```
// Звертаємося до атрибутів часу за допомогою
// операції доступу за маніпулятором
pt->hours(); pt->minutes(); pt->seconds();
```

Тоді операція доступу за маніпулятором `->` діє так само, як відповідна операція над указниками, а тому зводиться до суперпозиції операції розіменування * та «крапка-операції» селектора.

Лістинг 5.190

```
// Звертаємося до атрибутів часу суперпозицією
// розіменування та селектора
(*pt).hours(); (*pt).minutes(); (*pt).seconds();
```

Тепер візьмемо указник другого рівня:

```
// Указник другого рівня: указник на указник
Time** ppi=&pi;
```

До нього не можна застосувати оператор доступу за указником, оскільки значення указника не є структурою.

Інша справа – маніпулятор. Якщо перший аргумент виявиться не указником базового типу, а об'єктом програмованого типу – маніпулятором, то до нього буде застосовано програмований оператор доступу, який відповідно до свого визначення поверне як результат указник на об'єкт маніпуляції. Якщо ж оператор матиме другий аргумент, то ним має бути ім'я члена класу маніпуляції, і стандартну операцію доступу буде застосовано вже до указника на об'єкт маніпуляції.

```
// Маніпулятор: указник на час                                Лістинг 5.191
SmartPtr pTime=SmartPtr(new Time(22, 22, 22));
pTime->hours();// Оператор доступу за два етапи
pTime.operator->()->hours();// Рівнозначно
```

Об'єктом маніпулювання знову може виявитися маніпулятор. Як приклад розглянемо клас маніпуляторів маніпуляторами часу. Наведемо фрагмент визначення цього класу та реалізацію оператора доступу, залишивши читачеві право виконати повну реалізацію.

```
typedef SmartPtr NestedPointee; // Тип для маніпулювання   Лістинг 5.192
// Маніпулятор маніпуляторами
class NestedSmartPtr {
private:
   NestedPointee* _pointee; // Об'єкт маніпулювання
   NestedSmartPtr(NestedSmartPtr&); // Невикористовувана функція
public:
// Конструктор, закритий від конвертування типів
   explicit NestedSmartPtr(NestedPointee* pt=0);
// Деструктор
   ~NestedSmartPtr();
// (Не)копіювальне присвоєння
   NestedSmartPtr& operator=(NestedSmartPtr&);
// Операція доступу за указником
   const NestedPointee& operator->() const;
};
```

Тепер сигнатура оператора доступу інша, ніж була в маніпуляторі першого рівня. Якби тип результату був указником, то діяла б описана раніше схема, й оператор доступу було б застосовано до членів класу маніпулятора. Проте оскільки результат – відсилка до об'єкта, то до нього буде застосовано його програмовану операцію.

Тому оператор доступу до об'єкта маніпуляції, показаний у лістингу 5.193,

```
NestedSmartPtr ppTime(new SmartPtr(new Time(1, 2, 3)));   Лістинг 5.193
// Доступ до члена класу Time за допомогою маніпулятора другого рівня
ppTime->hours();
```

буде виконано за допомогою послідовного доступу: спочатку від маніпулятора другого рівня до маніпулятора першого, потім – від маніпулятора першого рівня до указника на об'єкт маніпуляції та, нарешті, до потріб-

ного члена в об'єкті маніпуляції. Наведений далі лістинг демонструє, що кожен крок у цій послідовності можна записати за допомогою окремого виразу. Ці вирази один за одним повертатимуть свій результат: об'єкт, указник, член класу.

Лістинг 5.194

```
NestedSmartPtr ppTime(new SmartPtr(new Time(1, 2, 3)));
// Доступ до об'єкта SmartPtr (за допомогою SmartPtr&)
ppTime.operator->();
// Доступ до указника на Time (за допомогою Time*)
ppTime.operator->()->operator->();
// Доступ до члена класу Time
ppTime.operator->()->.operator->()->hours();
```

Схему зв'язків між об'єктами та їх маніпуляторами наведено на рис. 5.36.

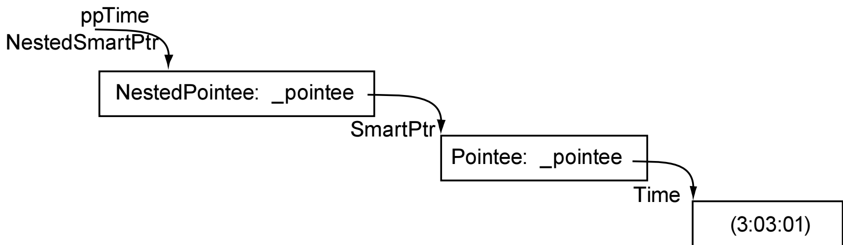


Рис. 5.36. Вкладений маніпулятор

Вправа 5.60

Завершіть визначення та реалізацію для класу `NestedSmartPtr`.

5.5.6. Маніпулятори масивів

Маніпулятори відрізняються від звичайних указників більшою захищеністю від аварійних ситуацій. Стандартна бібліотека `STL` містить клас інтелектуальних указників `auto_ptr`, які діють подібно до розглянутих нами маніпуляторів. Утім для маніпуляторів ми все ще не визначили операції індексування та адресної арифметики, учинивши так навмисне, аби відрізняти маніпулятори від масивів.

Після того, як ми досить детально проаналізували маніпулятори окремих об'єктів, можна почати створювати маніпулятори масивами. Розглянемо приклад класу, об'єкти якого поводяться водночас як указники та масиви. Порівняно зі звичайними маніпуляторами цей клас доповнено операціями адресної арифметики й індексування та засобами організації циклу.

```
typedef Time Pointee; // Тип для маніпулювання                  Лістинг 5.195
// Клас маніпуляторів масивами
class ArrayHandler {
private:
   size_t _size; // Розмір масиву
   Pointee* _pointee; // Указник на масив
   Pointee* _end; // Указник за масивом
   mutable Pointee* _current; // Указник поточного елемента масиву
   ArrayHandler(ArrayHandler&); // Закритий конструктор копіювання
public:
   class MisUse{}; // Обробник помилкових ситуацій
// Конструктор, закритий від конвертування типів
   explicit ArrayHandler(const size_t size=0);
   ~ArrayHandler(); // Деструктор
   ArrayHandler& operator=(ArrayHandler&); // (Не)копіювальне присвоєння
// Перевірка визначеності маніпулятора
   bool defined() const {return _pointee != 0;}
   size_t size() const {return _size;} // Селектор розміру масиву
// Специфічні операції над указниками
// Операція розіменування маніпулятора
   Pointee& operator*() const {return *_current;}
// Операція доступу за указником
   Pointee* operator->() const{return _current;}
   void start() const {_current=_pointee;} // Почати обробку масиву
// Перевірити вичерпання масиву
   bool stop() const {return _current==_end;}
// Операції індексування
   Pointee& operator[](const size_t i);
   const Pointee& operator[](const size_t i) const;
// Операції адресної арифметики
// Перемістити указник поточного елемента масиву
   ArrayHandler& operator+(size_t);
   const ArrayHandler& operator+(size_t) const;
// Операції префіксного інкременту
   ArrayHandler& operator++();
   const ArrayHandler& operator++() const;
// Операції префіксного декременту
   ArrayHandler& operator--();
   const ArrayHandler& operator--() const;
};
// Оператор виведення масиву
ostream& operator<<(ostream&, const ArrayHandler&);
```

Наведемо реалізації деяких методів маніпулятора масивів.

```
// Конструктор
ArrayHandler::ArrayHandler(const size_t size):
// Атрибути ініціалізуються згідно з порядком визначення
      _pointee(size == 0 ? 0 : new Pointee [size]), _size(size),
      _current(_pointee), _end(_pointee + _size) {}
// Деструктор
ArrayHandler::~ArrayHandler() {delete [] _pointee; return;}
// (Не)копіювальне присвоєння
ArrayHandler& ArrayHandler::operator=(ArrayHandler& src) {
   if (this != &src) {
      delete[] _pointee; _pointee = src._pointee;
      _current = src._current; _size = src._size; _end = src._end;
      src._pointee = 0; src._current = 0; src._size = 0; src._end = 0;
   }
   return *this;
}
// Перемістити указник поточного елемента масиву
ArrayHandler& ArrayHandler::operator+(const size_t i) {
   _current = _pointee + i;
   return *this;
}
// Оператор виведення масиву
ostream& operator<<(ostream& os, const ArrayHandler& ah) {
   for (size_t i = 0; i<ah.size(); ++i)
      os<<ah[i]<<',';
   os<<endl;
   return os;
}
```

Вправа 5.61

Завершіть і перевірте реалізацію класу `ArrayHandler`, виконайте наведену далі тестову програму та проаналізуйте застосовані в ній способи організації циклів на маніпуляторі масивів і способи доступу до об’єктів маніпуляції.

Лістинг 5.196

```
int main() {
   size_t quantity=5; QuasiPtr pTime(quantity);
   for(size_t i=0; i<pTime.size(); ++i)
      pTime[i]=Time(0, 0, i);
   pTime.start();
   while(!pTime.stop()) {
      cout<<*pTime<<endl; ++pTime;
   }
```

```
   pTime.start(); i=1;
   while(!pTime.stop()) {
      *pTime=Time(i, i, i); ++pTime; ++i;
   }
   for(size_t k=0; k<pTime.size(); k++)
      cout<<*(pTime+k)<<endl;
   return 0;
}
```

Зверніть увагу на те, що немає операцій постфіксного інкременту. Причина, звісно, полягає в тому, що не виконується копіювання, потрібне для реалізації цих операцій.

Підбиваючи підсумок, скажемо, що управління пам'яттю – дуже делікатна сфера програмування. Інкапсуляція указників у маніпуляторах ще не розв'язує всіх проблем, а застосування маніпуляторів потребує детального аналізу їхніх властивостей, відстеження всіх наслідків і сторонніх ефектів прийнятого рішення.

Опанувавши концепцію інтелектуального указника, ви можете вважати, що вийшли з категорії початківців і досягнули рівня середньої складності об'єктного програмування.

5.6. Функціональні об'єкти

Після приховування указників у маніпуляторах не можна не звернути увагу на ще одне поширене їх застосування. Ідеться про непрямий виклик функцій – як вільних, так і членів класів, розглянутих відповідно в підрозділах 4.12 і 5.13. Розглянемо тепер концепцію приховування указників функцій у так званих *функціональних об'єктах* (functional objects).

Щоб краще зрозуміти, де слабке місце указників функцій, згадаємо задачу обчислення визначених інтегралів за методом Сімпсона з лістингу 3.188. Як ви пам'ятаєте, підінтегральну дійсну функцію однієї дійсної змінної задавали указником і так досягали універсальності інтерфейсу, закладеного до сигнатури функції

```
double Simpson(const double a, const double b, const double eps,
   double (*const f)(double));
```

Завдяки застосуванню непрямого виклику за допомогою указника стає можливим застосування функції інтегрування до різних підінтегральних функцій. Достатньо лише передати її за допомогою параметра.

Проте все ще залишилася проблема, яку не було розв'язано задовільно: ми ілюстрували її на прикладі еліптичного інтеграла

$$I(a,b) = \frac{2}{\pi} \int_0^{\frac{\pi}{2}} \frac{dx}{\sqrt{a^2 \sin^2 x + b^2 \cos^2 x}} \, ,$$

підінтегральна функція якого містила два параметри `a` та `b`, для котрих не знайшлося місця в інтерфейсі.

5.6.1. Клас функторів

Запропонуємо рішення, що не має цієї вади. Функціональним об'єктом називатимемо екземпляр класу, у якому визначено операцію *виклику функції* (function call). Сам клас функціональних об'єктів називатимемо *функтором* (functor).

Розглянемо такий функтор:

Лістинг 5.197

```
// Об'єкти класу - параметричні функції,
// параметри яких задано атрибутами
class Elliptic {
private:
   double _a, _b; // Параметри функції
public:
// Конструктор ініціалізації параметрів
   Elliptic(double a=1, double b=2):_a(a),_b(b) {}
   ~Elliptic () {} // Деструктор
// Визначення функції функтора
   double operator()(double x) const {
      return 1.0/sqrt(_a*_a*cos(x)*cos(x)+_b*_b*sin(x)*sin(x));
   }
};
```

Чільне місце в ньому належить операторові виклику функції **`operator()`**. Це метод, який може мати скільки завгодно формальних параметрів (тут – один), а ті можуть мати задані за замовчуванням значення. Зрозуміло, що на правах методу оператор виклику функції має доступ до всіх атрибутів класу, які можуть відігравати роль додаткових параметрів.

Означимо функціональний об'єкт, наприклад

```
// Визначення функціонального об'єкта полягає в заданні
// значень його параметрів
Elliptic sample(3, 4);
```

Тепер можна викликати його операторну функцію за допомогою повного запису `sample.operator()(1.5)` або скороченого `sample(1.5)`.

У функції інтегрування відповідно змінимо заголовок, замінивши тип останнього аргументу функтором:

Лістинг 5.198

```
// Обчислення інтеграла еліптичних функцій
// методом Сімпсона із застосуванням функціонального об'єкта
double Simpson (const double a, const double b, const double eps,
// Функтор для функції інтегрування
                const Elliptic& f) {
   int n=2; double h=(b-a)*0.5; double ss=0; double s1=h*(f(a)+f(b));
   double s2=0; double s4=4*h*f(a+h); double s=s1+s2+s4;
   do {
      ss=s; n*=2; h/=2; s1*=0.5; s2=0.5*s2+0.25*s4; s4=0; int i=1;
      do {
         s4=s4+f(a+i*h); i+=2;
      } while(i<=n);
      s4=4*h*s4; s=s1+s2+s4;
   } while(fabs(s-ss)>eps);
   return s/3;
}
```

Цим ми перетворимо виклики функції `f` на виклики операторної функції відповідного функціонального об'єкта. Виклик самої функції інтегрування набуває вигляду

```
Simpson(0, pi/2, prec, Elliptic(a, b));
```

де `Elliptic(a, b)` – конструктор функціонального об'єкта з відповідними параметрами.

Вправа 5.62

Складіть тестову програму, що перевіряє коректність функції обчислення інтеграла еліптичних функцій методом Сімпсона, склавши судження для порівняння обчисленого значення та відповідного арифметико-геометричного середнього (підрозділ 2.6.2.4) з урахуванням рівності

$$\frac{1}{M(a,b)} = \frac{2}{\pi}\int_{a}^{\frac{\pi}{2}} \frac{dx}{\sqrt{a^2 \sin^2 x + b^2 \cos^2 x}},$$

Яку наведено у вправі 3.35.

5.6.2. Застосування функторів

Щоправда, запропоноване розв'язання також має вади: інтегрування довільної підінтегральної функції, яку раніше було передано указником,

перетворилося на обчислення еліптичних інтегралів від функцій заданого вигляду, поданих за допомогою функціонального об'єкта. Розширення сфери застосування функції інтегрування за Сімпсоном із використанням функціонального об'єкта полягатиме в переході до узагальненої функції інтегрування, типовий параметр якої задаватиме потрібний функтор.

Лістинг 5.199

```
// Узагальнена функція обчислення інтеграла
// методом Сімпсона. Типовий параметр – функтор
template <class Function>
double Simpson (const double a, const double b, const double eps,
// Функтор для функції інтегрування
                const Function& f) {
   int n=2; double h=(b-a)*0.5; double ss=0; double s1=h*(f(a)+f(b));
   double s2=0; double s4=4*h*f(a+h); double s=s1+s2+s4;
   do {
      ss=s; n*=2; h/=2; s1*=0.5; s2=0.5*s2+0.25*s4; s4=0; int i=1;
      do {
         s4=s4+f(a+i*h); i+=2;
      } while(i<=n);
      s4=4*h*s4; s=s1+s2+s4;
   } while(fabs(s-ss)>eps);
   return s/3;
}
```

Вправа 5.63

Складіть програми для наближеного обчислення за допомогою узагальненої функції з лістингу 5.199 таких інтегралів:

а) еліптичного інтеграла (клас з лістингу 5.197)

б) функції нормального розподілу Гаусса $\phi_{\mu,\sigma^2}(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$

Сформулюйте судження для перевірки коректності обчислень.

Зауваження. *Особливість застосування функторів із використанням механізму узагальнених функцій полягає в економії вхідного коду (один текст шаблону функції буде використано для генерування кожної з його конкретизацій), але водночас для кожної конкретизації буде згенеровано власний об'єктний код – кожен зі своїм типом функціонального об'єкта.*

Зазначимо, що можна застосовувати указники на функції фактично в тому самому варіанті, як це було в разі передавання указника функції за допомогою параметра. Для цього достатньо визначити маніпулятор дійсної функції дійсної змінної та конкретизувати ним шаблон узагальненої функції інтегрування:

```
// Клас маніпуляторів дійсних функцій дійсної змінної      Лістинг 5.200
class RealFunction {
private:
// Функція-заглушка для замовчувань
   static double _dummy(double) {return 0;}
   double (*const _f)(double); // Указник на функцію маніпулювання
// Невикористовувані функції й операції
   RealFunction(const RealFunction&);
   RealFunction& operator=(const RealFunction&);
public:
// Конструктор
   RealFunction(double (*const f)(double)=_dummy): _f(f) {}
   ~RealFunction() {}  // Деструктор
// Оператор виклику функтора
   double operator()(double x) const {return _f(x);}
};
```

Тепер можна конкретизувати функцію `Simpson()` класом `RealFunction`:

```
cout.precision(12); const double prec=1.e-12;            Лістинг 5.201
const double pi=3.1415926535897932;
// Інтеграл від функції-заглушки
cout<<Simpson(0, pi, prec, RealFunction())<<endl;
// Інтеграл від синуса
cout<<Simpson(0, pi, prec, RealFunction(sin))<<endl;
// Інтеграл від косинуса
cout<<Simpson(0, pi, prec, RealFunction(cos))<<endl;
```

Розглянемо інші застосування функторів. Перше полягає у визначенні за допомогою функтора критерію пошуку потрібного елемента в заданому пошуковому просторі. Цим простором може бути контейнер, а критерій виражатиметься функціональним об'єктом. Наведемо приклад функтора, який визначатиме критерій пошуку елемента, що дорівнює заданому:

```
// Клас – функтор критеріїв пошуку                         Лістинг 5.202
class Criterion {
private:
   Elem _elem; // Шуканий елемент
public:
   Criterion(const Elem& elem): _elem(elem) {}
   ~Criterion();
// Функція перевірки рівності
   bool operator()(const Elem& elem) const {return elem==_elem;}
};
```

Застосуємо його до відшукання потрібного елемента в послідовності:

Лістинг 5.203

```
// Функція пошуку за критерієм what у послідовності where
bool search(const Sequence& where, const Criterion& what) {
   bool found=false; where.start();
   while(!where.stop()) {
      if(what(getElem()) {found=true; break;}
      where.next();
   }
   return found;
}
```

Далі наведено просту тестову програму пошуку:

Лістинг 5.204

```
int main() {
   Sequence seq(10);
   seq.insert(1).insert(2).insert(3).insert(4); cout<<seq;
   Criterion crit0(0), crit3(3);
   cout<<search(seq, crit0)<<endl; cout<<search(seq, crit3)<<endl;
   return 0;
}
```

Вправа 5.64

Складіть програму, яка підраховує кількість членів послідовності, що дорівнюють заданому параметру `a`.

Вправа 5.65

Напишіть програму, яка за заданою послідовністю `seq` будує нову послідовність, що складається з усіх членів послідовності `seq`, які більші ніж даний параметр `a`.

Функціональні об'єкти можуть мати багато інших застосувань, які полягають в організації особливого доступу до структури даних. Як приклад розглянемо задачу подання багатовимірних масивів. Для конкретності візьмемо двовимірну матрицю.

Як відомо, у мовах С та С++ використано особливу концепцію масиву масивів. Пам'ятаймо також про типову помилку початківців щодо запису `A[i, j]`, який за правилами мови програмування надає доступ не до елемента матриці з індексами `i` та `j`, а до `j`-го елемента одномірного масиву. Якоюсь мірою цю особливість можна обійти, запропонувавши для вибору елемента двовимірного масиву позначення `A(i, j)`, досить поширене в інших мовах програмування, похідних від ФОРТРАНу.

Лістинг 5.205

```
// Клас двовимірних дійсних матриць
class Matrix {
private:
   size_t _size_n; // Кількість рядків
   size_t _size_m; // Кількість стовпців
   double** _pMatrix; // Указник до масиву
// Невикористовувані функції й операції
   Matrix(const Matrix&);
   Matrix& operator=(const Matrix&);
public:
   Matrix(const size_t m, const size_t  n); // Конструктор
   ~Matrix(); // Деструктор
// Оператори індексування
   double& operator()(size_t, const size_t );
   double operator()(size_t, const size_t ) const;
// Селектори розмірів
   size_t size_n() const {return _size_n;}
   size_t size_m() const {return _size_m;}
};
// Виведення матриці
ostream& operator<<(ostream&, const Matrix&);
```

Реалізацію наведемо без коментарів:

Лістинг 5.206

```
// Конструктор
Matrix::Matrix(const size_t m, const size_t  n): _size_n(n), _size_m(m)
{
// Створюємо масив указників на рядки
   _pMatrix=new double*[_size_m];
// Ініціалізуємо указники на рядки
   for(unsigned int i=0; i<_size_m; i++)
      _pMatrix[i]=new double [_size_n];
// У кожен елемент масиву заносимо нульове значення
   for(unsigned int k=0; k<_size_m; k++)
      for(unsigned int j=0; j<_size_n; j++)
         _pMatrix[k][j]=0;
   return;
}
// Деструктор
Matrix::~Matrix() {
// Спочатку видаляємо кожен рядок
   for(unsigned int i=0; i<_size_m; i++)
      delete [] _pMatrix[i];
```

```
// Потім видаляємо масив рядків
   delete [] _pMatrix;
   return;
}
// Селектор-модифікатор
double& Matrix::operator()(const size_t i, const size_t  j) {
   return _pMatrix[i-1][j-1];
}
// Селектор
double Matrix::operator()(const size_t i, const size_t  j) const {
   return _pMatrix[i-1][j-1];
}
// Виведення матриці
ostream& operator<<(ostream& os, const Matrix& M) {
   for(unsigned int i=1; i<=M.size_m(); i++) {
      for(unsigned int j=1; j<=M.size_n(); j++)
         cout<<M(i, j)<<' ';
      cout<<endl;
   }
   return os;
}
```

Як простий приклад використання класу `Matrix` наведемо програму створення одиничної квадратної матриці:

Лістинг 5.207

```
int main() {
   Matrix E(5, 5);
   E(1, 1)=E(2, 2)=E(3, 3)=E(4, 4)=E(5, 5)=1; cout<<E;
   return 0;
}
```

Вправа 5.66

За допомогою функтора визначте клас `Matrix_nxn` двовимірних квадратних дійсних матриць вигляду

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}.$$

Означте й реалізуйте операції додавання та множення матриць, а також функцію швидкого піднесення матриці до натурального степеня.

Використання функціональних об'єктів як засобу індексування змусило відмовитися від квадратних дужок, але натомість ми одержали можливість звично записувати індекси через кому, до того ж маємо простий спосіб створення матриці довільних, а не обов'язково сталих, розмірів.

Вправа 5.67

Доповніть клас матриць класом обробки аварійних ситуацій, коли немає вільної пам'яті та індекси виходять за межі матриці.

Вправа 5.68

Означте операції додавання та множення матриць, використовуючи функтори для їх подання й інтелектуальні указники для результатів.

Ті, хто вивчає іноземні мови, знають, що про рівень володіння мовою можна судити за вмінням правильно вживати ідіоми та крилаті вирази. «Вищий пілотаж» об'єктно-орієнтованого програмування теж базується на ідіомах. Як інтелектуальний указник, так і функціональний об'єкт не є поняттями мови програмування. Однак вони виражають стійке поєднання вкладених до класу конструкцій, зрозуміти вживання яких можна, лише вийшовши за їхні синтаксичні рамки.

5.7. Асоціативні масиви

Зазвичай у книгах із програмування пишуть, що індексами масивів найчастіше бувають цілі числа; до цього додають, що, узагалі кажучи, ними можуть бути значення довільних зліченних типів, але в жодному разі не дійсні числа.

У зв'язку з цим можна пригадати знайомі кожному зі школи чотиризначні таблиці математичних функцій Володимира Модестовича Брадіса, що являють собою масиви з дійсними індексами – по одному на кожну елементарну функцію. Кожна ж така таблиця – це масив, який дійсному індексу зіставляє значення відповідної елементарної функції. Називаючи таблиці Брадіса масивами, ми не маємо на увазі масиви на апаратному рівні, утворені лінійно впорядкованими фізичними елементами пам'яті, та ще й обладнані механізмами швидкого доступу за допомогою модифікування базової адреси за вмістом індексного регістра. Звичайно, значення функцій у таблицях Брадіса теж упорядковано, але пошук потрібного елемента таблиці відбувається не за його номером, а за значенням аргументу, з яким цей елемент асоціюється як значення відповідної елементарної функції. Беручи з таблиці значення функції, наприклад $\sin(46^{\circ}35')$, ми не цікавимося, під

яким порядковим номером це число (тут – 0,7263) потрапило до таблиці. Нас цікавить лише відповідність значення функції значенню її аргументу, у даному випадку, наприклад, куту, вираженому в градусах.

Розглянемо узагальнення поняття масиву до поняття асоціативного масиву, індекси якого мають довільні типи, зокрема дійсний. Ідеться про поширення концепції масиву як відображення множини індексів $I_1 \times I_2 \times ... \times I_n$ на область D значень його елементів –

$$array : I_1 \times I_2 \times ... \times I_n \to D,$$

яке кожному допустимому набору індексів зіставляє значення відповідного елемента області даних $array(i_1, i_2, ..., i_n) = d$, на довільні, зокрема програмовані, типи індексів. Індекси в цьому разі називають *ключами* (key). Тоді ключі до таблиць Брадіса – значення аргументів елементарних функцій.

5.7.1. Клас асоціативних масивів

Проблему доступу до даних за індексами довільного типу допомагає розв'язати концепція *асоціативного масиву* (associative array). Як і в прикладі матриць із попереднього підрозділу, асоціативний масив – це своєрідна оболонка для доступу до агрегату даних, який являє собою масив, у найпростішому випадку влаштований із пар «ключ – асоційоване з ним значення». У складніших випадках ключів може бути більше.

Асоціативний масив – іще одна ідіома програмування, буквальне розуміння якої зводилося б до масиву пар, тоді як її справжній зміст полягає в зіставленні ключів асоційованим із ними значенням.

Як приклад розглянемо асоціативний масив імен, ключами в якому слугуватимуть ідентифікаційні коди.

Лістинг 5.208

```
// Асоціативний масив кадрів
class AssotiativeArray  {
private:
// Структура асоціації
   struct Pair {
      unsigned int _id; // Ключ
      string _name; // Асоційоване значення
// Конструктор асоціації
      Pair(string name="", const unsigned int id=0):
         _name(name), _id(id) {}
   };
   size_t _volume; // Ємність масиву
   size_t _size; // Поточний розмір масиву
```

```
   Pair* _pairs; // Уміст масиву
// Заборонені для використання в масивах
   AssotiativeArray(const AssotiativeArray&);
   AssotiativeArray& operator=(const AssotiativeArray&);
public:
   class BadArray; // Вбудований клас для обробки помилкових ситуацій
// Конструктор, закритий для конвертування типів
   explicit AssotiativeArray(size_t);
   ~AssotiativeArray();// Деструктор
// Селектор-модифікатор елемента масиву за ключем
   string& operator[](unsigned int);
// Селектор елемента масиву за ключем
   const string& operator[](unsigned int) const;
   size_t size() const {return _size;} // Розмірність масиву (селектор)
};
```

Будову елемента масиву задає закрита структура пар Pair, наділена конструктором пари за рядком і невід'ємним числом. Зазвичай розміри масиву задано двома атрибутами: _volume відповідає за ємність, а _size – за наповненість. Сам масив ідентифікують указником _pairs.

Конструктор копіювання й оператор присвоєння закриті зі зрозумілих причин (поясніть, яких). Клас BadArray, як завжди, відповідає за обробку помилок.

Конструктор асоціативного масиву не має спричиняти неявне перетворення типів, тому при ньому є кваліфікатор явного виклику **explicit**.

Лістинг 5.209

```
// Конструктор, закритий для конвертування типів
AssotiativeArray::AssotiativeArray(const size_t volume):
// Ініціалізація ємності
   _volume(volume+1),
// Виділення пам'яті під пари
   _pairs(new Pair[volume+1]) {
   _size=0;
   return;
}
```

Функція деструктора очевидна – видалення масиву, оскільки пам'ять для указника _pairs виділено як масив.

Лістинг 5.210

```
// Деструктор
AssotiativeArray::~AssotiativeArray() {delete [] _pairs;}
```

Два оператори індексування мають принципово різне призначення. Перший виконує роль модифікатора та призначений для пошуку наявно-

го елемента в асоціативному масиві. У разі невдалого результату в масиві буде створено місце для нового елемента.

Лістинг 5.211

```
// Селектор-модифікатор елемента масиву
string& AssotiativeArray::operator[](unsigned int id) {
   for(int i=_size; i>0;--i)
      if(id==_pairs[i]._id) return _pairs[i]._name;
// Пошук закінчився невдало
   if(_size+1==_volume) throw BadArray("No more space in array");
// Приготували занесення нового елемента
   _pairs[++_size]._id=id;
   return _pairs[_size]._name;
}
```

Другий оператор індексування відіграє виключно роль селектора та повертає порожній рядок, коли немає елемента із заданим індексом.

Лістинг 5.212

```
// Селектор елемента масиву
const string& AssotiativeArray::operator[] (unsigned int id) const {
   for(int i=_size; i>0; i--)
      if(id==_pairs[i]._id) return _pairs[i]._name;
   return _pairs[0]._name; // Пошук закінчився невдало
}
```

Перейдемо до операції виведення. Для її реалізації потрібен засіб перегляду масиву від початку до кінця. Звісно, міра наповненості масиву відома (це його поточний розмір), але немає операції, яка давала б доступ до всієї пари чи обох її частин.

Пригадаємо конструкцію ітераторів, уперше запроваджену ще в підрозділі 2.8.1. Відповідно до неї для використання ітераторів достатньо трьох конструкцій: указника на початок контейнера (у розглядуваному випадку – асоціативного масиву), указника на місце поза контейнером для окреслення його меж і указника на поточний елемент. Потрібні також три функції: одна – для початку ітерації, друга – для перевірки умови її завершення, третя – для переходу до наступного поточного елемента.

Доповнимо клас асоціативних масивів ітераторами за наведеною далі схемою.

Лістинг 5.213

```
// Асоціативний масив кадрів, доповнений ітераторами
class AssotiativeArray  {
private:
// Структура асоціації
   struct Pair {
```

```
      unsigned int _id; // Ключ
      string _name; // Асоційоване значення
// Конструктор асоціації
      Pair(string name="", unsigned int id=0): _name(name), _id(id) {}
   };
   size_t _volume; // Ємність масиву
   size_t _size; // Поточний розмір масиву
   Pair* _pairs; // Умiст масиву
// Ітератор: указник на першу пару: _pairs, указник на місце
// за останньою парою: _pairs+_size
   mutable Pair* _current; // указник на поточну пару
// Заборонені для використання в масивах методи
   AssotiativeArray(const AssotiativeArray&);
   AssotiativeArray& operator=(const AssotiativeArray&);
public:
   class BadArray; // Вбудований клас для обробки помилкових ситуацій
// Конструктор, закритий для конвертування типів
   explicit AssotiativeArray(size_t);
   ~AssotiativeArray();// Деструктор
// Селектор-модифікатор елемента масиву за ключем
   string& operator[](unsigned int);
// Селектор елемента масиву за ключем
   const string& operator[](unsigned int) const;
   size_t size() const {return _size;} // Розмірність масиву (селектор)
// Методи ітератора
   void start() const {_current=_pairs+1;} // Почати ітерацію
   void next() const {_current++;} // Виконати крок ітерації
   bool stop() const  // Перевірити вичерпання масиву
      {return (_current>_pairs+_size);}
// Додаткові селектори:
// Селектор імені поточної пари
   const string& getName() const {return _current->_name;}
// Селектор ключа поточної пари
   const unsigned int& getId() const {return _current->_id;}
};
// Оператор виведення масиву
ostream& operator<<(ostream&, const AssotiativeArray&);
```

Тепер можна реалізувати утиліту виведення, виконавши ітерацію за вмістом асоціативного масиву:

Лістинг 5.214

```
// Оператор виведення масиву
ostream& operator<<(ostream& os, const AssotiativeArray& ar) {
```

```
   ar.start();
   while(!ar.stop()) {
      cout<<ar.getId()<<':'<<ar.getName()<<endl; ar.next();
   }
   return os;
}
```

Наведемо також приклад простої програми для тестування асоціативного масиву:

Лістинг 5.215

```
int main() {
   AssotiativeArray staff(5);
   try {
      staff[123456789]="John the Programmer";
      staff[345612678]="Merry the Hacker";
      staff[247665617]="Bob the Cracker";
      staff[375665611]="Tom the Writer";
      staff[195473116]="Jimmy the Tester";
      staff[111111111]="Six's a crowd";
   }
   catch(AssotiativeArray::BadArray mes) {cerr<<mes<<endl;}
   cout<<staff<<endl;
   return 0;
}
```

Вправа 5.69

Доповніть клас асоціативного масиву парними операціями індексування для пошуку чи заповнення елемента масиву за іменем особи, використовуючи наведені далі сигнатури:

Лістинг 5.216

```
unsigned int& operator[](const string&);
const unsigned int& operator[](const string&) const;
```

Не забудьте про однозначність ключів. Результат протестуйте.

5.7.2. Прихована реалізація

На завершення розкриємо зміст ще однієї ідіоми програмування – *прихованої реалізації* (pointer to implementation), яку зазвичай називають PImpl. Згадаймо, що структуру пар `Pair`, яка задає формат зберігання даних в асоціативному масиві, ми розмістили в його закритій частині. Отже, ця структура закрита, але видима, а це по суті зайва деталь реалізації, несуттєва для використання асоціативного масиву. Її можна було прихо-

вати від клієнта класу. Тому в таких випадках визначення прихованої структури виносять до файлу реалізації, залишивши в визначенні класу `AssotiativeArray` лише попереднє оголошення `Pair`. При цьому, щоправда, доведеться відмовитися від відкритої реалізації окремих методів (яких саме?) безпосередньо в заголовному файлі, адже в них використано приховану структуру. З іншого боку, унесення реалізації до заголовного файлу хоча й обґрунтоване потребами вбудованої реалізації методів, видається невиправданим порушенням принципу приховування інформації.

Лістинг 5.217

```
// Асоціативний масив кадрів із прихованими парами
class AssotiativeArray  {
private:
   struct Pair; // Прихована структура асоціації
   size_t _volume; // Ємність масиву
   size_t _size; // Поточний розмір масиву
   Pair* _pairs; // Уміст масиву
// Ітератор: указник на першу пару: _pairs
// указник на місце за останньою парою: _pairs+_size
   mutable Pair* _current; // указник на поточну пару
// Заборонені для використання в масивах методи
   AssotiativeArray(const AssotiativeArray&);
   AssotiativeArray& operator=(const AssotiativeArray&);
public:
   class BadArray; // Вбудований клас для обробки помилкових ситуацій
// Конструктор, закритий для конвертування типів
   explicit AssotiativeArray(size_t);
   ~AssotiativeArray();// Деструктор
// Селектор-модифікатор елемента масиву за ключем
   string& operator[](unsigned int);
// Селектор елемента масиву за ключем
   const string& operator[](unsigned int) const;
   size_t size() const {return _size;} // Розмірність масиву(селектор)
// Методи ітератора
   void start() const; // Почати ітерацію
   void next() const; // Виконати крок ітерації
   bool stop() const; // Перевірити вичерпання масиву
// Додаткові селектори:
   const string& getName() const; // Селектор імені поточної пари
   const unsigned int& getId() const; // Селектор ключа поточної пари
};
// Оператор виведення масиву
ostream& operator<<(ostream&, const AssotiativeArray&);
```

Вправа 5.70

Реалізуйте клас асоціативного масиву з прихованими парами.

Вправа 5.71

Означте для асоціативного масиву операції доступу за указником, розіменування та машинної арифметики.

Вправа 5.72

Означте й реалізуйте як асоціативний масив таблицю значень інтеграла ймовірностей

$$\Phi(U) = \frac{1}{\sqrt{2\pi}} \int_0^U e^{-\frac{t^2}{2}} dt$$

на відрізку [0; 3,0] із кроком 0,001. Значення слід заносити до масиву в міру надходження запитів на їх обчислення. Розміщуйте пари за зростанням аргументу `U`, організуйте швидкий пошук за ключем методом поділу навпіл.

5.8. Підрахунок відсилок

Розглянутий у попередніх розділах спосіб керування об'єктами за допомогою маніпуляторів усунув деякі незадовільні властивості використовуваних до цього указників. Щоправда, залишилася нерозв'язаною проблема спільного доступу кількох маніпуляторів до одного й того самого об'єкта. Осібні маніпулятори стали першою спробою пошуку розв'язання проблеми, яке було досягнуто повною забороною спільного використання об'єктів. Тепер розглянемо діаметрально протилежне рішення, яке регламентує доступ маніпуляторів спілки до спільного об'єкта.

У відкладеному копіюванні, про яке йтиметься, застосовано концепцію спільного володіння даними, яка відома в літературі під назвою *підрахунок відсилок* (reference counting). Маніпулятори, які ведуть такий підрахунок, називають *указниками спільного доступу* (shareable pointer), а самі об'єкти маніпуляції – *об'єктами спільного використання*.

5.8.1. Відкладене копіювання

Розглянемо указники спільного доступу на прикладі так званих рядків із відкладеним копіюванням. Якщо звичайний рядок містить указник на ланцюжок символів –

```
// Клас рядків                                        Лістинг 5.218
class String {
private:
   size_t _len; // Довжина рядка
   char* _allocator; // Сам рядок
.........................
};
```

то в рядку з відкладеним копіюванням є особлива обгортка – його подання у вигляді структури StringRep, – виконана відповідно до вже відомої нам ідіоми прихованої реалізації.

```
// Рядки з відкладеним копіюванням на базі             Лістинг 5.219
// указників спільного доступу
class DeferredString {
private:
   struct StringRep; // Структура подання рядка
   StringRep* _rep; // Доступ до самого подання
public:
   class BadString; // Клас обробки аварійних ситуацій
   DeferredString();// Конструктор порожнього рядка
// Конструктор C++-рядка за C-рядком
   explicit DeferredString(const char*);
   DeferredString(const DeferredString& s); // Конструктор копіювання
   ~DeferredString();// Деструктор
   DeferredString& operator=(const DeferredString&); // Оператор присвоєння
   size_t length() const; // Селектор довжини рядка
   bool empty() const; // Перевірка, чи не порожній рядок
// Оператори індексування:
   char& operator[](size_t); // для модифікації символу
   const char operator[](size_t) const; // для читання символу
};
```

Визначення подання рядка приховаємо у файлі реалізації.

```
// Подання рядка за допомогою його зображення          Лістинг 5.220
struct DeferredString::StringRep {
   size_t _len; // Довжина рядка
   char* _allocator; // Сам рядок
   size_t _refCounter; // Розмір спілки
   bool _shareable; // Ознака можливості спільного використання
   StringRep(const char* s=""); // Конструктор подання
   ~StringRep();// Деструктор подання
};
```

Два перші атрибути відомі нам за визначенням звичайних рядків: це довжина рядка _len і указник на нього _allocator. Лічильник _refCounter обліковує указники спільного доступу, пов'язані з одним і тим самим рядком *_allocator. Булевий атрибут _shareable дозволяє чи забороняє спільне використання цього рядка.

Подання рядка створює особливий конструктор:

Лістинг 5.221

```
// Конструктор подання
DeferredString::StringRep::StringRep(const char* s):
      _refCounter(1), // Рядок використовується одним маніпулятором
      _shareable(true), // Рядок можна використовувати спільно
      _len(strlen(s)+1), // Довжину рядка взято з параметра
      _allocator(new char[_len]) { // Виділяємо пам'ять для рядка
   strcpy(_allocator, s); // Копіюємо рядок подання
   return;
}
```

Унаслідок того, що є додаткова структура подання рядка, у його створенні беруть участь два конструктори: конструктор його подання й один із наведених далі конструкторів власне рядка:

Лістинг 5.222

```
// Конструктор порожнього рядка
DeferredString::DeferredString():
   _rep(new StringRep("")) {} // Ініціалізація подання порожнього рядка
// Конструктор C++-рядка за C-рядком
DeferredString::DeferredString(const char* ps):
   _rep(new StringRep(ps)) {} // Ініціалізація подання рядка
```

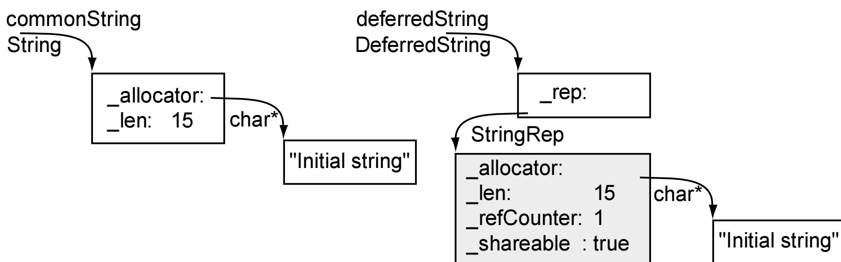


Рис. 5.37. Діаграми рядків і рядків із відкладеним копіюванням

На рис. 5.37 наведено для порівняння об'єкти commonString і deferredString, створені за визначеннями відповідно звичайного рядка та рядка з відкладеним копіюванням:

```
String commonString("Initial string");
DeferredString deferredString("Initial string");
```
Лістинг 5.223

Апріорі кожен створений рядок із відкладеним копіюванням уважають рядком спільного використання, і атрибут `_shareable` набуває значення **`true`**. Рядки спільного використання не копіюються, а користуються одним спільним поданням. Наприклад, визначимо два рядки

```
const DeferredString cstr("Constant string");
DeferredString str("Initial string");
```
Лістинг 5.224

і виконаємо присвоєння `str=cstr` (рис. 5.38). Оскільки інші маніпулятори не використовували подання рядка `str`, його буде видалено. Унаслідок присвоєння подання рядка `cstr` стає спільним, і ним користуються вже два маніпулятори (рис. 5.39). На нього можуть бути спрямовані інші маніпулятори – наприклад, після виконання копіювального конструктора `DeferredString copy1(str)` (рис. 5.40).

Наведемо текст реалізації копіювального конструктора:

Лістинг 5.225
```
// Конструктор копіювання
DeferredString::DeferredString (const DeferredString& s) {
// Перевіряємо можливість спільного використання оригіналу
   if(s._rep->_shareable) {
   // Спільне використання дозволено
      _rep=s._rep; // Приєднуємо до оригіналу приймач копіювання
      ++_rep->_refCounter; // Збільшуємо лічильник
   } else   // Спільне використання заборонено
    // Створюємо для приймача нове подання
      _rep=new StringRep(s._rep->_allocator);
   return;
}
```

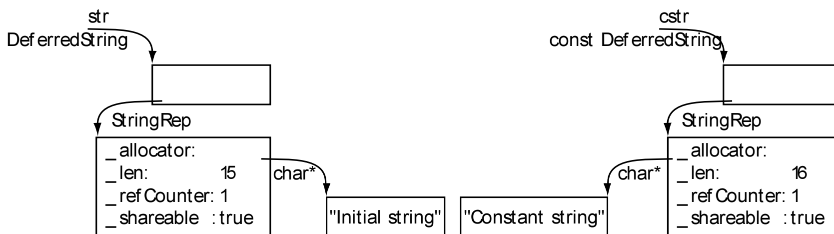


Рис. 5.38. Незалежні рядки із відкладеним копіюванням

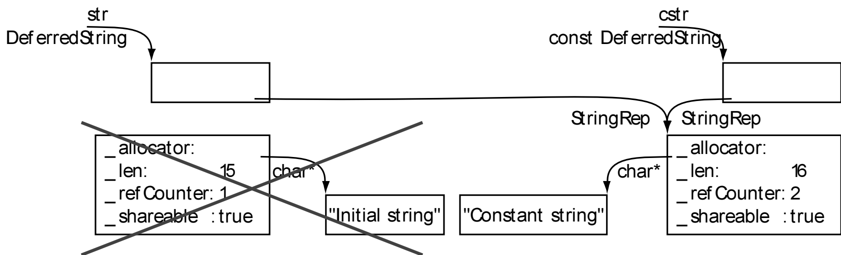


Рис. 5.39. Спільне використання рядка двома маніпуляторами

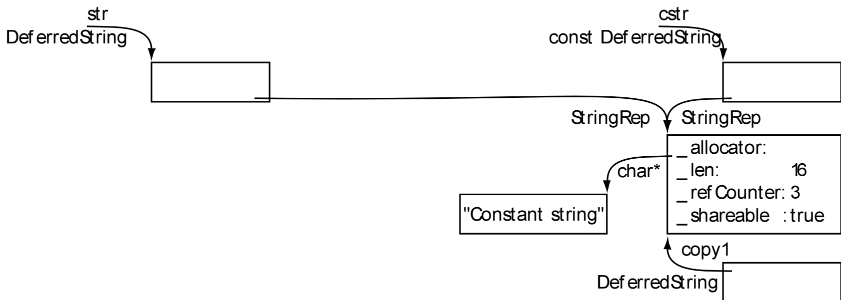


Рис. 5.40. Приєднання третього маніпулятора

Як бачимо, тут розглянуто два випадки: коли можна й не можна спільно використовувати зображення рядка. У першому випадку маніпулятор результату буде встановлено на зображення рядка спільного використання, у другому – створено новий екземпляр зображення.

Тепер присвоєння одному з маніпуляторів іншого значення зумовлюватиме не видалення зображення рядка, а лише його вилучення з корпорації маніпуляторів спільного доступу до цього зображення рядка, що зменшить лічильник посилань на одиницю.

Присвоєння, як завжди, складніше, оскільки воно виконує ще й обробку старого значення маніпулятора. Лічильник посилань при цьому зменшується на одиницю. Якщо маніпулятор використовував об'єкт одноосібно, його зображення рядка буде видалено. Знову ж таки, якщо права частина присвоєння дозволяє спільно використовувати зображення рядка, то ліва частина приєднується до нього, а ні – то з'являється нове зображення рядка.

Лістинг 5.226

```
// Оператор присвоєння
DeferredString& DeferredString:: operator=(const DeferredString& s) {
```

```
   if(this==&s) return *this;
// Підготовка приймача присвоєння: установлюємо кількість маніпуляторів
// спільного використання рядка лівої частини, попередньо зменшивши
// кількість користувачів
   if(--_rep->_refCounter==0)
      delete _rep; // Якщо користувачів не залишилося, видаляємо рядок
// Обробка джерела присвоєння: перевіряємо можливість спільного
// використання оригіналу
   if(s._rep->_shareable) {
   // Спільне використання дозволено: приєднуємо до оригіналу
   // приймач присвоєння
      _rep=s._rep; ++_rep->_refCounter;
   }
   else
   // Спільне використання заборонено. Створюємо для приймача
   // нове зображення рядка
      _rep=new StringRep(s._rep->_allocator);
   return *this;
}
```

Тепер залишилося з'ясувати причини, які можуть перешкоджати спільно використовувати подання. Головна з них – це звертання до оператора індексування, наприклад `str[1]`. Кожне таке звертання вже підозріле, адже сама операція індексування не знає, для чого її викликали. Це може бути спроба щось записати до рядка за допомогою його модифікатора, наприклад `str[1]='1'`, або передавання символу за допомогою відсилки до функції, готової замінити його, наприклад **`void`** `swap(`**`char&, char&)`**. Якщо досі селектор і модифікатор символу різнилися лише сталістю, відзначеною в сигнатурі, але мали однакові коди, то коди селектора та модифікатора рядків із відкладеним копіювання суттєво різняться. Селектор залишається таким самим:

Лістинг 5.227

```
// Оператор індексування для читання символу
const char DeferredString::operator[](const size_t i) const {
   if((i<0)||(_rep->_len<=i)) throw BadString("Bad index", i);
   return _rep->_allocator[i];
}
```

Кожна ж спроба викликати модифікатор зумовлює превентивні заходи, щоб захистити спілку маніпуляторів від модифікації спільно використовуваного рядка одним із її членів. Такі заходи за своєю суттю застережні й застосовуються завжди, навіть якщо виявляться надмірними. Рядок,

до якого спробували звернутися за допомогою модифікатора, надалі буде оголошено непридатним для спільного використання. Якщо досі маніпулятор використовував рядок осібно, то достатньо просто скасувати можливість його «усуспільнення» в майбутньому. Якщо ж поданням уже користувалися спільно, то його потрібно вилучити зі спільного застосування та заборонити приєднуватися до спілок у майбутньому. Жорсткість заходів залежить від небезпеки модифікації символу будь-ким, хто одержав доступ до місця його збереження. Модифікація може бути як завгодно надовго відкладеною, а тому спільне використання підозрілого рядка буде скасовано назавжди.

Порівняйте модифікатор для рядка з відкладеним копіюванням із модифікатором звичайного рядка, наприклад із підрозділу 4.8 (лістинг 4.162):

Лістинг 5.228

```
// Оператор індексування для модифікації символу
char& DeferredString::operator[](const size_t i) {
   if((i<0)||(_rep->_len<=i)) throw BadString("Bad index", i);
// Перевіряємо наявність спільного використання
   if(_rep->_refCounter>1) {
// Рядок використовують спільно: вилучаємо його зі спільного використання
      --_rep->_refCounter;
      _rep=new StringRep(_rep->_allocator); // Створюємо осібну копію
   }
   _rep->_shareable=false; // Закриваємо спільний доступ
   return _rep->_allocator[i];
}
```

Певна річ, у превентивності запропонованого рішення є певна надлишковість, тому що не кожен виклик маніпулятора зумовлює фактичну зміну подання. Проте, з іншого боку, немає простих засобів простежити за індексованою змінною, переданою до функції як відсилка. Але ми ще повернемося до цього пізніше, прийнявши запропоноване рішення лише як перше наближення.

Продовжимо далі розглядати приклад із рис. 5.40. Проаналізуємо реакцію класу на спробу вжити індексований вираз, наприклад `str[1]`. Одразу зазначимо, що вираз `cstr[1]` із цього погляду абсолютно безпечний (поясніть, чому) і не потребує жодних захисних заходів. Запропоноване перше наближення до повного розв'язання проблеми захисту спілки від спотворення спільно використовуваного рядка полягає у вилученні подання `str` зі спільного використання (рис. 5.41).

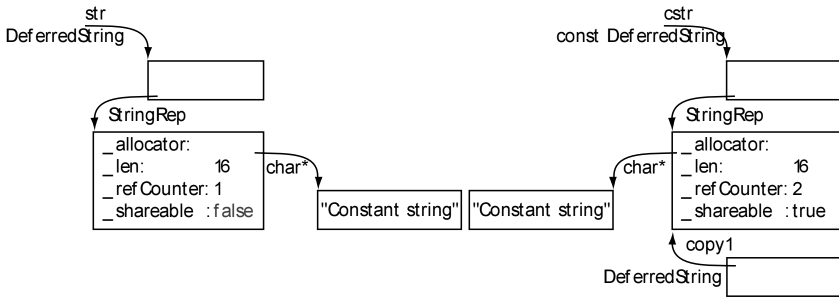


Рис. 5.41. Вилучення рядка із спільного використання

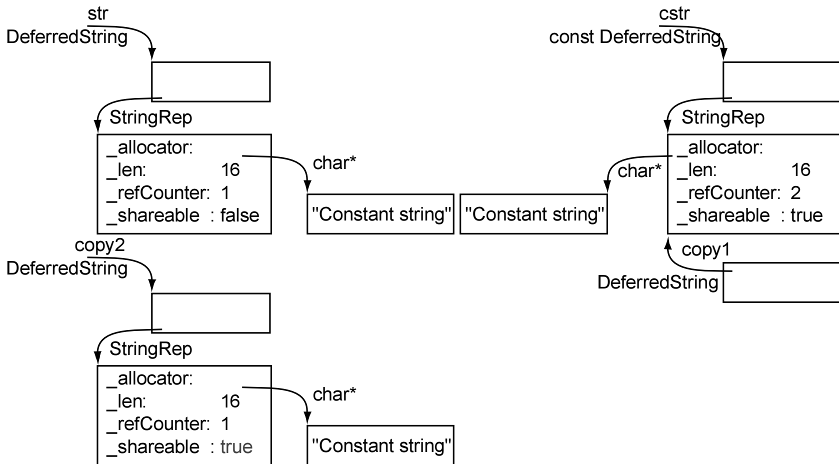


Рис. 5.42. Створення зайвої копії рядка

Таке вилучення не може бути переглянуте, і воно залишається чинним на весь час існування рядка `str`. Якщо ми копіюватимемо значення маніпулятора, наприклад, командою `DeferredString copy2(str)`, то приймач копіювання – маніпулятор `copy2` – одержить власне подання. Звісно, на копію не поширюється ненадійність оригіналу щодо його можливих спотворень, а тому копію оголошують придатною для спільного використання (рис. 5.42).

Запропоновані превентивні заходи не викликають особливого захвату, бо в розглянутому прикладі аж три копії рядка, призначеного для спільного використання. Якщо маніпулятор `str` здався підозрілим і тому його

було вилучено зі спілки, то новостворену копію цілком можна було б приєднати до старої спілки, створеної маніпулятором `cstr`. Тут іще є над чим попрацювати.

Відповідно уточнимо деструктор. Фактично подання тепер видаляється тоді, коли інші маніпулятори його не використовують.

Лістинг 5.229

```
// Деструктор
DeferredString::~DeferredString() {
// Установлюємо кількість маніпуляторів спільного використання рядка,
// попередньо зменшивши кількість користувачів
   if(--_rep->_refCounter==0)
      delete _rep; // Якщо їх не залишилося, видаляємо подання рядка
   return;
}
```

5.8.2. Метод проксі

Тепер удосконалимо захисні заходи, щоб якийсь член спілки не міг спотворити спільно використовуваний рядок. Надмірність захисту випливала з неспроможності модифікатора символу відрізнити свій виклик для читання від виклику для запису. Більш удалий варіант будують за допомогою конструкції, відомої під назвою *посередника*, чи *проксі* (proxy).

Лістинг 5.230

```
// Рядки з відкладеним копіюванням, доповнені
// проксі символів
class DeferredString {
private:
   struct StringRep; // Структура подання рядка
   StringRep* _rep; // Доступ до самого подання
   class CharProxy; // Клас посередників для символу
public:
   class BadString; // Клас обробки аварійних ситуацій
   DeferredString(); // Конструктор порожнього рядка
// Конструктор C++-рядка за C-рядком
   explicit DeferredString(const char*);
   DeferredString(const DeferredString& s); // Конструктор копіювання
   ~DeferredString();// Деструктор
// Оператори присвоєння
   DeferredString& operator=(const DeferredString&);
   DeferredString& operator=(const char*);
// Група методів підтримки проксі:
   void check(const size_t i) const; // перевірка легітимності індексу
   char read(const size_t i) const; // читання символу за індексом
   void write(const size_t i, char c); // запис символу за індексом
```

```
    CharProxy operator[](const size_t i); // селектор-модифікатор символу
    char operator[](const size_t i) const; // селектор символу
    size_t length() const;              // Селектор довжини рядка
    bool empty() const;                 // Перевірка, чи порожній рядок
};
```

Як бачимо із сигнатури оператора індексування, його результатом тепер стане не відсилка до самого символу, як це було раніше, а його посередник – проксі.

Розглянемо спочатку приклад. Візьмемо маніпулятор theSwiss і створимо копію його значення в aEuropean:

Лістинг 5.231

```
DeferredString theSwiss("Niklaus Wirth, TFH Zurich");
DeferredString aEuropean(theSwiss);
```

Результат показано на рис. 5.43.

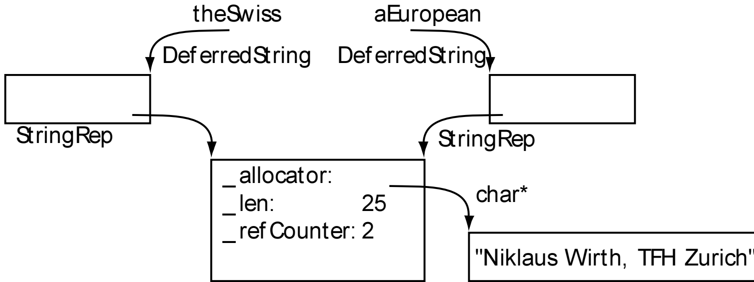


Рис. 5.43. Рядок спільного використання

Назву університету, у якому працював професор Н. Вірт, іноземці часто плутають і замість «Eidgenössische Technische Hochschule» (Державна технічна вища школа), скорочено ETH, пишуть «Technische Fachochschule» (Технічна вища школа), скорочено TFH. Звернімося до некоректного символу за допомогою маніпулятора theSwiss:

```
// Eidgenoessische, а не Technische
    theSwiss[15];
```

Оскільки рядок не сталий, то спроба звернутися до нього за індексом спричинить виклик селектора-модифікатора. Запропонований раніше метод превентивного захисту зумовив би вилучення подання рядка зі спілки. Новий же метод полягає у створенні проксі, який, коли до нього звернуться, однозначно з'ясує причину звертання й таким способом з'ясує,

що буде після виклику селектора-модифікатора, – читання чи записування (рис. 5.44).

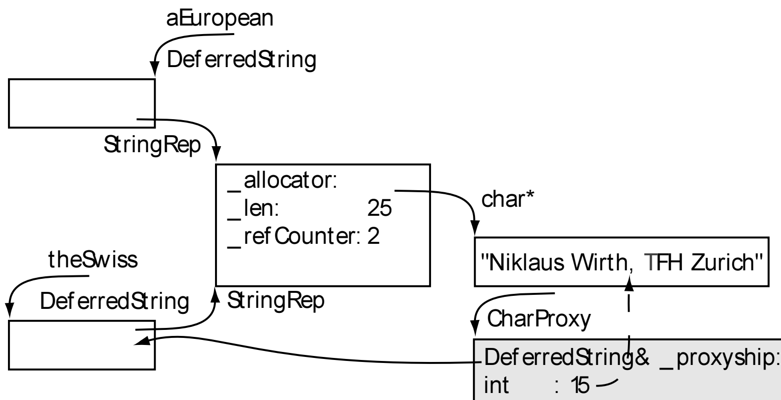


Рис. 5.44. Виникнення проксі символу

Проаналізуємо запропоноване рішення.

По-перше, ускладнюється сама структура подання рядків:

Лістинг 5.232

```
// Подання рядка від DeferredString з використанням
// проксі символів
struct DeferredString::StringRep {
   size_t _len; // Довжина рядка
   char* _allocator; // Сам рядок
   size_t _refCounter; // Розмір спілки
   StringRep(size_t, const char*); // Конструктор за символьним указником
   StringRep(size_t, const string&); // Конструктор за рядком
   ~StringRep(); // Деструктор
   StringRep* getOwnCopy(); // Псевдокопіювальний конструктор
   void assign(size_t, const char*); // Псевдоприсвоєння
private:
// Невикористовувані, а тому нереалізовані операції
   StringRep(const StringRep&); StringRep& operator=(const StringRep&);
};
```

Головне ускладнення полягає в заміні копіювального конструктора на його аналог – такий собі псевдоконструктор. Самі ж копіювальний конструктор і оператор присвоєння стануть закритими, і їх не буде реалізовано. Зазначений псевдоконструктор копіюватиме подання лише в разі його

спільного використання, повертаючи наявне подання, коли ним користуються одноосібно:

Лістинг 5.233

```
// Псевдокопіювальний конструктор
DeferredString::StringRep* DeferredString::StringRep::getOwnCopy() {
// Зображення копіюється лише тоді, коли його використовують спільно
   if(_refCounter==1)
   // Осібне використання: повертаємо оригінал
      return this;
   // Спільне використання: виходимо зі спілки
   _refCounter--;
// Створюємо копію
   return new StringRep(_len, _allocator);
}
```

У маніпуляторі явно виділяється функція, що відповідає за вихід поза рядок:

Лістинг 5.234

```
// Перевірка легітимності індексу
void DeferredString::check(const size_t i) const {
   if(_rep->_len<=i) throw BadString("Bad Index", i);
   return;
}
```

Змінюється й селектор:

Лістинг 5.235

```
// Селектор символу
char DeferredString::operator[](const size_t i) const {
   check(i);
   return _rep->_allocator[i]; // Селектор повертає сам символ
}
```

Модифікатор же набуває зовсім нового змісту: його функція полягає тепер у створенні замінника символу:

Лістинг 5.236

```
// Селектор-модифікатор символу
DeferredString::CharProxy DeferredString::operator[](const size_t i) {
   check(i);
   return CharProxy(*this, i); // Замість символу селектор повертає проксі
}
```

Розглянемо тепер будову закритого класу проксі:

Лістинг 5.237

```
// Клас представників символу
class DeferredString::CharProxy {
```

```
   friend class DeferredString;   // Надає особливі права рядку
private:
// Відсилка до рядка, символ якого подає проксі
   DeferredString& _proxyship;
   size_t _index;   // Індекс символу, який подає проксі
// Закритий від сторонніх конструктор
   CharProxy(DeferredString& s, const size_t  i);
public:
   operator char() const;   // Конвертор проксі в символ
   CharProxy& operator=(char c);   // Присвоєння символу проксі
};
```

Передусім зазначимо, що він дає маніпулятору виняткові права доступу до своїх атрибутів: посилання на маніпулятор `_proxyship` та індексу символу `_index`, – які подають цей проксі-об'єкт. Зверніть увагу на закритість конструктора проксі: ніхто, крім маніпулятора, не зможе створювати проксі-об'єкти.

Набір методів дуже простий: проксі можна перетворити на символ:

Лістинг 5.238

```
//Перетворювач проксі на символ
DeferredString::CharProxy::operator char() const {
// Читаємо символ, зазначений за допомогою проксі
   return _proxyship.read(_index);
}
```

і присвоїти йому символ:

Лістинг 5.239

```
// Присвоєння символу проксі
DeferredString::CharProxy& DeferredString::CharProxy::operator=(char c) {
// Тут, власне, відбувається записування символу на місце,
   _proxyship.write(_index, c);   // зазначене за допомогою проксі
   return *this;
}
```

Обидві операції виконують спеціальні методи `read` і `write`, додані до класу маніпулятора. Перший із них просто повертає символ, другий викликає псевдоконструктор копіювання подання рядка, тим самим вилучаючи його зі спілки, якщо воно їй належало, а потім замінює символ:

Лістинг 5.240

```
// Читання символу за індексом
char DeferredString::read(const size_t i) const
   {return _rep->_allocator[i];}
// Записування символу за індексом
```

```
void DeferredString::write(const size_t i, char c) {
   _rep=_rep->getOwnCopy(); // Вилучаємо подання рядка зі спілки
   _rep->_allocator[i]=c; // Замінюємо i-й символ рядка
   return;
}
```

Тепер присвоєння `theSwiss[15]='E'` з методу класу рядків перетворюється на метод класу проксі, а подання вилучається зі спілки саме в процесі виконання присвоєння, а не просто внаслідок звертання до модифікатора (рис. 5.45).

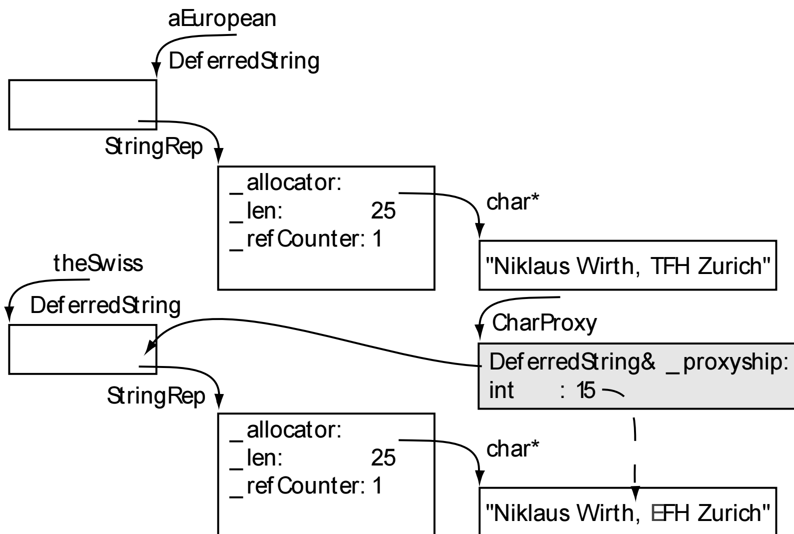


Рис. 5.45. Заміна символу через його проксі

Оскільки тут не вжито атрибут спільного використання `_shareable`, трохи спростяться копіювальні конструктор і оператор присвоєння:

Лістинг 5.241

```
// Конструктор копіювання
DeferredString::DeferredString(const DeferredString& ds) {
   ds._rep->_refCounter++; // Збільшуємо лічильник оригіналу копіювання
   _rep=ds._rep; // Приєднуємо копію до спілки
   return;
}
```

В операторі присвоєння є одне по-справжньому складне місце: це особливим способом запрограмований захист від присвоєння самому со-

бі. Уважно проаналізуйте код, щоб зрозуміти, як цей захист закодовано в місці перевірки кількості використань:

Лістинг 5.242

```
// Оператор присвоєння
DeferredString& DeferredString::operator=(const DeferredString& ds) {
   ds._rep->_refCounter++; // Збільшуємо кількість використань
// Умову судження читають так: якщо об'єкт копіює сам себе,
// то його лічильник використань більший за 1, бо хоч би яким
// він не був до цього, ми щойно збільшили його на одиницю
   assert((&ds==this)?_rep->_refCounter>1:true);
   if(--_rep->_refCounter==0) // Зменшуємо лічильник приймача на одиницю
// У разі присвоєння самому собі ніколи не потрапимо сюди. Видаляємо
// старе подання приймача присвоєння, яке використовували осібно
      delete _rep;
// Додаємо приймач присвоєння до спілки його джерела
   _rep=ds._rep;
   return *this;
}
```

Вправа 5.73

Реалізуйте оператор присвоєння з відкладеним копіюванням рядку іншого рядка, заданого символьним указником.

Підбиваючи підсумок аналізу другої версії маніпулятора, доходимо висновку, що технологія використання проксі полягає в заміні певного об'єкта (у розглядуваному випадку – символу) його представником в іншому класі. Проксі, як представник об'єкту, має тип, відмінний від типу оригіналу, тому їх не можна переплутати. Це дає змогу, зокрема, відкласти змінення об'єкта від моменту виклику модифікатора до виявлення факту присвоєння, запам'ятовуючи проксі символу, який може бути замінений. При цьому саме присвоєння символів (стандартна операція, недоступна нашому впливу) виконується оператором присвоєння їхнього проксі, у якому передбачено всі дії, потрібні для підтримки спільного використання подань рядків.

5.8.3. Узгодження перетворень указників і відсилок проксі

Впровадження проксі-класу дало змогу розпізнавати читання та записування символів, але створило нові проблеми. Усе, що доступно для символів, наприклад операція обчислення адреси символу в рядку `char* pc=&theSwiss[15]`, не визначено для проксі. Тому доведеться або відмовитися від цих засобів, або визначати кожен із них окремо для проксі. Вибравши друге, одержимо наступну, третю версію маніпулятора:

```
// Рядки з відкладеним копіюванням, доповнені                Лістинг 5.243
// проксі символів, збагаченими поведінкою символу
class DeferredString {
private:
   struct StringRep; // Структура подання рядка
   StringRep* _rep; // Доступ до самого подання
public:
   class CharProxy; // Клас посередників для символу
   class BadString; // Клас обробки аварійних ситуацій
   DeferredString();// Конструктор порожнього рядка
// Конструктор С++-рядка за С-рядком
   explicit DeferredString(const char*);
   DeferredString(const DeferredString& s); // Конструктор копіювання
   ~DeferredString();// Деструктор
// Оператори присвоєння
   DeferredString& operator=(const DeferredString&);
   DeferredString& operator=(const char*);
// Група методів підтримки проксі:
   void check(const size_t i) const; // Перевірка легітимності індексу
   char read(const size_t i) const; // Читання символу за індексом
// Записування символу за індексом
   void write(const size_t i, char c);
   CharProxy operator[](const size_t i); //Селектор-модифікатор символу
   char operator[](const size_t i) const; // Селектор символу
   size_t length() const; // Селектор довжини рядка
   bool empty() const; // Перевірка, чи не порожній рядок
};
```

Сам маніпулятор тут майже не змінився, лише став іншим рівень доступу до проксі, й у структурі подань рядків знову з'явиться атрибут можливості спільного використання `_shareable`. Клас `CharProxy` перенесено до відкритої частини класу, щоб його можна було використовувати за межами класу маніпулятора.

До класу проксі додано нові методи й одну утиліту:

```
// Клас представників символу                                Лістинг 5.244
class DeferredString::CharProxy {
   friend class DeferredString; // надає особливі права
private:
// Відсилка до рядка, символ якого подає проксі
   DeferredString& _proxyship;
   size_t _index; // Індекс символу, який подає проксі
// Закритий від сторонніх конструктор
   CharProxy(DeferredString& s, const size_t  i);
```

```
public:
// Додаткові конвертори типів:
   const char* operator&() const; // конвертор до указника на сталий символ
   char* operator&(); // конвертор до указника на символ
   operator char& (); // конвертор до псевдоніма символу
   operator char() const;
   CharProxy& operator=(char c); // Присвоєння символу проксі
};
```

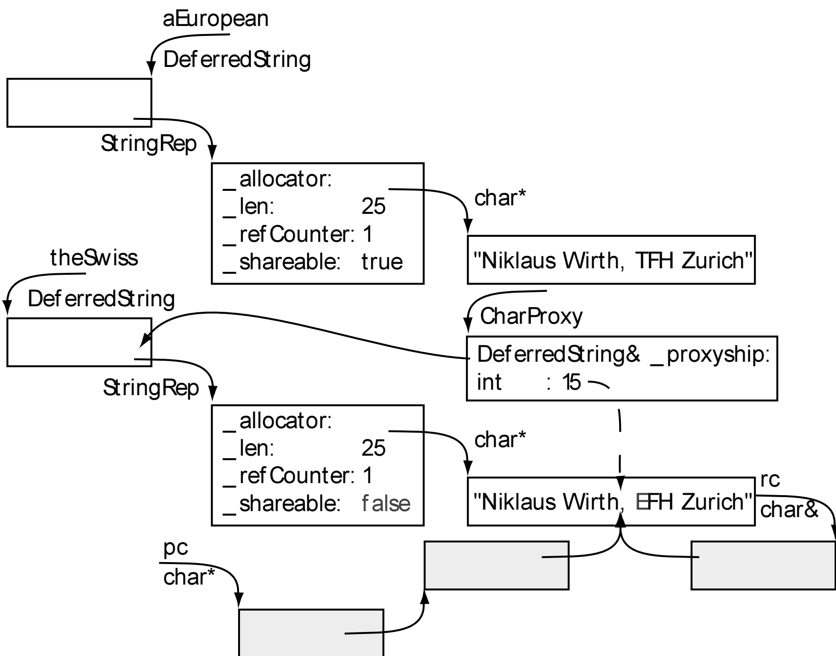


Рис. 5.46. Перетворення типів указників і відсилок проксі

Тепер можна указник на символ установити на його проксі:

```
char* pc=&theSwiss[15]; *pc='E';
```

Лістинг 5.245

або визначити його псевдонім символьного типу (рис. 5.46):

```
char& rc=theSwiss[15]; rc='E';
```

Лістинг 5.246

Це дасть змогу передати проксі як параметр до функції, призначеної для одержання за допомогою параметра указника чи псевдоніма символу:

```
const char f(char&);                                          Лістинг 5.247
f(theSwiss[15]); // Передаємо проксі як відсилку до символу
```

Дещо ускладниться клас зображень рядків:

```
// Подання рядка за допомогою StringRep із                    Лістинг 5.248
// використанням проксі символів, збагачених перетвореннями типів
struct DeferredString::StringRep {
   size_t _len; // Довжина рядка
   char* _allocator; // Сам рядок
   size_t _refCounter; // Розмір спілки
// DEJAVU: знову виникла ознака можливості спільного використання
   bool _shareable;
   StringRep(size_t, const char*); // Конструктор за символьним указником
   StringRep(size_t, const string&); // Конструктор за рядком
   ~StringRep(); // Деструктор
   StringRep* getOwnCopy();// Псевдокопіювальний конструктор
   void assign(size_t, const char*); // Пседоприсвоєння
// Дозволити спільне використання
   void makeShareable() {_shareable=true;}
// Заборонити спільне використання
   void makeUnshareable() {_shareable=false;}
// Перевірити наявність спільного використання
   bool isShareable() const {return _shareable;}
private:
// Невикористовувані, а тому нереалізовані операції
   StringRep(const StringRep&); StringRep& operator=(const StringRep&);
};
```

Наведемо реалізації нових методів класу проксі:

```
// Конвертор до указника на сталий символ                     Лістинг 5.249
const char* DeferredString::CharProxy:: operator&() const {
// Символ залишиться сталим, захист не потрібен:
// видаємо адресу символу, поданого своїм проксі
   return &_proxyship._rep->_allocator[_index];
}
// Конвертор до указника на символ
char* DeferredString::CharProxy::operator&() {
// Перевіряємо допустимість і наявність спільного використання
   if(_proxyship._rep->isShareable() && _proxyship._rep->_refCounter>1) {
// Спільне використання допустиме та наявне: вилучаємо подання зі спілки
      _proxyship._rep->_refCounter--;
// створюємо нове подання
```

```
      _proxyship._rep=new StringRep(_proxyship._rep->_len,
         _proxyship._rep->_allocator);
   }
// Оголошуємо подання непридатним для спільного використання
   _proxyship._rep->makeUnshareable();
// Видаємо адресу символу, відкритого для модифікацій
   return &_proxyship._rep->_allocator[_index];
}
// Конвертор до псевдоніма символу
DeferredString::CharProxy::operator char&() {
// Перевіряємо допустимість і наявність спільного використання
   if(_proxyship._rep->isShareable() && _proxyship._rep->_refCounter>1) {
// Спільне використання допустиме та наявне: вилучаємо подання зі спілки
      _proxyship._rep->_refCounter--;
// створюємо нове подання
      _proxyship._rep=new StringRep(_proxyship._rep->_len,
         _proxyship._rep->_allocator);
   }
// Оголошуємо подання непридатним для спільного використання
   _proxyship._rep->makeUnshareable();
// Видаємо відсилку до символу, відкритого для модифікацій
   return _proxyship._rep->_allocator[_index];
}
```

Оскільки в кожному з двох останніх операторів відкрито необмежений доступ до символу, зображення рядків стануть недоступними для спільного використання назавжди, бо ніхто не може наперед сказати, як довго указник або посилання будуть користуватися своїм правом змінити рядок чи кому вони делегують це право.

Знову заслуговує на увагу оператор присвоєння, права частина якого – символьний указник.

Вправа 5.74

Реалізуйте метод

```
DeferredString& DeferredString::operator= (const char*);
```

використовуючи псевдоприсвоєння зі `StringRep`, яке замінює подання приймача присвоєння на значення рядка за місцем його збереження, причому джерело присвоєння передано за допомогою символьного указника:

Лістинг 5.250

```
// Пседоприсвоєння
void DeferredString::StringRep::assign
                              (const size_t len, const char* ps) {
```

```
   if(ps==0) throw BadString("Attempt to use not defined pointer");
   if(_len!=len) {
      delete [] _allocator; _len=len; _allocator=new char[_len+1];
   }
   strcpy(_allocator, ps);
   return;
}
```

Розглянуті приклади, звісно, не вичерпують усього спектра застосувань об'єктів спільного використання, зокрема класів проксі. Проте для кращого розуміння всіх можливостей варто продовжити роботу в рамках парадигми узагальненого програмування, до вивчення якої ми переходимо в наступному розділі.

Завдання до розділу 5

1. 32-розрядне машинне слово складається з 4 байтів. Запрограмуйте функції перестановки байтів а) у зворотному порядку; б) в порядку зростання їхніх значень. Коректність функцій перевірте судженням.
2. Визначте клас `Word` машинних слів, що складаються з чотирьох байтів.
   a. Визначте і реалізуйте операції порозрядного заперечення, порозрядної кон'юнкції і диз'юнкції.
   b. Визначте калькулятор байтів `CalcWord`, який виконуватиме на замовлення довільну байтову операцію.
3. Клас машинних слів `Word` допускає різні реалізації, наприклад у вигляді класу `Word16` слів з двох байтів; класу `Word32` – з чотирьох байтів і, відповідно, двох півслів; класу подвійних слів `Word64` – з восьми байтів, двох слів і чотирьох півслів. Запропонуйте ієрархію цих класів і реалізуйте селектори байта та півслова.
4. Раціональне число має знак, чисельник і знаменник. Визначте клас `Ratio` раціональних чисел.
   a. Визначте операції введення і виведення для класу `Ratio`.
   b. Визначте арифметичні операції над раціональними числами. Дослідіть можливості використання виразів змішаних типів: раціональних і цілих.
   c. У класі раціональних чисел `Ratio` визначте зведення до дійсного типу.
   d. Визначте клас `RatioCalc` калькуляторів раціональних чисел із цілочисельними регістрами.
   e. Доповніть клас `Ratio` операцією наближеного множення (результат множення двох раціональних чисел – дійсне число).

5. Визначте клас VRatioFixed векторів раціональних чисел заданої розмірності. Визначте операції додавання і множення.
6. Визначте клас VRatio векторів раціональних чисел довільної розмірності. Визначте операцію додавання і множення для класу VRatio. Дослідіть можливості використання виразів змішаних типів, складених з векторів раціональних чисел заданої та довільної розмірностей.
7. Для векторів раціональних чисел визначте операції інкременту: почленного збільшення на 1 кожної з його координат.
8. Визначте клас Residue остач від ділення за модулем *m*.
   a. Визначте операції введення і виведення для класу Residue.
   b. Визначте арифметичні операції модулярного додавання, віднімання і множення. Дослідіть можливості використання значень змішаних типів: остач від ділення й цілих чисел.
   c. Визначте клас модулярних калькуляторів ResidueCalc.
   d. Визначте параметризований клас калькуляторів загорнутих чисел та його часткову спеціалізацію для модулярних калькуляторів.
9. Визначте клас FixBitString бітових рядків фіксованого розміру. Реалізуйте операції порозрядного заперечення, порозрядної кон'юнкції та диз'юнкції.
10. Визначте бітовий калькулятор CalcFixBit, який виконуватиме на замовлення довільну бітову операцію. Визначте клас днів тижня Week, наділений префіксними і постфіксними операціями обчислення наступного й попереднього дня тижня.
11. Визначте клас дат календаря Calendar, наділіть його обчисленням за датою дня тижня і порядкового номеру дня в році.
12. Визначте операції збільшення та зменшення дати на задану кількість днів, тижнів або років.
13. Створіть калькулятор дат для виконання обчислень над датами.
14. Визначте клас DeqList двосторонніх черг на базі списків.
15. Визначте клас DeqArray двосторонніх черг на базі масивів.
16. Опишіть абстрактний клас інтерфейсу черг Queue для обох типів черг.
17. Включіть до інтерфейсу розробленого в попередньому завданні класу і реалізуйте в кожному з класів функцію інвертування черги.
18. Визначте ітератори черг. Використовуючи ітератори, визначте й реалізуйте віртуальну операцію виведення вмісту черги.
19. Визначте функцію підглядання за чергою. Застосуйте її для пошуку потрібного елемента в черзі.

Навчальні проекти

1. Система електронного навчання `eL` складається з класів адміністратора, викладачів, студентів, предметів. Адміністратор – це клас одного об'єкту. Він створює новий предмет і призначає йому викладача. Викладач записує студентів на свій предмет. Кожен викладач має право вести кілька предметів, і кожен студент може записатися на кілька предметів. Визначте, реалізуйте та виконайте випробування системи `eL`. Продумайте та обґрунтуйте систему прав доступу до кожного з класів.
2. Рейтингова система електронного навчання `eLR` додатково до можливостей системи `eL` забезпечує підрахунок кредитів кожного студента та веде сумарний облік оцінок студентів за предметами протягом семестру. Визначте, реалізуйте та виконайте випробування системи `eLR`.
3. Система геометричних операцій на площині `Planimetria` оперує точками, відрізками і трикутниками. Визначте та реалізуйте засоби побудови трикутників на площині на вимогу замовника один із допустимих способів: за трьома вершинами, за стороною і двома прилеглими кутами, двома сторонами і кутом між ними, сторонами і висотами, медіанами, бісектрисами тощо. Ведіть облік об'єктів. Передбачте знаходження всіх елементів трикутника. Запрограмуйте перетворення трикутників: центральну симетрію, осьову симетрію, гомотетію.
4. Система алгебри комплексних чисел дає користувачеві змогу визначати комплексні числа в алгебричній і тригонометричній формі, створювати агрегати (вектори) різнотипних комплексних чисел, виконувати над двома довільними агрегатами покомпонентно довільні арифметичні операції та порівняння, виводити результати обчислень. Визначте, реалізуйте та виконайте випробування системи алгебри комплексних чисел.
5. Система планування часу дає користувачеві змогу визначати події, що вимагатимуть реагування. Подія задається іменем, описом, датою і часом її настання. Планувальник подає сигнал про настання події, на який користувач повинен відреагувати. В разі відсутності реакції сигнал повторюється тричі через певні проміжки часу. Визначте, реалізуйте та виконайте випробування планувальника.
6. Навчальний словник курсу англійської мови складається зі слів і виразів, впорядкованих за темами, а всередині теми – за алфавітом. Кожна тема має назву та номер. Визначте, реалізуйте та виконайте випробування класу словників, передбачивши засоби додавання нової теми, нового слова чи виразу до теми, уникаючи повторення слів.

6. Узагальнене програмування

Типізація, безсумнівно, належить до найважливіших інструментів сучасного програмування, оскільки являє собою надійний механізм захисту від використання програмних сутностей не за призначенням. Суть об'єктного програмування, викладеного в попередніх двох розділах, полягає у створенні програмованих типів даних з особливими властивостями. Якщо взяти кілька типів контейнерів (наприклад, стек, чергу або двобічну чергу чи послідовність), то до кожного з них можна застосувати операцію додавання елемента, але вона виконуватиметься відповідно до властивостей програмованого типу даних.

Однак типізація має й зворотний бік, який полягає в зростанні затрат на виготовлення чи радше дублювання великої кількості однотипних кодів, призначених кожен для свого типу даних. З'являються класи стека символів, цілих, дійсних або комплексних чисел, точок площини, відрізків тощо. Кожен із цих класів вирізняється типом об'єктів, але має такі самі властивості й поведінку, як інші класи. Ми вже стикалися з подібною ситуацією, коли розглядали узагальнені функції в підрозділі 3.6.1. Тепер настала черга узагальнених або *параметризованих класів* (parametric class), які будуватимемо за одним зразком із так званого *шаблону класу* (class template) за допомогою підстановки конкретних типів замість *ти́пових параметрів* (type parameter).

Параметризовані типи дають змогу сформулювати й розв'язати ще одну задачу типізації, а саме дати відповідь на запитання про роль нети́пових (наприклад, числових) параметрів класів. Узагальнені функції теж можуть бути параметризованими з використанням нети́пових параметрів, хоча частіше параметр приєднується до формальних параметрів. Ми далі коротко зупинимося на відмінностях кожного зі способів параметризації функцій. Якщо ж розглядати нети́пові параметри класу, можливі два шляхи. Візьмемо для прикладу два типи векторів: `Vector5` із п'яти й вектор `Vector1000` з тисячі дійсних чисел. Кожен з них повинен визначатися власним класом, а саме `class Vector5` і `class Vector1000`, а тому об'єкти кожного належать до власного типу. З іншого боку, можна визначити один тип `Vector`, кількість елементів якого буде параметром його конструктора. Тоді обидва вектори, відповідно з п'яти `Vector(5)` і тисячі `Vector(1000)` елементів – об'єкти одного й того самого типу. Якому рішенню віддати перевагу? Параметр у конструкторі дає можливість без додаткових затрат побудувати вектор якої завгодно розмірності. Перший спосіб визначення векторів `Vector5` і `Vector1000` вимагає ство-

рення власного класу для кожної розмірності, що приводить до дублювання як вхідного, так і об'єктного коду. Утім дуже легко уявити собі ситуацію, за якої бажано розрізняти типи векторів за їх розмірностями, – скажімо, виконуючи почленне додавання чи обчислюючи скалярний добуток двох векторів. У такому разі розмірність повинна бути параметром типу, а не конструктора.

У цьому розділі ми розпочнемо вивчати класи, параметризовані ти́повими й не тільки ти́повими параметрами.

6.1. Типи, що залежать від сталих параметрів

Використання класу як типу даних надає унікальні можливості для розроблення об'єктних програм. Базована на цьому типізація дає змогу наділити кожен клас властивими йому методами, і компілятор може перенести багато перевірок з етапу виконання програми на етап компіляції.

Пригадаймо класи векторів. Візьмемо для прикладу клас `WrappedVector` із лістингу 4.67.

Лістинг 6.1

```
// Клас загорнутих векторів
class WrappedVector {
private:
   static const size_t _n; // Розмірність вектора
   double* _v; // Власне вектор
public:
   class BadVector;
   WrappedVector();
   WrappedVector(const WrappedVector&);
   ~WrappedVector();
   size_t size() const {return _n;} // Селектор розмірності
   WrappedVector& operator=(const WrappedVector&); // Оператор копіювання
// Оператори індексування:
   double operator[](const size_t) const; //для читання елемента масиву
   double& operator[](const size_t); // для модифікації елемента масиву
};
```

Цей тип даних чудово підходить для відображення математичного поняття вектора заданої розмірності, яку визначено в класі статичним атрибутом `_n`. Задавши потрібне його значення, одержуємо вектори потрібної розмірності. Щоправда, у його визначенні є певна недовершеність. Наприклад, оператор

```
const size_t WrappedVector::_n=100;
```

має міститись у файлі реалізації класу, а тому може виявитися недоступним для коригування. Справді, у підрозділі 2.10.1 ми домовилися про те, що оголошення розміщуємо в заголовних файлах, а визначення – у файлах реалізації. Клієнтові класу, звісно, обов'язково надають заголовний файл класу, а його реалізацію, швидше за все, буде надано у відкомпільованому вигляді. Можна було б також вилучити визначення параметра з файлу реалізації класу та покласти цілковиту відповідальність за нього на клієнта. Однак тоді виникнуть суперечності між різними клієнтами одного й того самого класу в рамках однієї програми. До того ж незрозуміло, як всередині однієї програми використати одночасно вектори різних розмірностей.

Інше визначення вектора дає змогу надати кожному об'єктові цього типу власну розмірність:

Лістинг 6.2

```
// Клас векторів різної розмірності
class DissimilarVector {
private:
   const size_t _n; // Розмірність вектора
   double* _v; // Власне вектор
public:
// Стандартний набір із конструктора, деструктора, копіювань
   explicit DissimilarVector(const size_t);
   DissimilarVector (const DissimilarVector&);
   ~DissimilarVector();
   DissimilarVector& operator=(const DissimilarVector&);
// Селектор розмірності
   size_t size() const {return _n;}
// Оператори індексування:
   double operator[](const size_t) const;//для читання елемента вектора
   double& operator[](const size_t); //для модифікації елемента вектора
};
```

Тепер розмірність вектора буде задано динамічно під час його створення. Наприклад, визначення

```
DissimilarVector v(10), w(100);
```

зумовлює створення двох об'єктів `v` та `w` одного й того самого типу `DissimilarVector`, але з різними розмірностями. Виникає справедливе запитання: чи має бути у векторів розмірністю 10 і 100 один і той самий тип? Легко уявити собі задачі (наприклад, лінійної алгебри), у яких вектори різної розмірності трактувалися б як такі, що належать до різних типів.

Узагальнене програмування надає можливість підпорядкувати визначення типу об'єкта контексту його використання. Для цього перетворимо

розмірність на параметр класу та запишемо визначення узагальненого або параметризованого класу (залежного від числового параметра):

Лістинг 6.3

```
// Шаблон для визначення параметризованого класу
// векторів, розмірність яких задано числовим параметром
template <size_t size>
class Vector {
private:
   const size_t _n; // Розмірність вектора
   double _v[size]; // Власне вектор, що залежить від параметра класу
public:
// Стандартний набір із конструктора, деструктора, копіювань
   explicit Vector();
   Vector(const Vector& vec);
   ~Vector();
   Vector& operator=(const Vector&);
// Селектор розмірності
   size_t volume() const {return _n;}
// Оператори індексування:
   // для читання елемента вектора
   double operator[](const size_t i) const;
   // для модифікації елемента вектора
   double& operator[](const size_t i);
};
```

Тепер тип вектора залежить від сталого параметра – його розмірності. Відповідно зміниться вигляд визначення векторів: щоб створити бажаний об'єкт, потрібно спочатку визначити його клас за допомогою конкретизації шаблону параметризованого класу, задавши числове значення параметра шаблону. Це значення записують після імені параметризованого класу в кутових дужках:

Лістинг 6.4

```
// Конкретизація шаблону параметризованого класу
Vector<10> v;   // значенням параметра 10
Vector<100> w;  // Значенням параметра 100
```

Конкретні класи, одержані з узагальненого визначення, уважають різними. Це означає, що для кожного класу буде створено окремі набори кодів для всіх методів, а сам шаблон задає лише схему будови параметризованого класу. Далі ми ще обговорюватимемо проблему розміщення кодів реалізації, яка випливає з особливого способу створення об'єктного коду класу. Тут лише зазначимо, що тепер вектори `v` та `w` мають різні типи: перший – `Vector<10>`, вектор із десяти компонентів; другий –

`Vector<100>`, вектор зі ста компонентів (рис. 6.1). Отже, спроба виконати присвоєння `v=w` призведе до синтаксичної помилки.

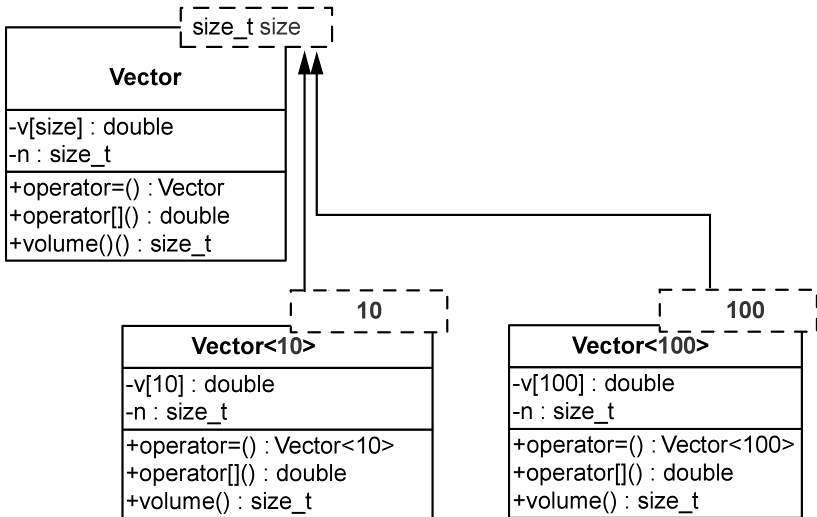


Рис. 6.1.

Параметризований клас векторів дає змогу зрозуміло та цілком коректно визначити операцію скалярного множення векторів, яку тут наведено як утиліту, реалізовану у вигляді узагальненої функції:

Лістинг 6.5

```
// Шаблон узагальненої операції скалярного множення
// параметризованих векторів
template <size_t size>
double operator* (const Vector<size>& u, const Vector<size>& v) {
   double res=0;
   for(size_t i=0; i<u.volume(); i++)
      res+=u[i]*v[i];
   return res;
}
```

Зверніть увагу на те, що сам оператор множення одержав числовий параметр, для якого, до речі не знайдеться місця у списку формальних параметрів, оскільки операція множення бінарна.

Згадаймо також (підрозділ 3.6.1), що узагальнена функція теж задає лише схему своєї будови, але не сам код. Тому її потрібно конкретизувати залежно від набутих значень параметрів узагальнення, а самі парамет-

ри мають бути відомими на етапі компіляції. Це означає, що компілятор будує об'єктний код лише на виконання конкретного виклику. Тому в контексті визначення

```
Vector<10> v1, v2; Vector<100> w1, w2;
```

Лістинг 6.6

компіляція виклику v1*v2 зумовить генерування об'єктного коду, який відповідав би тексту

Лістинг 6.7

```
// Конкретизація шаблону узагальненої операції скалярного
// множення параметризованих векторів значенням параметра size=10
double operator* (const Vector<10>& u, const Vector<10>& v) {
   double res=0;
   for(size_t i=0; i<u.volume(); i++)
      res+=u[i]*v[i];
   return res;
}
```

Компіляція ж виклику w1*w2 змусить компілятор згенерувати іншу конкретизацію операції множення –

Лістинг 6.8

```
// Конкретизація шаблону узагальненої операції скалярного
// множення параметризованих векторів значенням параметра size=100
double operator* (const Vector<100>& u, const Vector<100>& v) {
   double res=0;
   for(size_t i=0; i<u.volume(); i++)
      res+=u[i]*v[i];
   return res;
}
```

Водночас спроба виконати множення v1*w2 різнотипних векторів стане синтаксично некоректною. Тому перевага таких контекстно-залежних типів – можливість визначити окремий клас із власним набором методів або утиліт для кожного значення розмірності. Однак за певних обставин ця перевага може стати вадою, оскільки зростає обсяг коду: адже кожне використання нового значення параметра спричиняє генерування нової реалізації з подальшою її компіляцією.

Тепер стає зрозумілим, чому розглядаючи параметричну підінтегральну функцію еліптичного інтегралу $I(a,b)=\dfrac{2}{\pi}\int\limits_a^{\frac{\pi}{2}}\dfrac{dx}{\sqrt{a^2\sin^2 x+b^2\cos^2 x}}$,

ми не зробили спроби занести a і b до параметрів шаблону, оскільки їх значення звичайно визначаються на етапі виконання, а не компіляції.

Що ж стосується методів, то параметризованим буде кожен із них, реалізацію якого записано з відсилкою на параметризований клас:

Лістинг 6.9

```
// Реалізація присвоєння в параметризованому класі
// Vector<size>
template <size_t size>
Vector<size>& Vector<size>::operator= (const Vector<size>& vec) {
   if (this==&vec) return *this;
   for (size_t i=0; i<_n; i++)
      _v[i]=vec._v[i];
   return *this;
}
```

Оскільки реалізацію параметризованих методів буде відкомпільовано лише на замовлення в разі виникнення потреби в ній, її вхідний код розміщують у заголовному файлі, щоб зробити його доступним для клієнтів класу.

Вправа 6.1

Завершіть реалізацію параметризованого класу векторів. Складіть і виконайте тестову програму для перевірки двох і більше конкретизацій.

За бажання інформацію про те, що якийсь тип згенеровано з параметризованого типу, можна приховувати за допомогою визначення типів **typedef**, наприклад:

```
typedef Vector<10> Vector10;
```

Тоді програма, що використовуватиме згенерований тип, звертатиметься до нього за допомогою звичайного ідентифікатора. Таке приховування може бути корисним у разі створення бібліотек програмних кодів, користувачі яких можуть навіть не знати про наявність узагальнених типів.

Вправа 6.2

Визначте клас векторів із заданою розмірністю, утворених із комплексних чисел в алгебричному вигляді. Передбачте в ньому конструктор, який за парою дійсних векторів підхожої розмірності, складених відповідно з дійсних і уявних частин комплексних чисел, будує вектор комплексних чисел.

Розглянемо приклад параметризованого класу, що залежить від кількох параметрів. Наведемо текст відповідного заголовного файлу повністю, щоб побачити особливості розміщення вхідного коду, які обговоримо в наступному підрозділі:

```
// Шаблон для визначення параметризованого класу            Лістинг 6.10
// текстових екранів розміром height x width
template <size_t height, size_t width>
class Screen {
// Однойменні сталі та несталі методи
   const Screen& home() const; // Перенести курсор на початок екрана
   Screen& home();
   const Screen& move() const; // Змістити курсор на один символ праворуч
   Screen& move();
   const Screen& back() const; // Змістити курсор на один символ ліворуч
   Screen& back();
   const Screen& show() const; // Показати вміст екрана
   Screen& show();
// Перенести курсор у задану позицію
   const Screen& move(const size_t, const size_t) const;
   Screen& move(const size_t, const size_t);
   const Screen& clear() const; // Очистити екран
   Screen& clear();
   const Screen& showCurrent() const; // Показати символ, перед яким
                                      // міститься курсор
   Screen& showCurrent();
   char get() const; // Селектор
   char get();
   const Screen& set(char) const; // Модифікатор
   Screen& set(char);
private:
   static const size_t maxHeight; // Максимальна висота екрана
   static const size_t maxWidth; // Максимальна ширина екрана
   static const char _filler; // Наповнювач
   size_t _height; // Фактична висота екрана
   size_t _width; // Фактична ширина екрана
   char* _wContent; // Наповнення екрана
   mutable size_t _cursor; // Місце курсора
// Нереалізовані, а тому закриті копіювання:
// копіювання екранів не передбачено
   Screen(const Screen&);
   Screen& operator=(const Screen&);
};
```

Зверніть увагу на те, що параметризовим виявляється все наповнення класу, зокрема його статичний атрибут. Для його використання поза межами класу потрібно повністю визначити шаблон параметризованого атрибута за допомогою відсилки до його параметризованого класу:

```
// Визначення узагальненого статичного атрибута          Лістинг 6.11
template <size_t height, size_t width>
const char Screen<height, width>::_filler='.';
```

Так само записуємо реалізації методів:

```
// Реалізація конструктора параметризованого екрана      Лістинг 6.12
// Екран заповнюється рядком pc, якщо його задано, а ні -
// то наповнювачем _filler
template <size_t height, size_t width> Screen<height, width>::
      Screen(char* s):
// Ініціалізація довжини та ширини екрана параметрами класу
      _width(width), _height(height),
// Створюємо образ екрана в пам'яті
      _wContent(new char [_height*_width+1]),
// Курсор установлюємо перед першим символом
      _cursor(0) {
// Указник може бути невизначеним; тоді вважаємо рядок порожнім,
// а ні - то беремо його довжину
   size_t lens=(s==0? 0 :strlen(s));
// Якщо довжина рядка наповнення надто велика, урізаємо його
   size_t len=(lens>_height*_width? _height*_width :lens);
// Наповнення екрана починаємо з кінця символом закінчення рядка
   _wContent[_height*_width]='\0'; size_t k=0;
   if(s!=0)
// Уставляємо на початку екрана переданий параметром рядок
      for(; k<len; k++)
         *(_wContent+k)=*s++;
// Доповнюємо екран до кінця наповнювачем
   for(; k<_height*_width; k++)
      *(_wContent+k)=_filler;
}
```

Реалізація інших методів майже дослівно повторює реалізацію з підрозділу 5.2 (лістинг 5.29).

```
// Реалізація деструктора                                Лістинг 6.13
template <size_t height, size_t width> Screen<height, width>::~Screen() {
   delete [] _wContent;
}
// Реалізація сталого методу «Перенести курсор на початок екрана»
template <size_t height, size_t width>
const Screen<height, width>& Screen<height, width>::home() const {
   _cursor=0;
```

```
   return *this;
}
// Реалізація методу «Перенести курсор на початок екрана»
template <size_t height, size_t width>
Screen<height, width>& Screen<height, width>::home() {
   _cursor=0;
   return *this;
}
// Реалізація сталого методу «Змістити курсор на один символ праворуч»
template <size_t height, size_t width>
const Screen<height, width>& Screen<height, width>::move() const {
   if((++_cursor)>=width*height) _cursor=0;
   return *this;
}
// Реалізація методу «Змістити курсор на один символ праворуч»
template <size_t height, size_t width>
Screen<height, width>& Screen<height, width>::move() {
   if((++_cursor)>=width*height) _cursor=0;
   return *this;
}
// Реалізація методу «Змістити курсор на один символ ліворуч»
template <size_t height, size_t width>
const Screen<height, width>& Screen<height, width>::back() const {
   if(--_cursor<0) _cursor=0;
   return *this;
}
// Реалізація методу «Змістити курсор на один символ ліворуч»
template <size_t height, size_t width>
Screen<height, width>& Screen<height, width>::back() {
   if(--_cursor<0) _cursor=0;
   return *this;
}
// Реалізація сталого методу «Показати вміст екрана»
template <size_t height, size_t width>
const Screen<height, width>& Screen<height, width>::show() const {
   size_t cursor=_cursor;
   cout<<"cursor="<<_cursor<<endl; home();
   for(size_t i=0; i<_height; i++) {
      for(size_t j=0; j<_width; j++) {
         cout<<get(); move();
      }
      cout<<endl;
   }
   _cursor=cursor;
```

```
   return *this;
}
// Реалізація методу «Показати вміст екрана»
template <size_t height, size_t width>
Screen<height, width>& Screen<height, width>::show() {
   size_t cursor=_cursor; cout<<"cursor="<<_cursor<<endl; home();
   for(size_t i=0; i<_height; i++) {
      for(size_t j=0; j<_width; j++) {
         cout<<get(); move();
      }
      cout<<endl;
   }
   _cursor=cursor;
   return *this;
}
// Реалізація сталого методу «Перенести курсор у задану позицію»
template <size_t height, size_t width>
const Screen<height, width>& Screen<height, width>::
            move(const size_t i, const size_t j) const {
   if((i>=_height)||(j>=_width)) _cursor=0;
   else _cursor=_width*i+j;
   return *this;
}
// Реалізація методу «Перенести курсор у задану позицію»
template <size_t height, size_t width>
Screen<height, width>& Screen<height, width>::move(const size_t i, const
size_t j) {
   if((i>=_height)||(j>=_width)) _cursor=0;
   else _cursor=_width*i+j;
   return *this;
}
// Реалізація методу «Очистити екран»
template <size_t height, size_t width>
Screen<height, width>& Screen<height, width>::clear() {
   for(size_t i=0; i<_height*_width; i++)
      *(_wContent+i)=_filler;
   _cursor=0;
   return *this;
}
// Реалізація методу «Показати символ, пере яким міститься курсор»
template <size_t height, size_t width>
const Screen<height, width>& Screen<height, width>::showCurrent() const {
   cout<<get()<<endl;
   return *this;
```

```
}
// Реалізація селектора
template <size_t height, size_t width>
char Screen<height, width>::get() const {
   return *(_wContent+_cursor);
}
// Реалізація модифікатора
template <size_t height, size_t width>
Screen<height, width>& Screen<height, width>::set(char a) {
   *(_wContent+_cursor)=a;
   return *this;
}
```

Вправа 6.3.

Виконайте тестування узагальненого класу `Screen` на кількох прикладах екранів із різними розмірами.

Визначення узагальненого класу може мати параметри із заданими за замовчуванням значеннями, які записують звичайним способом у визначенні класу (далі наведено лише визначення параметрів):

Лістинг 6.14

```
template <size_t height=24, size_t width=80>
class Screen;
```

Щоб скористатися замовчуванням, визначення об'єкта записують із неповним або порожнім списком параметрів:

Лістинг 6.15

```
// Обидва параметри задано за замовчуванням
Screen<> oldTypeScreen;
// Другий параметр задано за замовчуванням
Screen<30> anotherScreen;
```

Узагальнені класи можуть мати статичні атрибути, як, наприклад, наповнювач `_filler` у класі `Screen`. Кожна конкретизація узагальненого класу одержує власний екземпляр статичного атрибута. У наведеному раніше прикладі було задано одне спільне значення наповнювача для всіх конкретизацій, однак можна також визначати конкретизований статичний атрибут окремо для кожної конкретизації класу – наприклад, так (див. також рис. 6.2):

Лістинг 6.16

```
const char Screen<10, 10>::_filler='-';
const char Screen<3, 4>::_filler='+';
const char Screen<5, 8>::_filler=':';
const char Screen<>::_filler='*';
```

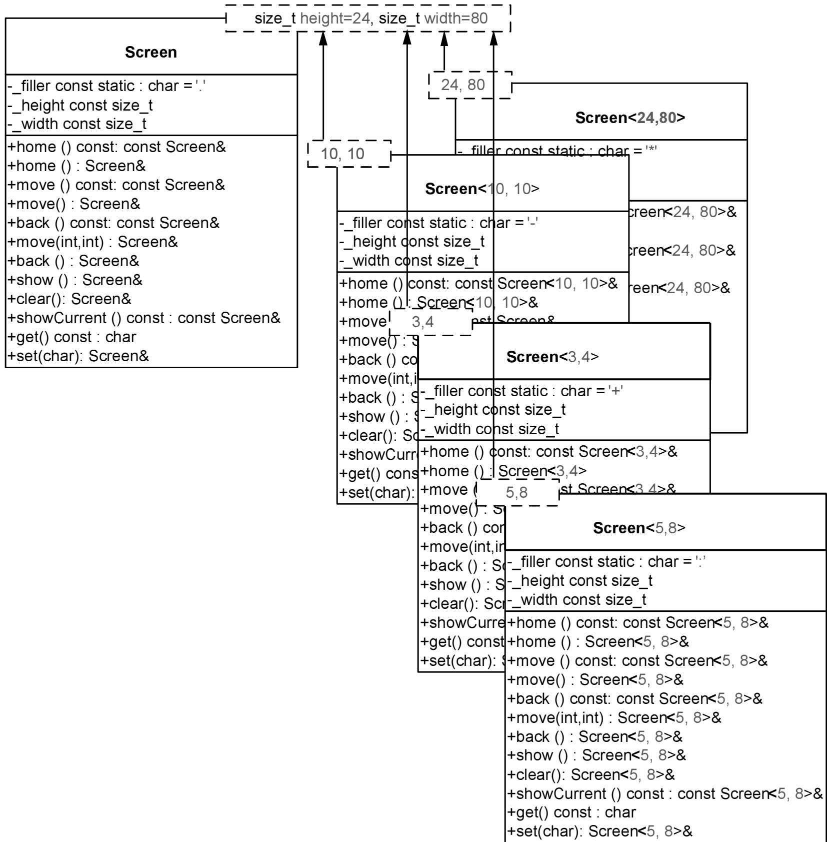


Рис. 6.2. Конкретизації шаблону `Screen`

Вправа 6.4

Складіть тестовий приклад і переконайтеся, що різні конкретизації екрана одержать різні наповнювачі. Перевірте, зокрема, наповнювачі для визначених раніше екранів `oldTypeScreen` і `anotherScreen`.

Роблячи висновки із наведених прикладів, потрібно констатувати зв'язок посиленої типізації класів із значним розбуханням коду. Дійсно, вектори різної розмірності, так само, як екрани різних розмірів наділяються кожен власним типом. Так, наприклад, вектори `Vector<10>` і `Vector<100>` суть різні типи. Те ж саме можна сказати про екрани, на-

приклад, `Screen<10,10>`, `Screen<24,80>`, `Screen<3,4>` тощо. Платою за строгу типізацію стає власний набір об'єктного коду для кожної конкретизації шаблону, саме це називається *розбуханням* коду (code bloating). Тому, враховуючи розбухання коду, з огляду на склад наявних функцій узагальнення екранів не можна вважати вдалим. В той же час узагальнення векторів виявилося виправданим, оскільки воно дало змогу розмежувати їхні типи за важливою характеристикою розмірності.

6.2. Параметризовані класи

Переходячи до загального розгляду контекстно-залежних типів, ще раз зазначимо, що ідеї *узагальненого програмування* (generic programming), застосовані до визначення функцій, можна успішно поширити на рівень класів. Проаналізуємо, як узагальнити деякі з визначених у попередніх розділах класів.

Розглянемо, наприклад, клас `Array`. У ньому можна узагальнити тип елемента масиву. Досі нам були відомі дві можливості: відразу задати масив елементів конкретного типу (наприклад, дійсних чисел) або винести визначення типу елементів масиву за його межі, скориставшись для нього символічним позначенням, конкретизованим за допомогою визначення типів **`typedef`**.

Пригадаймо визначення масиву з підрозділу 5.4.1.

Лістинг 6.17

```
// Простий масив елементів типу Elem
class Array {
public:
   class BadArray; // Вбудований клас для обробки помилкових ситуацій
// Конструктор, закритий для конвертування типів
   explicit Array(const size_t);
   ~Array(); // Деструктор
// Селектор-модифікатор елемента масиву
   Elem& operator[](const size_t index);
// Селектор елемента масиву
   const Elem& operator[](const size_t index) const;
   size_t size() const; // Розмірність масиву (селектор)
private:
   const size_t _size; // Розмірність масиву
   Elem* const _allocator; // Уміст масиву
// Операції та функції, заборонені для використання в масивах
   Array(const Array&);
   Array& operator=(const Array&);
};
ostream& operator<<(ostream& os, const Array& seq);
```

Наведене визначення масиву не можна компілювати, поки до нього не буде приєднано визначення прихованого типу `Elem`. Воно може міститися навіть в іншому файлі, приєднаному до заголовного файлу масиву. У різних застосуваннях класу `Array` тип `Elem` може відображати різні типи даних залежно від приєднаного визначення. Однак у межах однієї програми тип `Elem` може мати лише одне, а не кілька визначень. Наприклад, якщо в тексті програми з'явилось визначення типів

```
typedef int Elem;
```

то тип `Array` стає масивом із цілими елементами. Тепер щоразу, коли в програмі трапиться визначення об'єкта типу `Array`, наприклад `Array ai(3)`, його буде інтерпретовано як виклик конструктора масиву цілих чисел, хоча цього не видно з назви класу. Проте якщо в іншій програмі використано інше визначення типу елемента, наприклад **`typed?-ef double`** `Elem`, то цей самий клас уже трактуватиметься як масив дійсних чисел, і це може спричинити непорозуміння. Тому з визначення об'єкта класу `Array`, узагалі кажучи, складно однозначно відповісти на запитання про тип його елементів, оскільки формально тип `Elem` виведено з глобального контексту, і його уточнення може міститися де завгодно.

Тепер спробуємо вибрати найкращий спосіб створення масиву як контейнера, придатного для зберігання щоразу іншого типу елементів. Оскільки спосіб реалізації операцій доступу до елемента спільний для масивів будь-якого типу, можна тиражувати клас `Array` так, щоб одночасно використовувати масиви елементів різних типів. Найпростіший вихід – переписати наведене раніше визначення масиву, підставивши замість позначення `Elem` потрібний тип даних і надати кожному варіанту класу унікальне ім'я, яке б відрізняло масив одного типу від іншого. На рис. 6.3 наведено визначення масивів трьох типів, створені за допомогою команд пошуку та заміни (find/replace):

ArrayOfInt	**ArrayOfChar**	**ArrayOfPoint**
-v[size] : int -n : unsigned int	-v[size] : char -n : unsigned int	-v[size] : Point -n : unsigned int
+operator[]() : int& +operator[] () const : int +size() : size_t()	+operator[]() : char& +operator[] () const : char +size() : size_t()	+operator[]() : Point& +operator[] () const : const Point& +size() : size_t()

Рис. 6.3. Визначення трьох масивів

Як бачимо, визначення класів для кожного з типів відрізняються одне від одного лише іменами цих типів (так само, як і реалізації). Тому якщо

треба задати, скажімо, масив комплексних чисел, то можна взяти тексти для одного з масивів наявного типу елементів і замінити в них цей тип на тип комплексних чисел. Це рішення важко назвати дуже вдалим. Окрім того, що виготовлення таких копій класів уручну надто контрастує з уявленням про програмування як творчий процес розроблення програм, спроба перекласти його на автоматичний засіб пошуку та заміни може призвести до непорозумінь.

Але є серйозніша проблема. Процесу створення кожної програмної системи властиве постійне збільшення обсягів вхідних кодів. Час від часу до того чи іншого вхідного коду вносять певні правки, і тоді виникає завдання визначення їхнього впливу на програмну систему. Мінімальний вплив полягає у повторній компіляції об'єктного коду за модифікованим вхідним. Змінений код може впливати на інші частини системи: наприклад, модифікація компонента може призвести до потрібності повторної компіляції композита. Такі ситуації система програмування відстежує автоматично, аналізуючи зв'язки між файлами, визначені директивами препроцесора. Але жодна система програмування не буде відстежувати залежності між класами `ArrayOfInt`, `ArrayOfChar`, `ArrayOfPoint`, `ArrayOfComplex` тощо. Відношення, що виникають внаслідок такого зв'язку, називають спряженістю класів. Згадані класи спряжені, оскільки у них дублюється один і той самий код, а тому модифікація в одному з них, наприклад, виправлення виявлених помилок, повинна повторитися в усіх інших спряжених, але формально незалежних класах, що робить супровід таких програм невиправдано складним або навіть практично неможливим.

Набагато привабливіше і, головне, ефективніше в усіх аспектах перекласти на компілятор завдання генерування конкретних варіантів класу (у наведеному прикладі – масивів) за єдиним формально визначеним зразком. Формальність будемо розуміти так, що спряженість споріднених класів стає категорією системи програмування і одержує назву *шаблону параметризованих класів* (*class template*). Для цього потрібно змінити статус позначення типу елемента масиву `Elem`: тепер він стане ти́повим параметром. Конкретний клас утворюють із узагальненого, заданого шаблоном параметризованих класів, за допомогою підстановки деякого підхожого типу замість формального ти́пового параметру, як це робили у разі використання узагальнених функцій.

Лістинг 6.24

```
// Шаблон для визначення параметризованого класу
// масивів, тип елементів яких задано типовим параметром
template <typename Elem>
class Array {
```

```
public:
   class BadArray; // Вбудований клас для обробки помилкових ситуацій
// Конструктор, закритий для конвертування типів
   explicit Array(const size_t);
   ~Array(); // Деструктор
// Селектор-модифікатор елемента масиву
   Elem& operator[](const size_t index);
// Селектор елемента масиву
   const Elem& operator[](const size_t index) const;
   size_t size() const; // Розмірність масиву (селектор)
private:
   const size_t _size; // Розмірність масиву
   Elem* const _allocator; // Уміст масиву
// Операції та функції, заборонені для використання в масивах
   Array(const Array&);
   Array& operator=(const Array&);
};
```

Тип елемента масиву відіграє роль формального параметра, замість якого під час створення об'єктів буде підставлено фактичні типи. Відповідно змінюється формат імені конкретизованого класу. На відміну від звичайних імен класів ім'я конкретизації узагальненого класу складається з двох частин: імені узагальненого класу й імені фактичного ти́пового параметра, узятого в кутові дужки, як для числового параметра, наприклад `Array<int>`, `Array<char>`, `Array<Point>`.

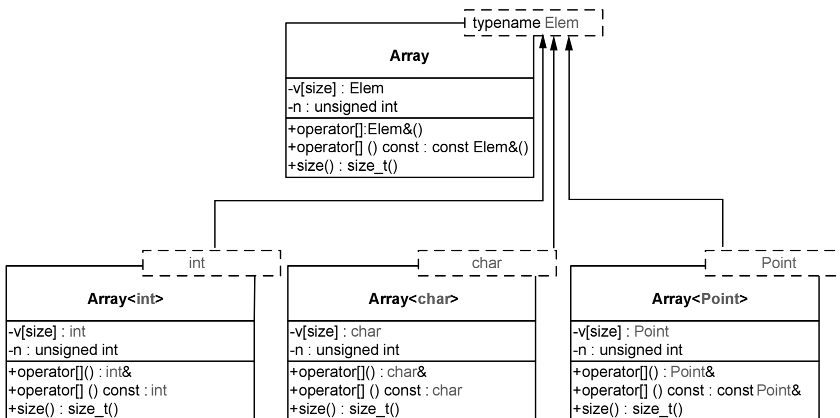


Рис. 6.4. Конкретизації шаблону масивів

Тепер зникає проблема ручного тиражування вихідних кодів, оскільки виникає можливість автоматично генерувати конкретні масиви з різними типами елементів за єдиною схемою шаблону (рис. 6.4).

Відповідно параметризованою стає утиліта виведення. Це помітно вже з її оголошення, тому що воно міститься за межами визначення класу:

Лістинг 6.25

```
// Шаблон узагальненої операції виведення
// параметризованих масивів
template <typename Elem>
   ostream& operator<<(ostream&, const Array<Elem>&);
```

Те саме стосується реалізації утиліти виведення

```
// Шаблон реалізації узагальненої операції виведення
// параметризованих масивів
template <typename Elem>
ostream& operator<<(ostream& os, const Array<Elem>& ar) {
   char chr=' ';
   cout<<"Array of "<<ar.size()<<':';
   for(size_t i=0; i<ar.size(); i++) {
      cout<<chr<<ar[i]; if(chr=' ') chr=',';
   }
   cout<<'.'<<endl;
   return os;
}
```

Аналогічно зміниться формат реалізації методів класу:

Лістинг 6.26

```
template <typename Elem> Array<Elem>::
      Array(const size_t sz):_size(sz), _allocator(new Elem[_size]) {
   return;
}
template <typename Elem> Array<Elem>::~Array() {
   delete [] _allocator;
}
template <typename Elem> Elem& Array<Elem>::
                                 operator[](const size_t index) {
   return _allocator[index];
}
template <typename Elem> const Elem& Array<Elem>::
                           operator[](const size_t index) const {
   return _allocator[index];
}
template <typename Elem> size_t Array<Elem>::size() const {
   return _size;
}
```

Як і в разі узагальнених функцій, визначення узагальненого класу дає не сам клас, а лише схему (шаблон) для його побудови. Створення конкретного класу за узагальненим визначенням параметризованого класу називають *конкретизацією шаблону* (template specialization). Один і той самий узагальнений клас можна конкретизувати за допомогою різних типів даних, стандартних і програмованих класів, визначених у програмі чи в бібліотеках (користувацьких або стандартних). Кожен зі згенерованих таким способом класів має власне осібне ім'я, що знову ж складається з двох частин: імені узагальненого класу й імені фактичного типу. Як приклад наведемо використання узагальненого масиву, яке містить водночас чотири різні конкретизації.

Лістинг 6.27

```
#include <string>
#include <iostream>
using namespace std;
#include "Array.h"
#include "Complex.h"
int main() {
   Array<int> ai(3); // Масив цілих чисел
   cout<<ai<<endl; ai[0]=1; ai[1]=2; ai[2]=3; cout<<ai<<endl;
   Array<char> ci(4); // Масив символів
   cout<<ci<<endl; ci[0]='a'; ci[1]='b'; ci[2]='c'; ci[3]='d';
   cout<<ci<<endl;
   Array<string> si(2); // Масив рядків (бібліотечний тип)
   cout<<si<<endl; si[0]="first"; si[1]="second"; cout<<si<<endl;
   Array<Complex> compi(3); //Масив комплексних чисел (програмований тип)
   cout<<compi<<endl; compi[0]=Complex(1, 1); compi[1]=Complex(1, 2);
   compi[2]=Complex(2, 1); cout<<compi<<endl;
   return 0;
}
```

У наведеній програмі використано такі конкретизації узагальненого класу масивів `Array`: масиви цілих чисел `Array<int>`, символів `Array<char>`, рядків зі стандартної бібліотеки `Array<string>` і комплексних чисел `Array<Complex>`, – визначених у попередньому розділі. Усі чотири класи різні, ніяк не пов'язані один з одним. Спільне для них лише те, що їхні визначення та реалізації автоматично згенеровано компілятором з одного й того самого узагальненого класу. Тому в разі модифікації програм обробки масивів достатньо буде змінювати лише один файл, що містить шаблон параметризованого класу. Унесення змін до коду шаблону автоматично призводитиме до модифікації кожної з конкретизацій під час її повторної компіляції.

Відмінності реалізацій можна простежити на прикладі виконання програми з лістингу 6.27. Після створення масивів стандартних типів виявиться, що вони заповнені сміттям. Конструктор масиву за допомогою оператора **new** лише резервує місце в пам'яті. Масив символів виявляється заповненим порожніми рядками. Можна припустити, що до кожного елемента масиву під час його створення оператором **new** буде викликано конструктор рядків без параметра. У ході ж створення масиву комплексних чисел для кожного елементу масиву викликається конструктор класу комплексних чисел. Варто виконати відповідний комп'ютерний експеримент, помістивши до конструктора комплексних чисел контрольну точку налагоджувача.

Вправа 6.5

Виконайте в режимі тестування програму з лістингу 6.27, доповнивши її створенням масивів інших програмованих типів: точок площини, відрізків, трикутників. Перевірте, що станеться, якщо клас елементів масиву не матиме конструктора без параметрів.

Знову маємо справу із явищем, схожим на розбухання коду, виявленим у попередньому розділі. Але тепер природа явища стає принципово іншою. Оскільки типи різних конкретизацій масивів принципово різняться, а тому різняться і коди їх обробки, то варто говорити не про розбухання об'єктного коду, а про економію вхідного, яка дозволяє тиражувати потрібний об'єктний код за спільним зразком.

Особливо варто проаналізувати, яким способом закодовано реалізацію класу та де вона міститься. Зазвичай повний опис класу ділять на дві частини: специфікацію та реалізацію. Специфікацію розміщують у заголовних файлах, код реалізації – у файлах вхідних текстів, які зазвичай мають розширення **.cpp**. Однак для узагальнених класів виникають проблеми, пов'язані з тим, що замість реалізації узагальнений клас містить лише схему для її генерування, яка має бути доступною на момент конкретизації. Справді, компіляція об'єктного коду за вхідним кодом реалізації узагальненого класу стає можливою лише під час обробки клієнта параметризованого класу, який задає його конкретизацію. Так, тільки виявивши, скажімо, що використовується масив цілих чисел, компілятор почне генерувати коди для його методів, а в разі масиву символів – відповідно методи масиву символів.

Тому до файлу, що містить код клієнта параметризованого класу, доведеться приєднувати як шаблон самого класу, так і шаблони реалізацій його методів і утиліт. Оскільки домовлено, що приєднуються лише заголовні файли, шаблони реалізації класу розміщують у його заголовному файлі.

Узагалі кажучи, є дві моделі компіляції узагальнених класів. Одну ми вже фактично розглянули. Вона полягає в *компіляції приєднанням* (compilation by including) до місця використання узагальненого класу повного тексту шаблону його визначення разом із шаблонами реалізації. Цю модель підтримують усі компілятори, які розуміють шаблони **template**. Однак її застосування суперечить принципу розділення інтерфейсу та реалізації, оскільки в заголовному файлі тепер міститимуться й перший, і друга.

Інша модель – *відокремлена компіляція* (compilation by separation). Її суть полягає в розмежуванні реалізації й інтерфейсу. Вона виявилася складнішою в реалізації, тому доступні системи програмування не підтримують її. Узагальнені класи, призначені для відокремленої компіляції, позначають ключовим словом export. Вивчення цієї можливості виходить за межі розгляданого тут курсу.

Повернімося до прикладу. Компілюючи головну функцію main(), компілятор одержить на вхід окрім власне тексту функції приєднані файли

Лістинг 6.28

```
#include <string>
#include <iostream>
using namespace std;
#include "Array.h"
#include "Complex.h"
```

зокрема файл **Array.h** з визначенням шаблону узагальненого масиву.

Виявивши визначення конкретного масиву, наприклад цілого – Array<**int**> ai(3), компілятор стане конкретизувати узагальнений клас Array до класу Array<**int**>. Конкретизації підлягає як будова класу, так і реалізація методів. Для того, щоб компілятор зміг відповідно конкретизувати методи, йому потрібний доступ до їхніх шаблонів. Цього легко досягти, якщо шаблони реалізації методів узагальненого класу міститимуться безпосередньо в заголовному файлі Array.h.

Наведемо повний текст файлу Array.h.

Лістинг 6.29

```
#ifndef _ARRAY_H_
#define _ARRAY_H_
#include <iostream>
using namespace std;
// Масив елементів типу Elem. Визначення класу
template <typename Elem=double>
class Array {
public:
   explicit Array(const size_t); // Конструктор масиву
```

```
   ~Array(); // Деструктор масиву
   Elem& operator[](const size_t index); // Модифікатор елемента масиву
// Селектор елемента масиву
   const Elem& operator[](const size_t index) const;
   size_t size() const; // Розмір масиву (селектор)
private:
   size_t _size; // Розмір масиву
   Elem* _allocator; // Указник на розміщення масиву
// Операції та функції, не визначені для масивів
   bool operator==(const Array&) const;
   Array(const Array&);
   Array& operator=(const Array&);
};
// Утиліта виведення
template <typename Elem>
ostream& operator<<(ostream& os,const Array<Elem>& seq);
// Масив елементів типу Elem. Реалізація класу. Реалізація конструктора
template <typename Elem>
Array<Elem>::Array(const size_t sz): _size(sz), _allocator(new El-
em[_size]) {
   return;
}
// Реалізація деструктора
template <typename Elem> Array<Elem>::~Array() {
   delete [] _allocator;
}
// Реалізація селектора-модифікатора елемента масиву
template <typename Elem> Elem& Array<Elem>::operator[](const size_t in-
dex) {
   return _allocator[index];
}
// Реалізація селектора елемента масиву
template <typename Elem>
const Elem& Array<Elem>::operator[](const size_t index) const {
   return _allocator[index];
}
// Реалізація селектора розміру масиву
template <typename Elem> size_t Array<Elem>::size() const {
   return _size;
}
// Реалізація утиліти виведення
template <typename Elem>
ostream& operator<<(ostream& os,const Array<Elem>& ar) {
```

```
   char chr=' ';
   cout<<"Array of "<<ar.size()<<':';
   for(size_t i=0; i<ar.size(); i++) {
      cout<<chr<<ar[i]; if(chr=' ') chr=',';
   }
   cout<<'.'<<endl;
   return os;
}
#endif
```

Вправа 6.6

Протестуйте узагальнений масив для різних типів його елементів.

Вправа 6.7

Визначте узагальнений клас рядків знаків за оголошенням шаблону

```
template <typename Symbol> class String;
```

Ідеться про рядки, сформовані зі знаків якогось типу, не обов’язково символьного `char`. Цей шаблон можна конкретизовувати типом довгих символів `wchar_t` й іншими, наприклад програмованими. Щоб побудувати ще одну конкретизацію, визначте клас смайликів і конкретизуйте шаблон рядків смайликами. Визначте шаблон утиліти виведення рядка. Виведіть рядок, складений зі смайликів.

Вправа 6.8

Визначте шаблон узагальненої операції лексикографічного порівняння узагальнених рядків за сигнатурою

```
template <typename Symbol>
bool operator< (const String<Symbol>&, const String<Symbol>&);
```

Протестуйте кілька конкретизацій.

6.3. Вкладені параметризовані типи

Подібно до того, як один і той самий кошик можна використовувати для зберігання і яблук, і мікросхем, будова та поведінка параметризованих контейнерів не залежить від їх умісту. Параметризований масив можна конкретизувати під масив цілих, дійсних або комплексних чисел, рядків чи трикутників. Оскільки кожна інша конкретизація створює свій інтерфейс, можливі нові застосування вхідного коду, тобто самого шаблону параметризованого класу.

Справді, клас **class** Array<**int**> – це приклад одного, а класи **class** Array<**char**> або **class** Array<Point> – іншого застосування шаблону параметризованих класів **template** <**typename** Elem> **class** Array. При цьому можна створювати які завгодно нові застосування цього шаблону. Варто лише не забувати, що кожна конкретизація шаблону зумовлює створення повної копії всього об’єктного коду, одержаного після підставляння нового ти́пового параметра чи параметрів.

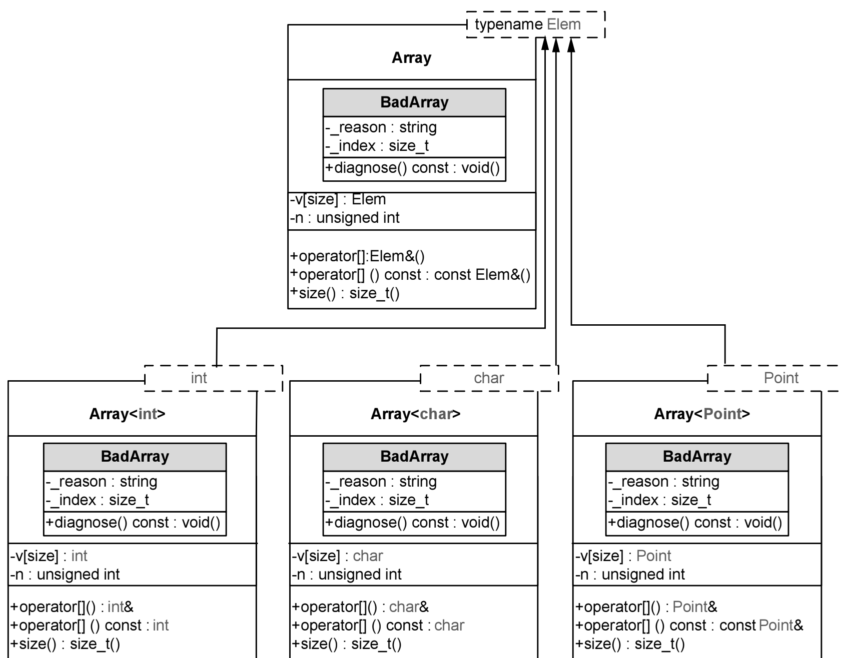


Рис. 6.5. Вкладені шаблони

Проаналізуємо поведінку класів, вкладених до параметризованих. Це може бути, наприклад, службовий клас, такий як BadArray. Якщо при ньому немає параметра, то може здатися, що вкладений клас непараметризований. Однак він має бути вкладеним до кожної нової конкретизації шаблону основного класу, а тому сам стає параметризованим.

На рис. 6.5 зображено три конкретизації шаблону параметризованого класу масивів Array. Оскільки шаблон містить вкладений клас, то кожна його конкретизація отримує власний варіант вкладеного класу. Справді, шаблон класу BadArray, вкладений до шаблону **template** <**typename**

Elem> **class** Array, конкретизовано відповідно класами **class** Array<**int**>::BadArray, **class** Array<**char**>::BadArray, **class** Array<Point>:: BadArray, згенерованими для кожної з трьох конкретизацій основного класу. Для кожного вкладеного класу компілятор створить окрему копію об'єктного коду, навіть якщо й специфікація, і реалізація фактично не залежать від ти́пового параметра.

Отож, спочатку наведемо повне визначення класу Array, помістивши до нього як визначення, так і реалізацію класу BadArray.

Лістинг 6.30

```
// Масив елементів типу Elem
// Визначення класу Array разом із реалізацією класу
// обробки помилкових ситуацій BadArray
template <typename Elem> class Array {
public:
// Клас для обробки помилкових ситуацій
   class BadArray {
   private:
      string _reason; // Причина аварійної ситуації
      size_t _index;  // Місце її виявлення
   public:
// Конструктор обробника
      BadArray(const string& reason="unknown", const size_t index=0):
              _reason(reason), _index(index) {
         return;
      }
      ~BadArray() {} // Деструктор обробника
// Функція виведення діагностики
      void diagnose() const {
         cerr<<_reason; if(_index!=0) cerr<<' '<<_index;
         cerr<<endl;
         return;
      }
   };
// Продовження визначення класу масивів
   explicit Array(const size_t); // Конструктор масиву
   ~Array(); // Деструктор масиву
   Elem& operator[](const size_t index); // Модифікатор елемента масиву
// Селектор елемента масиву
   const Elem& operator[](const size_t index) const;
   size_t size() const; // Розмір масиву (селектор)
private:
   size_t _size; // Розмір масиву
   Elem* _allocator; // Указник на місце локалізації масиву
```

```
// Операції та функції, не визначені для масивів
   bool operator==(const Array&) const;
   Array(const Array&);
   Array& operator=(const Array&);
};
```

Доповнимо реалізації операцій індексування реакцією на вихід індексу за межі масиву:

Лістинг 6.31

```
// Реалізація селектора-модифікатора елемента масиву
template <typename Elem> Elem& Array<Elem>::
                               operator[](const size_t index) {
   if(index<0 || index>=_size) throw BadArray("Bad index: ", index);
   return _allocator[index];
}
// Реалізація селектора елемента масиву
template <typename Elem> const Elem& Array<Elem>::
                               operator[](const size_t index) const {
   if(index<0 || index>=_size) throw BadArray("Bad index: ", index);
   return _allocator[index];
}
```

За допомогою наведеної далі тестової програми можна перевірити працездатність узагальненого масиву, доповненого обробкою виходу індексу за межі масиву. Щоб простежити за конкретизаціями шаблону вкладеного класу обробки помилок, у ній визначено декілька конкретизацій масиву: цілих, символів і дійсних чисел.

Лістинг 6.32

```
// Тестова програма для обробника аварійних ситуацій
// у разі виходу індексів за межі масивів
int main() {
   Array<int> ai(3); ai[0]=1; ai[1]=2; ai[2]=3;
   Array<char> ci(4); ci[0]='a'; ci[1]='b'; ci[2]='c'; ci[3]='d';
   Array<double> di(5);
   di[0]=0.1; di[1]=0.2; di[2]=0.3; di[3]=0.4; di[4]=0.5;
// 1. Перевіряємо масив цілих чисел
   try {
       cout<<ai<<endl; ai[100];
   }
// Помилковий індекс у масиві цілих
   catch(const Array<int>::BadArray& x) {
      cerr<<"Non-existing number"<<endl; x.diagnose();
   }
// 2. Перевіряємо масив символів
```

```
   try {
       cout<<ci<<endl; ci[1000];
   }
// Помилковий індекс у масиві символів
   catch(const Array<char>::BadArray& x) {
      cerr<<"Non-existing symbol"<<endl; x.diagnose();
   }
// 3. Перевіряємо масив дійсних чисел
   try {
       cout<<di<<endl; di[-1];
   }
// Помилковий індекс у масиві дійсних
   catch(const Array<double>::BadArray& x) {
      cerr<<"Non-existing real"<<endl; x.diagnose();
   }
   return 0;
}
```

Як видно з тексту програми, блок **catch** обробника аварійних ситуацій щоразу приймає екземпляр класу обробки помилок, який має інший тип. Відбувається своєрідне клонування класу BadArray у контексті різних конкретизацій масивів (рис. 6.6). Це або тип Array<**int**>::BadArray – клас обробки помилок, вбудований у клас масиву цілих чисел, або класи Array<**char**>::BadArray чи Array<**double**>::BadArray із відповідними призначеннями. У цьому розумінні сам вбудований клас стає узагальненим, оскільки визначає ціле сімейство класів, що відрізняються контекстом свого використання, тобто конкретизацією. Код самої обробки помилкових ситуацій для кожної конкретизації генерується окремо.

Array<int>:: BadArray	Array<char>:: BadArray	Array<double>:: BadArray
_reason : string = "Non-existing number" _index : size_t = 100	_reason : string = "Non-existing symbol" _index : size_t = 1000	_reason : string = "Non-existing real" _index : size_t = 4294967295

Рис. 6.6. Конкретизації вкладених шаблонів

Класом BadArray можна оперувати, як й іншими узагальненими класами. Щоб програмний код був зрозуміліший, визначення класу обробки аварійних ситуацій із відповідним посиланням на ти́повий параметр доцільно помістити за межами класу Array. Те саме стосується й самого класу BadArray, реалізації методів якого теж варто винести назовні.

Спочатку наведемо визначення класу Array:

Лістинг 6.33

```
#ifndef _ARRAY_H_
#define _ARRAY_H_
```

```
#include <iostream>
using namespace std;
// Масив елементів типу Elem
// Визначення класу Array разом із посиланням на клас обробки
// помилкових ситуацій BadArray
 template <typename Elem>
// Масив елементів типу Elem
class Array {
public:
// Випереджальне оголошення класу для обробки помилкових ситуацій
   class BadArray;
   explicit Array(const size_t); // Конструктор масиву
   ~Array(); // Деструктор масиву
   Elem& operator[](const size_t index); // Модифікатор елемента масиву
// Селектор елемента масиву
   const Elem& operator[](const size_t index) const;
   size_t size() const; // Розмір масиву (селектор)
private:
   size_t _size; // Розмір масиву
   Elem* _allocator; // Указник на місце локалізації масиву
// Операції та функції, не визначені для масивів
   bool operator==(const Array&) const;
   Array(const Array&);
   Array& operator=(const Array&);
};
```

У цьому самому заголовному файлі містяться реалізації операцій над масивом:

Лістинг 6.34

```
// Реалізація селектора-модифікатора елемента масиву
template <typename Elem>
Elem& Array<Elem>::operator[](const size_t index) {
   if(index>=_size) throw BadArray("Bad index: ", index);
   return _allocator[index];
}
// Реалізація селектора елемента масиву
template <typename Elem>
const Elem& Array<Elem>::operator[](const size_t index) const {
   if(index>=_size) throw BadArray("Bad index: ", index);
   return _allocator[index];
}
```

Сюди ж унесемо визначення класу BadArray:

```
// Клас для обробки помилкових ситуацій                    Лістинг 6.35
template <typename Type>
class Array<Type>::BadArray {
private:
   string _reason; // Причина аварійної ситуації
   size_t _index;  // Місце її виявлення
public:
// Конструктор обробника
   BadArray(const string& reason="unknown", const size_t index=0);
   ~BadArray();   // Деструктор обробника
   void diagnose() const; // Функція виведення діагностичних повідомлень
};
```

Доповнимо заголовний файл реалізацією методів класу BadArray:

```
// Реалізація конструктора обробника                       Лістинг 6.36
template <typename Type> Array<Type>::BadArray::
   BadArray(const string& reason, const size_t index):
            _reason(reason), _index(index) {
   return;
}
// Реалізація деструктора обробника
template <typename Type> Array<Type>::BadArray::~BadArray() {
   return;
}
// Реалізація функції виведення діагностичних повідомлень
template <typename Type> void Array<Type>::BadArray::diagnose() const {
   cerr<<_reason<<endl;
   if(_index!=0) cerr<<' '<<_index;
   cerr<<endl;
   return;
}
```

Уміст файлу закінчується командою препроцесора **#endif**.

Можна протестувати клас за допомогою тестової програми, наведеної раніше в лістингу 6.27, а можна записати узагальнену функцію обробки аварійних ситуацій – наприклад, таку:

```
// Шаблон функції тестування обробника аварійних           Лістинг 6.37
// ситуацій у разі виходу індексів за межі масивів
template <typename Type>
void testArray(const Array<Type>& a) {
   const size_t bigIndex=100;
   try {
```

```
      cout<<a<<endl; a[bigIndex];
   }
   catch(const Array<Type>::BadArray& x) {
      x.diagnose();
   }
   return;
}
```

Вона, звісно, тепер не міститиме фрагментів, специфічних для конкретних типів. Основну функцію тестової програми перепишемо так:

Лістинг 6.38

```
// Тестова програма для обробника аварійних ситуацій
// у разі виходу індексів за межі масивів
template <typename Type>
void testArray(const Array<Type>& a);
int main() {
   Array<int>ai(3); ai[0]=1; ai[1]=2; ai[2]=3;
   Array<char>ci(4); ci[0]='a'; ci[1]='b'; ci[2]='c'; ci[3]='d';
   Array<double>di(5);
   di[0]=0.1; di[1]=0.2; di[2]=0.3; di[3]=0.4; di[4]=0.5;
// Конкретизації функції тестування:
   testArray(ai);                // параметром цілого типу
   testArray(ci);                // параметром символьного типу
   testArray(di);                // параметром дійсного типу
   return 0;
}
```

Іншим прикладом узагальненого класу із вбудованим типом може бути клас обмежених стеків на базі масиву. Спробуйте розробити його самостійно.

Вправа 6.9

Реалізуйте узагальнений клас обмежених стеків на базі масиву, передбачивши обробку аварійних ситуацій за допомогою вбудованого типу відповідно до такої специфікації:

Лістинг 6.39

```
// Узагальнений клас обмежених стеків на базі масиву
template <class Elem>
class StackContainingArray {
private:
   static const size_t _bos; // Дно стека: Bottom Of the Stack
   size_t _top; // Верхівка стека
   Array<Elem>_stackArray; // Масив для розміщення стека
public:
```

```
   class BadStack; // Обробник помилкових ситуацій
// Конструктор стека
   explicit StackContainingArray(const size_t size):
            _stackArray(size), _top(_bos) {}
   ~StackContainingArray() {} // Деструктор стека
   bool empty() const; // Перевірити, чи порожній стек
   bool full() const; // Перевірити, чи заповнено стек
   const Elem& top() const; // Показати верхівку стека
   void pop(); // Виштовхнути верхівку стека
   void push(const Elem& value) // Проштовхнути елемент до стека
   size_t size() const; // Показати розмір стека
   size_t volume() const; // Показати ємність стека
};
```

Вправа 6.10

Запропонуйте специфікації параметризованих класів обмежених черг `BoundedQueue` та двобічних черг `BoundedDeque`, передбачивши обробку аварійних ситуацій за допомогою вбудованого типу. Реалізуйте черги на базі масивів підхожих типів.

Однак функції вкладених типів не обмежуються обробкою аварійних ситуацій. Прикладом змістовного типу, вкладеного до іншого типу, а саме до класу списків, може бути структура його вузлів `ListNode`, знову ж таки окрема для кожного ти́пового параметра `ListElem` шаблону параметризованих списків. Окрім полів даних, у структурі `ListNode` зручно мати конструктор, що зробить реалізацію зрозумілішою.

Тепер визначимо сам клас списків `List`, у реалізації якого використано вкладену структуру `ListNode`. Зверніть увагу на те, що для визначення класу списків не потрібно знати внутрішню будову структури `ListNode`. Тому в визначенні класу списків достатньо обмежитися її випереджальним оголошенням.

Лістинг 6.40

```
// Клас списків List елементів довільного типу ListElem
// Реалізація з використанням структури ListNode
template <typename ListElem> class List {
private:
   struct ListNode // Вкладений тип: структура вузла списку
// Атрибути класу списків:
   ListNode* _start; // Указник на перший вузол списку
   ListNode* _end; // Указник на останній вузол списку
   mutable ListNode* _current; // Указник на поточний вузол списку
   mutable bool _stop; // Атрибут-індикатор вичерпання списку
   size_t _amount; // Кількість елементів у списку
```

```
// Нереалізовані, а тому закриті копіювання:
// копіювання списків не передбачено
   List(const List&); List& operator=(const List&);
public:
   List(); // Конструктор порожнього списку
   ~List(); // Деструктор списку
   bool empty(){return _amount==0;} // Перевірити, чи порожній список
   size_t amount(){return _amount;} // Селектор кількості елементів
                                    // у списку
   List& add(const ListElem&); // Додати елемент у початок списку
   bool find(const ListElem&); // Знайти елемент списку
   List& del(); // Видалити поточний елемент списку
   List& del(const ListElem&); // Видалити заданий елемент списку
// Ітератори
   const List& start() const; // Почати ітерування списку
   const List& next() const; // Зробити крок ітерації
   bool stop() const {return _stop;} // Селектор атрибута вичерпання
                                     // списку
// Селектор поточного елемента списку
   const ListElem& getElem() const {return *_current->_elem;}
};
// Вкладений тип: структура вузла списку
template <typename ListElem> struct List<ListElem>::ListNode {
   const ListElem* _elem; // Член списку
   ListNode* _prev; // Указник на попереднього члена списку
   ListNode* _next; // Указник на наступного члена списку
// Конструктор вузла списку
   ListNode(const ListElem& elem,ListNode* next, ListNode* prev):
      _elem(&elem), _next(next), _prev(prev) {}
};
// Утиліта виведення списку
template <typename ListElem>
ostream& operator<< (ostream& os, const List<ListElem>& list);
```

Далі наведено реалізацію класу списків, деталі якої роз’яснено в коментарях.

Лістинг 6.41

```
// Конструктор порожнього списку
template <typename ListElem> List<ListElem>::List():
         _start(0),_end(0),_current(0),_amount(0),_stop(true) {
   return;
}
// Деструктор списку
template <typename ListElem> List<ListElem>::~List() {
```

```
// У порожньому списку нічого видаляти
   if(empty()) return;
// Видаляємо елементи списку в циклі
   start();
   do
      del();
   while(!stop());
   return;
}
```

Вправа 6.11

Перепишіть деструктор, реалізований тут за допомогою циклу з постумовою, із використанням циклу з передумовою. Якій із двох версій (з передумовою чи постумовою) варто віддати перевагу?

Лістинг 6.42

```
// Знайти елемент списку
template <typename ListElem>
bool List<ListElem>::find(const ListElem& anElem) {
// У порожньому списку нічого шукати
   if(empty()) return false;
// Поелементно переглядаємо список у циклі
   start();
   do {
// Якщо шуканий елемент знайдено, виходимо з циклу:
// поточний елемент – шуканий
      if(getElem()==anElem) return true;
      next();
   } while(!stop());
// Якщо цикл дійшов до кінця, шуканого елемента немає в списку
   return false;
}
// Видалити заданий елемент зі списку
template <typename ListElem>
List<ListElem>& List<ListElem>::del(const ListElem& anElem) {
// Якщо шуканий елемент знайдено, видаляємо його
   if(find(anElem)) del();
   return *this;
}
// Видалити поточний елемент списку
template <typename ListElem>
List<ListElem>& List<ListElem>::del() {
   ListNode* aPrev=_current->_prev; // Запам'ятовуємо попередній
                                    // елемент списку
```

```
   ListNode* aNext=_current->_next; // Запам'ятовуємо наступний
                                    // елемент списку
   delete _current; // Видаляємо поточний елемент
   _amount--;
   _current=aNext; // Оголошуємо поточним елементом наступний за видаленим
// Реорганізовуємо зв'язки в списку
// Якщо видалений елемент не був першим,
// зв'язуємо попередній елемент із наступним
   if(aPrev!=0) aPrev->_next=aNext;
   else
// а ні - то новий поточний елемент стає першим
      _start=_current;
// Якщо видалений елемент не був останнім,
// зв'язуємо наступний елемент із попереднім
   if(aNext !=0) aNext->_prev=aPrev;
// а ні - то новий поточний елемент стає останнім
   else _end=_current;
   _stop=_current==0;   // Перевіряємо список на вичерпаність
   return *this;
}
// Додати елемент у початок списку
template <typename ListElem>
List<ListElem>& List<ListElem>::add(const ListElem& anElem) {
   if(empty())
// Порожній список стає одноелементним
      _start=_end=_current= new ListNode(anElem, 0, 0);
   else {
// Додаємо елемент у початок непорожнього списку
      _current=new ListNode(anElem, _start, 0);
      _start->_prev=_current; _start=_current;
   }
   _amount++;
   return *this;
}
// Почати ітерування списку
template <typename ListElem>
const List<ListElem>& List<ListElem>::start() const {
   _current=_start; _stop=_current==0;
   return *this;
}
// Зробити крок ітерації
template <typename ListElem>
const List<ListElem>& List<ListElem>::next() const {
```

```
   _current=_current->_next; _stop=_current==0;
   return *this;
}
```

Реалізація утиліти виведення списку – хороший приклад застосування ітерації за списком:

Лістинг 6.43

```
// Утиліта виведення списку
template <typename ListElem>
ostream& operator<< (ostream& os, const List<ListElem>& list) {
   char separator=':'; list.start();
   while(!list.stop()) {
      os<<separator<<list.getElem();
      if(separator==':') separator=',';
      list.next();
   }
   return os;
}
```

Вправа 6.12

Протестуйте узагальнений список на прикладах списків символів, дійсних і комплексних чисел.

Вправа 6.13

Розгляньте узагальнений список і узагальнений рядок. Визначте й реалізуйте узагальнену операцію порівняння однотипних списку та рядка на рівність.

Вправа 6.14

Запропонуйте специфікації параметризованих класів необмежених черг `Queue` та двобічних черг `Deque`. Реалізуйте черги на базі списків підхожих типів.

6.4. Поліпараметризовані класи

Найцікавіші застосування параметризованих типів виникають у разі використання кількох параметрів у одному шаблоні. Як приклад можна навести зовсім простий клас векторів, параметризованих за розмірністю й типом своїх елементів. Пропонуємо визначити його самостійно.

Вправа 6.15

Визначте за оголошенням

```
template <typename Elem, size_t size> class Vector;
```

клас параметризованого вектора, котрий залежить від двох параметрів: ти́пового `Elem`, який визначає тип елемента вектора, і сталої `size`, що задає розмірність вектора (рис. 6.7).

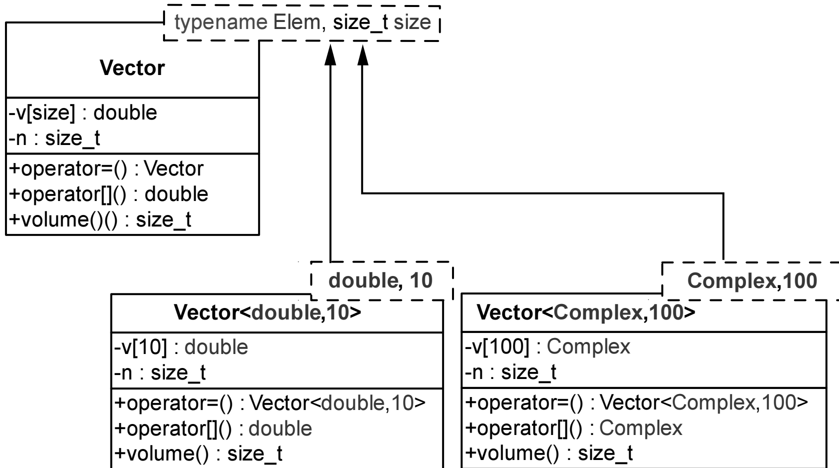


Рис. 6.7.

Іншим прикладом може слугувати параметризований клас послідовностей. Його параметри – тип елемента `Elem` і розмір блока пам'яті `seqBlock`.

Лістинг 6.44

```
// Послідовний контейнер елементів типу Elem
template <typename Elem, size_t seqBlock=100> class Sequence {
public:
   class BadSeq; // Вкладений тип: обробник помилкових ситуацій
// Конструктор послідовності
   explicit Sequence(const size_t size=seqBlock);
   ~Sequence(); // Деструктор послідовності
// Додати елемент у зазначене місце або в кінець послідовності
   Sequence& insert(const Elem& elem,
                    size_t index=std::numeric_limits<size_t>::max());
// Видалити елемент за його номером або з кінця послідовності
   Sequence& remove(const size_t
                    index=std::numeric_limits<size_t>::max());
// Модифікатор елемента за його номером
   const Elem& operator[](const size_t index) const;
// Селектор елемента за його номером
```

```
   Elem& operator[](const size_t index);
   size_t capacity() const; // Поточна ємність
   size_t size() const; // Поточний розмір
   bool empty() const; // Чи порожня послідовність?
   void clear(); // Очистити послідовність
   static void setBlockSize(const size_t); // Змінити розмір блока
// Ітератори
   void start() const {_current=_start;}
   void next() const {_current++;}
   bool stop() const {return (_current==_end);}
// Селектор поточного елемента послідовності
   const Elem& getElem() const {return *_current;}
private:
   size_t _size; // Розмір послідовності
   Array<Elem>* _seqArray; // Контейнер для зберігання послідовності
   mutable Elem* _current; // Поточний член послідовності
   Elem* _start; // Початок послідовності
   Elem* _end; // Кінець послідовності
   static size_t _seqBlock; // Стандартний розмір блока контейнера
// Збільшити ємність на times блоків
   void enlarge(const size_t times=2);
// Нереалізовані, а тому закриті копіювання:
// копіювання послідовностей не передбачено
   Sequence(const Sequence&); Sequence& operator=(const Sequence&);
};
// Утиліта виведення послідовності
template <typename Elem, size_t seqBlock>
ostream& operator<<(ostream& os, const Sequence<Elem, seqBlock>& seq);
```

Зауважимо, що другий параметр шаблону має задане за замовчуванням значення, а тому його можна пропускати, створюючи об’єкти. Наведемо реалізації класу.

Лістинг 6.45

```
// Конструктор послідовності
template <typename Elem, size_t seqBlock>
Sequence<Elem, seqBlock>::Sequence(const size_t size):
      _size(0),_start(0), _end(0), _current(0),
      _seqArray(new Array<Elem>(size)) {
   return;
}
// Деструктор послідовності
template <typename Elem, size_t seqBlock>
Sequence<Elem, seqBlock>::~Sequence() {
   delete _seqArray;
```

```
   return;
}
// Додати елемент у зазначене місце чи в кінець послідовності
template <typename Elem, size_t seqBlock>
Sequence<Elem, seqBlock>& Sequence<Elem, seqBlock>::
      insert(const Elem& elem, const size_t index) {
   if(_start==0) {
// Послідовність порожня, елемент буде на початку масиву
      _start=&(*_seqArray)[0];
      _end=_start;
   }
// Якщо значення індексу надто велике (зокрема, задане за
// замовчуванням значення параметра): додаємо елемент у кінець,
// а тому індексу надаємо значення розміру
   if(_size<index) index=_size;
// У разі потреби збільшуємо ємність
   if(_size+1>capacity()) enlarge();
// Збільшуємо розмір і указник на кінець
   ++_size; ++_end;
// Зсуваємо залишок від місця вставляння елемента до кінця послідовності
   for(size_t i=_size-1; i>index; i--)
      (*_seqArray)[i]=(*_seqArray)[i-1];
// Записуємо значення нового елемента на підготоване для нього місце
   (*_seqArray)[index]=elem;
   return *this;
}
// Збільшити ємність на times блоків
template <typename Elem, size_t seqBlock>
void Sequence<Elem, seqBlock>::enlarge(const size_t times) {
// Виділяємо новий масив потрібного розміру
   Array<Elem>* newArray=new Array<Elem>(times*_seqBlock+capacity()+1);
// Переписуємо до нього старі значення
   for(size_t i=0; i<_size; i++)
      (*newArray)[i]=(*_seqArray)[i];
   delete _seqArray; // Видаляємо старий масив
// Замінюємо його новим і перевстановлюємо ітератори
   _seqArray=newArray;
   _start=&(*_seqArray)[0]; _end=&(*_seqArray)[_size-1]+1;
   return;
}
// Видалити елемент за його номером або з кінця послідовності
template <typename Elem, size_t seqBlock> Sequence<Elem, seqBlock>&
         Sequence<Elem, seqBlock>::remove(const size_t index) {
   assert(!empty()); // Функція не застосовна до порожньої послідовності
```

```
   if(_size<index)
      index=_size; // Видаляємо останній елемент
// Посуваємо члени послідовності, що містяться праворуч
// від видаленого, на одне місце ліворуч
   for(size_t i=index-1; (i<_size)&&(i>0); i++)
      (*_seqArray)[i-1]=(*_seqArray)[i];
   --_size; // Зменшуємо розмір і перевстановлюємо ітератори
   if(empty()) _start=_end=0;
   else --_end;
   return *this;
}
```

Вправа 6.16

Завершіть реалізацію класу послідовностей. Протестуйте його.

Наведемо реалізацію операції виведення послідовності:

Лістинг 6.46

```
// Утиліта виведення послідовності
template <typename Elem, size_t seqBlock>
ostream& operator<<(ostream& os, const Sequence<Elem, seqBlock>& seq) {
   char chr=' '; cout<<"Size "<<seq.size()<<'('<<seq.capacity()<<')';
   seq.start();
   while(!seq.stop()) {
      cout<<chr<<seq.getElem(); if(chr=' ') chr=','; seq.next();
   }
   cout<<endl;
   return os;
}
```

Інший важливий приклад поліпараметризованого класу – параметризований асоціативний масив, який можна одержати, узагальнивши конструкцію з підрозділу 5.7. Нагадаємо, що асоційований масив складається з пар, які називають асоціаціями. Як ключ в асоціативному масиві можна використовувати як перший ти́повий параметр `First`, так і другий `Second`.

Лістинг 6.47

```
// Параметризований клас асоціативних масивів
template <typename First, typename Second>
class AssotiativeArray {
private:
   struct Pair;     // Вкладений тип: структура асоціації
// Атрибути класу асоціативних масивів:
   size_t _volume;  // ємність масиву
   Pair* _pairs;    // власне масив пар асоціацій
   size_t _size;    // поточний розмір масиву пар
```

```
   mutable Pair* _current; // указник на поточну пару
   void enlarge(); // Збільшити ємність асоціативного масиву
// Нереалізовані, а тому закриті копіювання:
// для асоціативних масивів копіювання не передбачено
   AssotiativeArray(const AssotiativeArray&);
   AssotiativeArray& operator=(const AssotiativeArray&);
public:
// Конструктор порожнього асоціативного масиву
   explicit AssotiativeArray(const size_t volume):_volume(volume+1),
            _pairs(new Pair[volume+1]),_size(0), _current(_pairs) {
      return;
   }
// Деструктор асоціативного масиву
   ~AssotiativeArray() {delete [] _pairs;}
// Ітератори
   void start() const {_current=_pairs+1;}
   void next() const {_current++;}
   bool stop() const {return (_current>_pairs+_size);}
// Селектори та модифікатори
   size_t size() const {return _size;}
   const First& getFirst() const {return _current->_first;}
   const Second& getSecond() const {return _current->_second;}
// Доступ за ключем First, можливе додання нового елемента до масиву:
// якщо елемента не знайдено, його додають до масиву
   Second& operator[](const First& first) {
      for(size_t i=_size; i>0; i--)
         if(first==_pairs[i]._first) return _pairs[i]._second;
      if(_size+1==_volume) enlarge();
      _pairs[++_size]._first=first;
      return _pairs[_size]._second;
   }
// Селектор за ключем First
   const Second& operator[] (const First& first) const {
      for(size_t i=_size; i>0; i--)
         if(first==_pairs[i]._first) return _pairs[i]._second;
      return _pairs[0]._second;
   }
// Доступ за ключем Second, можливе додання нового елемента до масиву:
// якщо елемент не знайдено, його додають до масиву
   First& operator[] (const Second& second) {
      for(size_t i=_size; i>0; i--)
         if(second==_pairs[i]._second) return _pairs[i]._first;
      if(_size+1==_volume) enlarge();
      _pairs[++_size]._second=second;
```

```
      return _pairs[_size]._first;
   }
// Селектор за ключем Second
   const First& operator[] (const Second& second) const {
      for(size_t i=_size; i>0; i--)
         if(second==_pairs[i]._second) return _pairs[i]._first;
      return _pairs[0]._first;
   }
};
// Вкладений тип: структура асоціації
template <typename First, typename Second>
struct AssotiativeArray<First, Second>::Pair {
   First _first; Second _second;
// Конструктор без параметрів
   Pair(): _first(First()), _second(Second()) {}
// Змістовний конструктор
   Pair(const First& first, const Second& second):
      _first(first), _second(second) {}
};
```

Вправа 6.17

Функцію збільшення масиву `enlarge()` реалізуйте самостійно.

Ось проста програма тестування класу асоціативних масивів:

Лістинг 6.48

```
int main() {
   AssotiativeArray<string, int> staff(5);
   staff[123456789]="John the Programmer";
   staff[345612678]="Merry the Hacker";
   staff[247665617]="Bob the Cracker";
   staff[375665611]="Tom the Writer";
   staff[195473116]="Jimmy the Tester";
   staff[111111111]="Six's a crowd";
   cout<<staff<<endl;
   return 0;
}
```

Вправа 6.18

Проаналізуйте виконання тестової програми. Доповніть її тестуванням доступу за другим параметром.

На завершення цього підрозділу зробимо зауваження стосовно реалізації конструктора асоціативного масиву. По суті, виділяючи пам'ять під масив, конструктор має заповнювати її фіктивними парами асоціацій.

Цим зумовлене обмеження на конкретизації ти́пових параметрів, у яких мають бути конструктори без параметрів. Це звична вимога до створення масиву. Однак проблема полягає не лише в обмеженні. Фактично кожен елемент асоціативного масиву створюють двічі: спочатку фіктивно під час створення масиву пар, потім – у ході фактичного додання пари до масиву. У наступному підрозділі ми усунемо цю надмірність, відмовившись від застосування масиву.

6.5. Параметризовані вкладені типи

Наступний крок в узагальненому програмуванні – параметризовані вкладені типи, які дають змогу гнучко комбінувати ти́пові параметри.

Розглянемо спочатку узагальнення асоціативного масиву, а саме список пар такого вигляду:

Лістинг 6.49

```
// Клас асоціативних списків пар вигляду (Source, Target)
template <typename Source, typename Target>
class AssotiativeList {
private:
   struct Pair; // Вкладений тип: структура асоціації
   struct ListNode; // Вкладений тип: структура вузла списку
// Атрибути класу списків:
   ListNode* _start; // указник на перший вузол списку
   ListNode* _end; // указник на останній вузол списку
   mutable ListNode* _current; // указник на поточний вузол списку
   mutable bool _stop; // Індикатор вичерпання списку
   size_t _amount; // кількість елементів у списку
// Нереалізовані, а тому закриті копіювання:
// копіювання списків не передбачено
   AssotiativeList(const AssotiativeList&);
   AssotiativeList& operator=(const AssotiativeList&);
public:
   AssotiativeList(); // Конструктор порожнього асоціативного списку
   ~AssotiativeList(); // Деструктор асоціативного списку
   bool find(const Source&); // Пошук елемента за ключем
   bool empty(){return _amount==0;} // Перевірка, чи порожній список
   size_t amount(){return _amount;} // Селектор кількості елементів
                                     // у списку
   AssotiativeList& del(const Source&); // Видалення заданого
                                         // елемента списку
   AssotiativeList& del(); // Видалення поточного елемента списку
// Ітератори:
   const AssotiativeList& start() const; // почати ітерування списку
```

```
   const AssotiativeList& next() const; // зробити крок ітерації
   bool stop() const {return _stop;} // перевірка вичерпання списку
// Селектори поточної пари
   const Target& getTarget() const {
      return _current->_elem->_target;
   }
   const Source& getSource() const {
      return _current->_elem->_source;
   }
};
// Утиліта виведення списку
template <typename Source, typename Target>
ostream& operator<<(ostream&,const AssotiativeList<Source, Target>&);
// Вкладений тип: структура асоціації
template <typename Source, typename Target>
struct AssotiativeList<Source, Target>::Pair {
   const Source& _source; const Target& _target;
// Змістовний конструктор
   Pair(const Source& source,const Target& target):
      _source(source), _target(target) {}
};
// Вкладений тип: структура вузла списку
template <typename Source, typename Target>
struct AssotiativeList<Source, Target>::ListNode {
   const Pair* _elem; // Член списку
   ListNode* _prev; // Указник на попереднього члена списку
   ListNode* _next; // Указник на наступного члена списку
// Конструктор вузла списку
   ListNode(const Pair& elem, ListNode* next, ListNode* prev):
      _elem(&elem), _next(next), _prev(prev) {}
};
```

Реалізація класу списків пар певною мірою узагальнює асоціативний масив і список:

Лістинг 6.50

```
// Конструктор порожнього списку пар, ідентичний до
// конструктора списку
template <typename Source, typename Target>
AssotiativeList<Source, Target>::AssotiativeList():
   _start(0),_end(0),_current(0),_amount(0),_stop(true) {
      return;
}
// Деструктор списку
template <typename Source, typename Target>
```

```
AssotiativeList<Source, Target>::~AssotiativeList() {
// У порожньому списку нічого видаляти
   if(empty()) return;
// Поелементно очищаємо список у циклі
   start();
   do
      del();
   while(!stop());
   return;
}
// Пошук елемента списку за першим компонентом пари як за ключем
template <typename Source, typename Target>
bool AssotiativeList<Source, Target>:: find(const Source& source) {
// У порожньому списку нічого шукати
   if(empty()) return false;
// Поелементно переглядаємо список у циклі
   start();
   do {
      if(getSource()==source)
// Якщо шуканий елемент знайдено, виходимо з циклу:
// поточний елемент – це шуканий
         return true;
      next();
   } while(!stop());
// Якщо цикл дійшов до кінця, шуканого елемента немає в списку
   return false;
}
// Видалення елемента зі списку за ключем
template <typename Source, typename Target>
AssotiativeList<Source, Target>& AssotiativeList<Source, Target>::
   del(const Source& source) {
   if(find(source)
// Якщо шуканий елемент знайдено, видаляємо його
      del();
   return *this;
}
// Видалення поточного елемента списку
template <typename Source, typename Target>
AssotiativeList<Source, Target>& AssotiativeList<Source, Target>::del() {
   ListNode* aPrev=_current->_prev; // Запам'ятовуємо попередній
                                    // елемент списку
   ListNode* aNext=_current->_next; // Запам'ятовуємо наступний
                                    // елемент списку
   delete _current; // Видаляємо поточний елемент
```

```
   _amount--;
   _current=aNext; // Оголошуємо поточним елементом наступний за видаленим
// Реорганізовуємо зв'язки в списку
// Якщо видалений елемент не був першим, зв'язуємо попередній
// елемент із наступним
   if(aPrev!=0) aPrev->_next=aNext;
// а ні - то новий поточний елемент стає першим
   else _start=_current;
// Якщо видалений елемент не був останнім, зв'язуємо наступний
// елемент із попереднім
   if(aNext !=0) aNext->_prev=aPrev;
// а ні - то новий поточний елемент стає останнім
   else _end=_current;
// Перегляд списку завершено
   _stop=_current==0;
   return *this;
}
```

Вправа 6.19

Реалізуйте функції організації ітерації `start()` і `next()`.

Вправа 6.20

Якими засобами потрібно доповнити клас асоціативних списків для того, щоб можна було реалізувати оператор його виведення? Визначте й реалізуйте цей оператор.

Тепер займемося розробленням параметризованого класу, що міститиме вкладений клас, параметризований за допомогою іншого ти́пового параметра. Особливість цієї конструкції полягає в генеруванні якої завгодно кількості нових вкладених типів.

Розглянемо клас функціональних послідовностей, об'єкти якого конкретизуватимуться як послідовності функціональних пар. Усі послідовності мають спільну область визначення, але в кожної з них може бути власна множина значень. Відповідну діаграму класів наведено на рис. 6.8.

Як видно з діаграми, маємо параметризований клас послідовностей типу `MappedSequence<Elem>`, складених із членів типу `Elem`. Послідовність назвемо функціональною, якщо кожному її члену поставлено у відповідність його образ – об'єкт іншого класу `Target`. Саму відповідність задано асоціативним списком відповідностей, його тип – `AssotiativeList<Elem, Target>`. Зберемо образи всіх членів початкової послідовності `MappedSequence<Elem>` до іншої послідовності, її типом стане `Sequence<Target>` – тип послідовності, що складається з об'єктів типу `Target`.

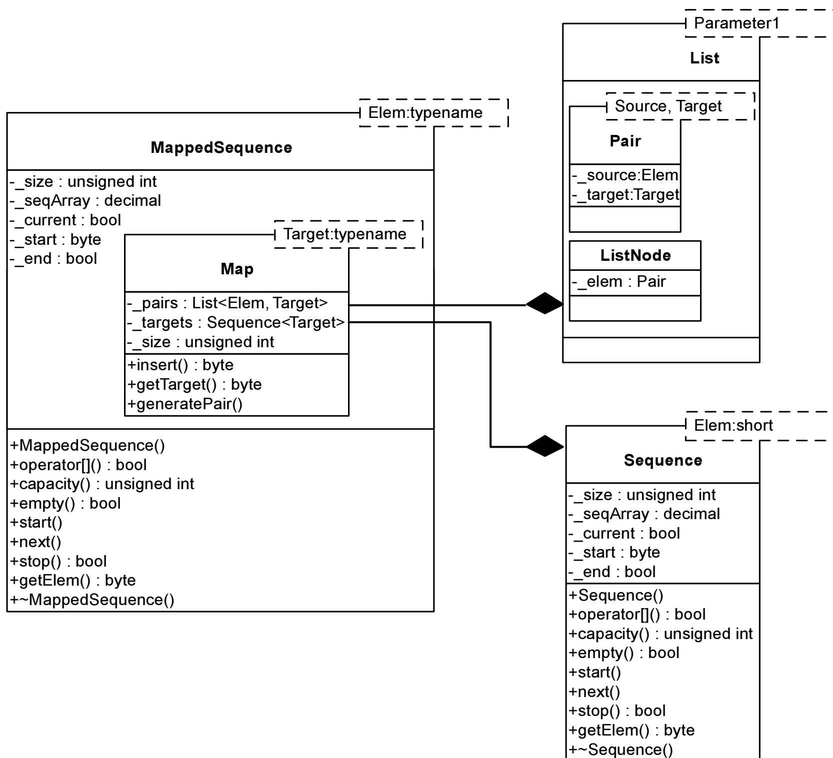


Рис. 6.8. Подвійна параметризація

Так з’являється функціональний клас `Map<Elem, Target>`, вкладений до класу функціональних послідовностей `MappedSequence Elem>`. Він містить у собі список функціональних пар `AssotiativeList <Elem, Target>` і послідовність `Sequence<Target>` образів початкової послідовності `MappedSequence<Elem>`.

Нове в цій діаграмі – вкладений тип функціонального параметра, не пов’язаного із зовнішнім типом, до якого його вкладено. Справді, початковий клас послідовностей залежить від одного ти́пового параметра `Elem`, тоді як вкладений тип `Map<Elem, Target>` –від двох ти́пових параметрів. Це означає, що за однією й тією самою послідовністю `MappedSequence <Elem>` можна побудувати різні функціональні класи.

Розглянемо приклад. Візьмемо послідовність a, складену з трьох точок площини:

```
a={z1=(0,1); z2=(1,1); z3=(1,0)}.
```

Її тип – `MappedSequence<Point>`. Побудуємо список пар `list`, який кожній точці площини ставить у відповідність комплексне число

```
list(complex)={((0,1), i); ((1,1), 1+i); ((1,0), 1)}.
```

Його тип – `AssotiativeList<Point, Complex>`. Тоді послідовність образів `complex(a)` послідовності `a` матиме тип `Sequence<Complex>` і складатиметься з трьох комплексних чисел

```
complex(a)={i; 1+i; 1}.
```

Якщо ж точці площини поставити у відповідність дійсне число (наприклад, її абсцису), то список пар відповідностей перетвориться на

```
list(absciss)={((0,1), 0); ((1,1), 1); ((1,0), 1)},
```

типу `AssotiativeList<Point, double>`, а послідовністю образів стане послідовність із двох членів –

```
absciss(a)={0; 1}.
```

Отже, в одній конкретизації `MappedSequence<Point>` шаблону `template <typename Source> class MappedSequence<Source>` використано дві різні конкретизації шаблону вкладеного класу `template <typename Elem> template <typename Target> class MappedSequence<Elem>::Map`, а саме `Map<Point, double>` і `Map<Point, Complex>`.

Вправа 6.21

Визначте послідовність ординат для наведеної тут послідовності точок площини.

Розглянемо тепер визначення класу функціональних послідовностей:

Лістинг 6.51

```
// Клас функціональних послідовностей, заданих
// відображеннями типу Source до типу Target
template <typename Source, size_t seqBlock=100> class MappedSequence {
public:
// ****************************************
public:
// Вкладений функціональний клас: окрім типового параметра Source,
// залежить від власного параметра Target
```

```
   template <typename Target> class Map;
// Конструктор функціональної послідовності
   explicit MappedSequence(const size_t size=seqBlock);
   ~MappedSequence(); // Деструктор функціональної послідовності
// Додати до послідовності
   MappedSequence& insert(const Source& elem,
      size_t index=std::numeric_limits<size_t>::max());
// Видалити елемент за його номером
   MappedSequence& remove (const size_t index=std::
      numeric_limits<size_t>::max());
   const Source& operator[](const size_t index) const; // Селектор
   Source& operator[](const size_t index); // Модифікатор
   size_t capacity() const; // Поточна ємність
   size_t size() const; // Поточний розмір
   bool empty() const; // Чи порожня послідовність?
   void clear(); // Очистити послідовність
   static void setBlockSize(const size_t); // Змінити розмір блока
// Ітератори
   void start() const {_current=_start;}
   void next() const {_current++;}
   bool stop() const {return (_current==_end);}
   const Source& getElem() const {return *_current;}
private:
   size_t _size; // Розмір послідовності
   Array<Source>* _seqArray; // Контейнер для зберігання
                             // розміру послідовності
   mutable Elem* _current; // Поточний член послідовності
   Elem* _start; // Початок послідовності
   Elem* _end; // Кінець послідовності
   static size_t _seqBlock; // Стандартний розмір блока контейнера
   void enlarge(const size_t times=2); // Збільшити ємність на times
блоків
// Нереалізовані, а тому закриті копіювання:
// копіювання послідовностей не передбачено
   MappedSequence(const MappedSequence&);
   MappedSequence& operator=(const MappedSequence&);
};
// *********************** Вкладений функціональний клас: окрім
// типового параметра Source, залежить від власного параметра Target
template <typename Elem, size_t seqBlock>
template <typename Target> class MappedSequence<Elem, seqBlock>::Map {
public:
   Map(): _size(0), // Конструктор функціонального класу
```

```
   _pairs(new AssotiativeList<Source, Target>()), // створює асоціативний
                                                  // список пар
   _targets(new Sequence<Target, seqBlock>()) // і порожню послідовність
                                              // образів
   {}
// Деструктор функціонального класу
   ~Map() {
      delete _pairs; // Видалення асоціативного списку
      delete _targets; // Видалення послідовності образів
      return;
   }
// Додання нового образу до послідовності
   Sequence<Target, seqBlock>& insert(const Target& target,
      size_t index=std::numeric_limits<size_t>::max()) {
// Делегування функції класу послідовностей
      _targets=&_targets->insert(target, index);
      return *_targets;
   }
// Повернення поточного образу
   const Target& getTarget() const {return _targets->getElem();}
// Створення нової пари
   Map& generatePair(const Source& where, const Target& what) {
      _pairs=&_pairs->add(where, what); ++_size;
      return *this;
   }
// Індексування за першим аргументом пар
   const Target& operator[](const Source& where) {
       assert(_pairs->find(where))
       return _pairs->getTarget();
   }
private:
   size_t _size;
   AssotiativeList<Source, Target>* _pairs; // Список функціональних пар
   Sequence<Target, seqBlock>* _targets; // Послідовність образів
};
// ****************** Кінець вкладеного функціонального класу
```

До визначення класу доцільно, як завжди, долучити сигнатуру оператора виведення

```
template <typename Source, size_t seqBlock>
ostream& operator<<(ostream& os,
         const MappedSequence<Source, seqBlock>& seq);
```

Повну реалізацію класу MappedSequence залишаємо читачеві. Її легко одержати з реалізації класу послідовностей з лістингу 6.44.

Вправа 6.22

Проаналізуйте наведену далі тестову програму та доповніть її блоком випробувань **try**, збільшивши кількість тестових прикладів.

Лістинг 6.52

```
int main() {
   const int size=10;
// Визначення функціональної послідовності
   MappedSequence<Point, size> pseq;
// Визначення вкладених функціональних класів
   MappedSequence<Point, size>::Map<Point> pinv;
   MappedSequence<Point, size>::Map<double> mod;
   MappedSequence<Point, size>::Map<Complex> pcomp;
// Заповнення послідовності
   pseq.insert(Point(0, 1)).insert(Point(1, 2)).insert(Point(2, 3)).
      insert(Point(3, 4));
   cout<<pseq; pseq.start();
   while(!pseq.stop()) {
      Point current=Point(pseq.getElem());
      Complex u=Complex(current.x(), current.y()); pcomp.insert(u);
      pcomp.generatePair(pseq.getElem(),pcomp.getTarget());
      cout<<pcomp[pseq.getElem()]<<',';
      double z=current.x(); current.x()=current.y(); current.y()=z;
      pinv.insert(current);
      pinv.generatePair(pseq.getElem(),pinv.getTarget());
      cout<<pinv[pseq.getElem()]<<',';
      double ro=current.module(); mod.insert(ro);
      mod.generatePair(pseq.getElem(),mod.getTarget());
      cout<<mod[pseq.getElem()]<<';'; cout<<endl; pseq.next();
   }
   return 0;
}
```

Наведемо ще один приклад використання функціональних послідовностей. Для цього розглянемо конкретизацію шаблону **template** <**typename** Source> **class** MappedSequence<Source> послідовністю слів MappedSequence<String>. За послідовністю слів можна побудувати дві послідовності: їхніх довжин (тип цієї послідовності – Sequence<**int**>) й обернених слів (її тип – Sequence<String>). На рис. 6.9 показано, що в цьому разі конкретизація шаблону словами MappedSequence<String> міститиме дві вкладені конкретизації функціональних класів: числами

`MappedSequence<String>::Map<int>` і рядками `MappedSequence<String>::Map<String>`. Об'єкти першого класу матимуть вигляд послідовностей пар, що складатимуться зі слів і їхніх довжин. Об'єкти другого класу – пари «слово, обернене слово».

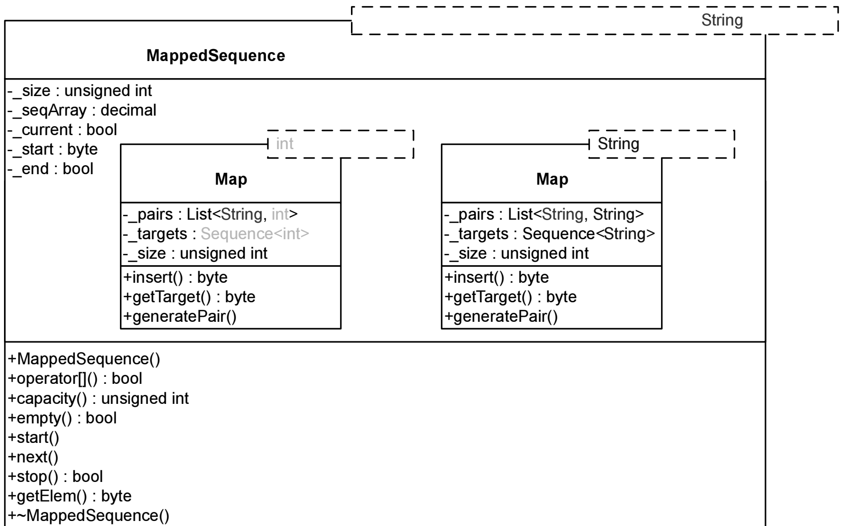


Рис. 6.9. Вкладені поліпараметризовані класи

Вправа 6.23

Користуючись діаграмою рис. 6.9, складіть програму, що будуватиме відображення послідовності слів відповідно в послідовність їхніх довжин і обернених слів.

Розглянуті в цьому розділі методи узагальненого програмування демонструють можливості генерування нових застосувань параметризованих класів за допомогою конкретизації ти́пових параметрів без переписування коду.

6.6. Розповсюдження шаблонів

В цьому підрозділі ми розглянемо корисну техніку часткової спеціалізації шаблонів, яка дозволить об'єднати під спільним іменем параметризованого класу різні його реалізації. Це досягається додатковими обмеженнями на параметри шаблонів як типові, так і чисельні. Часткова спе-

ціалізація допускається лише для програмованих типів, тобто для класів і структур. Що стосується функцій, то певною мірою їх часткова спеціалізація досягається за рахунок поєднання шаблонів і конкретних функцій для явно вказаних типів (лістинг 3.168). Спосіб вишуканішої спеціалізації шляхом використання функцій-статичних членів буде розглянуто наприкінці цього підрозділу.

Розглянемо калькулятор комплексних чисел, схожий на запропонований в лістингу 5.45 для випадку дійсних чисел. Відмінність полягатиме у прямому вживанні операцій, заданих у вигляді програмованих операцій класу `Array` з лістингу 4.58.

Лістинг 6.53

```
// Комплексний калькулятор
class ComplexCalc {
private:
   Complex_a; // Регістр акумулятора
   Complex_x; // Регістр x
public:
// Конструктор
   ComplexCalc(const Complex& a=0, const Complex& x=0):_a(a),_x(x) {}
// Селектори
   const Complex& getA()const {return _a;}
   const Complex& getX()const {return _x;}
// Тип указника бінарних операції над комплексними числами
   typedef const Complex (*const Operator)(const Complex&, const Complex&);
// Функція виклику довільноъ бінарної операції
   const Complex doOper(Operator po){ return _a=op(_a,_x);}
};
```

Фрагмент випробування класу на лістингу 6.54.

Лістинг 6.54

```
// Тест комплексного калькулятора
ComplexCalc c(1, Complex(0,1)); cout<<c.doOper(operator+)<<endl;
```

Спроба побудувати узагальнення у вигляді параметризованого класу (лістинг 6.54) наткнеться на проблему конкретизації цього шаблону стандартними типами даних.

Лістинг 6.55

```
// Параметризований калькулятор
template <class Arithm> class Calc {
private:
   Arithm _a; // Регістр акумулятора
   Arithm _x; // Регістр x
public:
```

```
// Конструктор
   Calc(const Arithm& a=0, const Arithm& x=0): _a(a),_x(x) {}
// Селектори
   const Arithm& getA()const {return _a;}
   const Arithm& getX()const {return _x;}
// Тип указника бінарних операції над даними типу Arithm
   typedef const Arithm (*const Operator) (const Arithm&, const Arithm&);
// Функція виклику довільної бінарної операції
   const Arithm doOper(Operator po) {return _a=op(_a,_x);}
};
```

Дійсно спроба конкретизації комплексними числами, як і будь-яким іншим програмованим типом з бінарними операціями потрібної сигнатури цілком можлива

Лістинг 6.56

```
// Тест конкретизації калькулятора
Calc<Complex> c(1, Complex(0,1)); cout<<c.doOper(operator+)<<endl;
```

Але конкретизація стандартними типами даних, наприклад, дійсними числами призведе до синтаксичної помилки, викликаної відсутністю відповідної програмованої операції. В такому випадку застосовують стандартний прийом загортання даних у обгортку. Наведену на лістингу 6.57.

Лістинг 6.57

```
// Обгортка примітивних типів даних
template <typename Type> class Wrapper {
private:
// Відсилка до вмісту обгортки
   const Type& _data;
public:
// Конструктор обгортки
   Wrapper(Type& data):_data(data) {}
// Неявний конвертор обгортки до типу Type
   operator const Type&() const {return _data;}
// Селектор
   const Type& data() const {return _data;}
// Модифікатор
   Type& data() {return _data;}
};
```

Визначення класу доповнимо обгортками операцій, одна з яких наведена на лістингу 6.58.

Лістинг 6.58

```
// Обгортка операції додавання
template <typename Type>
const Type operator+ (const Wrapper<Type>& a, const Wrapper<Type>& b) {
```

```
   return a.data()+b.data();
}
```

Тепер залишилося визначити калькулятор загорнутих арифметичних типів

Лістинг 6.59

```
// Часткова спеціалізація шаблону Calc
template <template<typename> class Wrapper, class Arithm>
class Calc<Wrapper<Arithm>> {
   Arithm _a; // Регістр акумулятора
   Arithm _x; // Регістр x
public:
// Конструктор
   Calc (const Arithm& a=0, const Arithm& x=0): _a(a),_x(x) {}
// Селектори
   const Arithm& getA() const {return _a;}
   const Arithm& getX() const {return _x;}
// Модифікатори
   void setA (const Arithm& a ) {_a=a;}
   void setX (const Arithm& x ) {_x=x;}
// Тип указника бінарних операцій над загорнутими типами
   typedef const Arithm (* const Operator) (const Wrapper<Arithm>&,
      const Wrapper<Arithm>&);
// Функція виклику довільної бінарної операції
   const Arithm doOper(Operator op) {return _a=op(_a,_x);}
};
```

Зустрівши в тексті конкретизацію шаблону `Calc`, компілятор спершу з'ясує, чи не підпадає вона під наявний частинний випадок. Так визначення `Calc<Wrapper<double>> d(1,1);` як і `Calc<Wrapper<Complex>> cd(1,1);` означатимуть застосування часткової спеціалізації, в той час як `Calc<Complex> cc(1,1);` – основного шаблону. Але в кожному разі мова йтиме про загорнуті типи даних.

Вправа 6.24

Виконати тестування калькулятора лістингу 6.54 і шаблонів з лістингів 6.55-6.59.

Тепер залишилося розібратися з тим, який же випадок калькулятора вважати основним, а який частковим. Для цього визначимо клас універсальних калькуляторів, придатних для всіх випадків.

Лістинг 6.60

```
// Шаблон універсального калькулятора
template <class Arithm> class UniCalc {
   Arithm _a; // Регістр акумулятора
```

```
   Arithm _x; // Регістр x
public:
// Конструктор
   UniCalc (const Arithm& a =0, const Arithm& x=0): _a(a),_x(x) {}
// Селектори
   const Arithm& getA() const {return _a;}
   const Arithm& getX() const {return _x;}
// Модифікатори
   void setA (const Arithm& a ) {_a=a;}
   void setX (const Arithm& x ) {_x=x;}
// Тип указника бінарних операцій над загорнутими типами
   typedef const Arithm (* const Operator) (const Wrapper<Arithm>&,
      const Wrapper<Arithm>&);
// Функція виклику довільної бінарної операції
   const Arithm doOper(Operator op) {
      return _a=op(_a,_x);
   }
};
```

Тепер конкретизації калькулятора виглядатимуть природніше як `UniCalc<Complex> cc(1,1);` або `UniCalc<double> d(1,1)`.

Частковою спеціалізацією шаблону лістингу 6.60 може стати варіант калькулятора комплексних чисел без обгорток.

Лістинг 6.61

```
// Часткова спеціалізація калькулятора комплексних чисел
template <> class UniCalc<Complex> {
private:
   Complex_a; // Регістр акумулятора
   Complex_x; // Регістр x
public:
// Конструктор
   UniCalc(const Complex& a=0, const Complex& x=0):_a(a),_x(x) {}
// Селектори
   const Complex& getA()const {return _a;}
   const Complex& getX()const {return _x;}
// Тип указника бінарних операцій над комплексними числами
   typedef const Complex (*const Operator) (const Complex&, const Complex&);
// Функція виклику довільної бінарної операції
   const Complex doOper(Operator po) {return _a=op(_a,_x);}
};
```

В контексті визначення лістингу 6.61 визначення як `UniCalc<Complex> cc(1, 1);` призведе до використання калькулятора не загорнутих комплексних чисел.

Вправа 6.25

Виконати тестування універсального калькулятора лістингу 6.60 і його часткової спеціалізації з лістингу 6.61.

Повернемося до розділу 5.5, в якому ми визначили декілька типів маніпуляторів, зокрема вкладені маніпулятори і маніпулятори масивів. Пригадаємо, як використання шаблону дозволило нам зекономити поняття при визначенні масивів різних типів. Замість `ArrayOfInt`, `ArrayOfChar`, `ArrayOfPoint`, `ArrayOfComplex` тощо ми в підрозділі 6.2 визначили параметризований клас масивів **`template`** `<`**`class`** `Type>` **`class`** `Array`, конкретизації якого: `Array<`**`int`**`>`, `Array<char>`, `Array<Point>` - давали нам шукані класи конкретних масивів. У випадку маніпуляторів ми теж можемо визначити параметризований клас **`template`** `<`**`class`** `Pointee>` **`class`** `SmartPtr`, наприклад, у спосіб, поданий на лістингу 6.53.

Лістинг 6.62

```
// Параметризований маніпулятор об’єктами Pointee
template <class Pointee> class SmartPtr {
private:
   Pointee* _pointee; // Об’єкт маніпулювання
   SmartPtr(SmartPtr&); // (Не)копіювальний конструктор
   SmartPtr& operator=(SmartPtr&); // (Не)копіювальне присвоєння
public:
   explicit SmartPtr(Pointee* p=0); // Конструктор, закритий від
                                    // конвертування типів
   ~SmartPtr();// Деструктор
   bool defined() const; // Перевірка визначеності маніпулятора
// Специфічні операції над указниками
// Операція доступу за указником
   Pointee* operator->() const{return _pointee;}
// Операція розіменування маніпулятора
   Pointee& operator*() const{return *_pointee;}
   SmartPtr& reset(Pointee&); // Приєднання об’єкта
   Pointee& release(); // Від’єднання об’єкта
};
template <class Pointee> ostream& operator<<(ostream&,
   const SmartPtr<Pointee>&); // Виведення об’єкта маніпуляції
```

Вправа 6.26

Запрограмуйте реалізацію параметризованого класу `SmartPtr`, користуючись лістингами підрозділів 5.5.1 або 5.5.2.

Далі можемо визначити параметризований клас вкладених маніпуляторів **`template`** `<`**`class`** `Pointee>` **`class`** `NestedSmartPtr` (лістинг 6.63)

```
// Параметризований маніпулятор маніпуляторами                Лістинг 6.63
template <typename NestedPointee> class NestedSmartPtr {
private:
   NestedPointee* _pointee; // Маніпулятор першого рівня
   NestedSmartPtr(NestedSmartPtr& source);
NestedSmartPtr& operator=(NestedSmartPtr&);
public:
// Конструктор, закритий від конвертування типів
   explicit NestedSmartPtr(NestedPointee* pt=0);
   ~NestedSmartPtr() {delete _pointee;} // Деструктор
// Специфічні операції над указниками
// Операція доступу за указником
   const NestedPointee& operator->() const {return *_pointee;}
// Операція розіменування маніпулятора
   NestedPointee& operator*() const {return *_pointee;}
   NestedPointee* release();// Від'єднання об'єкта
   void reset(NestedPointee* pt =0); // Приєднання об'єкта
};
```

Він відрізнятиметься від класу SmartPtr перш за все іменем, сигнатурою та реалізацією операції доступу за указником (порівняйте лістинги 6.62 і 6.63 в місці визначення оператора ->).

Вправа 6.27

Реалізуйте параметризований класу NestedSmartPtr, користуючись лістингами підрозділ у 5.5.5.

Шаблон лістингу 6.63 можна конкретизувати іншими класами, які в свою чергу можуть самі бути маніпуляторами.

```
// Визначення маніпулятора першого рівня                      Лістинг 6.64
typedef SmartPtr<Time> SmartPtrTime;
// Визначення маніпулятора другого рівня
typedef NestedSmartPtr<SmartPtr<Time>> NestedPtrTime;
// Визначення маніпулятора третього рівня
typedef NestedSmartPtr<NestedSmartPtr<SmartPtr<Time>>>
   Nested2PtrTime;
```

Виникає свого роду рекурсивне розгортання конкретизацій. Перш ніж продовжити розгляд вкладених маніпуляторів, розглянемо простіші приклади техніки, що дістала назву *часткової спеціалізації* (partial specialization).

Залишилося зробити ще один крок узагальнення, який дозволив би нам вживати одне ім'я, а саме SmartPtr, замість двох. Для цього потрібно

виділити особливий випадок застосування шаблону з іменем SmartPtr на випадок коли його параметр сам виявляється параметризованим класом, заданим за допомогою шаблону.

```
// Параметризований маніпулятор маніпуляторами        Лістинг 6.65
// довільної глибини, часткова спеціалізація шаблону SmartPtr
template <template <typename> class Manipulator, typename Manipulated>
class SmartPtr {
private:
   Manipulator< Manipulated>* _pointee; // Маніпулятор
   SmartPtr (SmartPtr& source); SmartPtr& operator=(SmartPtr&);
public:
// Конструктор, закритий від конвертування типів
   explicit SmartPtr (Manipulator<Manipulated>* pt=0);
   ~SmartPtr() {delete _pointee;} // Деструктор
// Специфічні операції над указниками
// Операція доступу за указником
   const SmartPtr& operator->() const {return *_pointee;}
// Операція розіменування маніпулятора
   Manipulator< Manipulated>& operator*() const {return *_pointee;}
   SmartPtr* release(); // Від'єднання об'єкта
   void reset(Manipulator< Manipulated>* pt =0); // Приєднання об'єкта
};
```

Визначення стають природнішими: SmartPtr<Complex> стане визначенням типу маніпулятора першого рівня, SmartPtr<SmartPtr<Complex>> – типом маніпулятора другого рівня; SmartPtr<SmartPtr<SmartPtr <Complex>>> – третього.

Вправа 6.28

Виконайте тестування вкладених маніпуляторів.

Побудуємо ще одну часткову спеціалізацію шаблону маніпуляторів на випадок маніпулятора масивами. В лістингу 5.209 для цього досить замінити визначення одного класу частковою спеціалізацією іншого. В лістингу 6.66 наведемо його фрагмент

```
// Часткова спеціалізація маніпуляторів на випадок масивів  Лістинг 6.66
template <typename Pointee>
class SmartPtr< Pointee []> {
private:
   size_t _size; // Розмір масиву
   Pointee* _pointee; // Указник на масив
   Pointee* _end; // Указник за масивом
```

```
   mutable Pointee* _current; // Указник поточного елемента масиву
   SmartPtr (SmartPtr&); // Закритий конструктор копіювання
   ArrayHandler& operator=(ArrayHandler&); // (Не)копіювальне присвоєння
public:
   explicit SmartPtr (Pointee* ptr, const size_t size); // Конструктор
   ~ SmartPtr(); // Деструктор
   bool defined() const {return _pointee != 0;} // Перевірка визначеності
                                              // маніпулятора
   size_t size() const {return _size;} // Селектор розміру масиву
// Специфічні операції над указниками
// Операція розіменування маніпулятора
   Pointee& operator*() const {return *_current;}
// Операція доступу за указником
   Pointee* operator->() const{return _current;}
   void start() const {_current=_pointee;} // Почати обробку масиву
   bool stop() const {return _current==_end;} // Перевірити вичерпання
                                            // масиву
// Операції індексування
   Pointee& operator[](const size_t i);
   const Pointee& operator[](const size_t i) const;
};
```

Вправа 6.29

Реалізуйте операції адресної арифметики в маніпуляторі масивів. Виконайте його тестування.

На завершення розглянемо простий приклад обчислень на етапі компіляції шляхом використання часткової спеціалізації шаблонів. Визначимо структуру факторіалів, параметризовану цілим числом

Лістинг 6.67

```
template <unsigned int N> struct Factorial {
   static unsigned int factorial() {
      return N*Factorial<N-1>::factorial();
   }
};
```

Доповнимо її частковою спеціалізацією

Лістинг 6.68

```
template <> struct Factorial<1> {
   static unsigned int factorial() {
      return 1;
   }
};
```

яка відіграватиме роль бази рекурсії. Тепер обчислення факторіалу довільної константи, скажімо, 10 можна організувати у вигляді виклику `Factorial<6>::factorial()`. Всю підготовку – розгортання рекурсії – буде виконано статично на етапі компіляції. Ясна річ, що для цього параметр шаблону має бути сталою.

Вправа 6.30.

Визначте параметризований клас і його часткову спеціалізацію для обчислення чисел Фібоначчі.

Крім констант, статичні функції в часткових спеціалізація можуть оперувати статичними змінними. Як приклад розглянемо задачу підрахунку чисел Фібоначчі лінійною рекурсією з використанням допоміжної статичної пам'яті.

Лістинг 6.69

```
// Масив для зберігання обчислених чисел Фібоначчі
const unsigned int n=10;
unsigned int nFibonacci[n];
// Шаблон структури зі статичною функцією
template <int N> struct QuickFibonacci {
   static void fibonacci() {
      QuickFibonacci<N-1>::fibonacci();
      nFibonacci[N]=nFibonacci[N-2]+nFibonacci[N-1];
      return;
   }
};
// Часткові спеціалізації для бази рекурсії
template <> struct QuickFibonacci<1> {
   static void fibonacci() {
      nFibonacci[1]=1;
      return;
   }
};
template <> struct QuickFibonacci<0> {
   static void fibonacci() {
      nFibonacci[0]=1;
      return;
   }
};
```

Тоді після виклику `QuickFibonacci<n>::fibonacci();` відповідний елемент масиву `nFibonacci[n]` міститиме шукане число Фібоначчі.

Вправа 6.31

Не використовуючи циклів і розгалужень за допомогою часткових спеціалізацій шаблонів, напишіть програму підрахунку і виведення суми цілих чисел, заданих у символьному виді, наприклад, `three+two+one+zero=six`.

Детально стратегії проектування програм, основані на шаблонах, вивчаються у книзі А. Александреску [13].

Завдання до розділу 6

1. Визначте параметризований клас черг `Queue`.
   c. Реалізуйте операцію виведення черги.
   d. Визначте функцію `merge` злиття двох черг за зразком:
   `merge`($a_1, a_2,\ldots, b_1, b_2,\ldots$) = $a_1b_1a_2b_2\ldots$
2. Визначтетареалізуйте на базі черги (задача 1) параметризований клас `TopsyTurvyQueue`, додаткова операція якої `topsyturvy()` перевертає чергу в зворотньому порядку.
3. Визначте і реалізуйте на базі черги (задача 1) параметризований клас черг з підгляданням `PeekbackQueue`. Реалізуйте операцію виведення черги з підгляданням.
4. Визначте клас впорядкованих списків елементів довільного типу. Додатково до стандартних функцій класу реалізуйте функцію пошуку заданого елемента списку та функцію злиття двох впорядкованих списків у один впорядкований список.
5. Визначте клас тетраедрів `Tetrahedron`, передбачивши функції доступу до його складових: граней `Face`, ребер `Edge` і вершин `Vertex`.
6. Визначте параметризований контейнер двосторонніх черг `DEQ`. Конкретизуйте його для зберігання вершин многокутника. Складіть функцію підрахунку периметра многокутника.
7. Визначте клонований ітератор для параметризованих контейнерів із прямим доступом. Застосуйте його для реалізації функції
   e. пошуку діленням навпіл;
   f. сортування злиттям.
8. Визначте і реалізуйте на базі стека параметризований клас `TopsyTurvyStack`, додаткова операція якого `topsyturvy()` перевертає стек.
9. Визначте і реалізуйте на базі односторонньої черги параметризований клас `TopsyTurvyQue`, додаткова операція якого `topsyturvy()` перевертає чергу.

10. Визначте клас NoRepSeqбезповторних послідовностей значень довільного типу. Передбачити операції доступу за індексом, пошуку та додавання елемента.
11. Визначте клас SortedSeq впорядкованих безповторних послідовностей значень довільного типу. Передбачте операції доступу за індексом, пошуку та додавання елемента.
12. Для кожного типу послідовностей із задач 10 і 11 визначте і реалізуйте операції префіксного й постфіксного інкрементів, що є почленними інкрементами кожного елемента.
13. Визначте абстрактний клас Sequence послідовностей з невіртуальним інтерфейсом.
14. Побудуйте ієрархію послідовностей NoRepSeq і SortedSeq зі спільним невіртуальним інтерфейсом Sequence.
15. Чергою з пріоритетами називається черга, що складається з пар: предмет і його пріоритет. Першим вилучається найближчий до початку черги предмет з найвищим пріоритетом. Визначте клас BoundedPriorityQueue обмежених черг з пріоритетами предметів довільного типу.
16. Визначте клас PriorityQueue необмежених черг з пріоритетами предметів довільного типу.
17. Визначте параметризований клас обгорток для арифметичних типів даних, передбачивши в ньому операцію зміни знаку, арифметичні операції, операції інкременту і декременту, введення й виведення.
18. Пріоритетна черга з підгляданням PeekbackPriorityQueue має додатковий метод, який за пріоритетом створює чергу всіх предметів заданого пріоритету, наявних в черзі з пріоритетами. Визначте клас PeekbackPriorityQueue.
19. Визначте та реалізуйтеарифметичні операції, суміщені з присвоєнням, а саме додавання і множення матриць, а також множення матриці на число для класів
    a. цілочисельних квадратних матриць IMatrix;
    b. дійсних квадратних матриць RMatrix;
    c. комплексних квадратних матриць CMatrix;
    d. визначте параметризований клас Matrix, конкретизаціями якого служитимуть матриці класів а–в;
    e. визначте клас MatrCalc калькулятора матриць, що виконує наведені в умові задачі операції над матрицями довільних типів;
    f. визначте клас UniCalc універсального калькулятора, що виконує арифметичні операції над числами й матрицями.

20. Не використовуючи циклів і розгалужень за допомогою часткових спеціалізацій шаблонів, напишіть програму виведення
    a. перших 100 натуральних чисел;
    b. факторіалів перших 10 чисел;
    c. перших 10 степенів двійки.

Навчальні проекти

1. Визначте, реалізуйте та виконайте випробування параметризованого класу рядків, що складатимуться з символів довільного типу, конкретизаціями якого служитимуть `char`, `wchar_t`, форматовані символи, розфарбовані символи, тощо.
2. Визначте, реалізуйте та виконайте випробування параметризованого класу інтелектуальних указників:
   a. монолітних;
   b. осібних.

   Проаналізуйте та порівняйте доступні способи передачі указників параметрами.
3. Визначте, реалізуйте та виконайте випробування параметризованого класу функторів для функцій однієї змінної довільного числового типу. Використайте функтори для табуляції значень функції на заданому відрізку із заданим кроком.
4. Визначте, реалізуйте та виконайте випробування параметризованого класу векторів довільного типу фіксованого розміру, наділивши його операціями додавання, віднімання, множення на скаляр, скалярного множення та порівняння на рівність і нерівність. Передбачте перевірку коректності індексів.
5. Визначте, реалізуйте та виконайте випробування параметризованого класу квадратних матриць заданого порядку, наділивши його операціями додавання, віднімання, множення на скаляр, множення та порівняння на рівність і нерівність. Передбачте перевірку коректності індексів.
6. Визначте, реалізуйте та виконайте випробування параметризованого класу асоціативного списку пар `AssotiativeList` з лістингу 6.49.
7. Визначте, реалізуйте та виконайте випробування параметризованого класу функціональних послідовностей `MappedSequence` з лістингу 6.51.

7. Ієрархічне програмування

Як писав Едсгер Дейкстра в 1975 році [25], програмування належить до найскладніших галузей прикладної математики. Відтоді воно хоч і вийшло за межі прикладної математики, але поєднало в собі елементи інженерних і фундаментальних наук, а отже, стало ще складнішим. Цю думку поділяють не лише студенти-інформатики, готуючись до іспитів із програмування, а й самі розробники програмного забезпечення.

Говорячи про складність програмування, ми не заперечуємо, що є й прості програми, зокрема подібні до розглянутих у перших розділах. Проблеми ж, про які йтиметься тепер, стосуватимуться програмних систем, що за складністю наближаються до так званих промислових програмних продуктів. Їхні властивості – тривалий час життя, багато користувачів і великий колектив розробників.

Проблема складності програмного забезпечення постійно перебуває в центрі уваги вчених і розробників. Фредерік Брукс [26], обґрунтовуючи неможливість створення «срібної кулі» – універсального інструмента, який на порядок підвищив би продуктивність праці програмістів, – твердить, що складність програмного забезпечення криється в природі самих програмних систем. Інший авторитет у галузі програмної інженерії Греді Буч [27] зазначав, що складність програмування спричинена внутрішньою складністю самих програм, яку неможливо усунути, але якою можна керувати, уносячи порядок до хаосу, що складається з мільйонів команд. Неорганізована складність призводить до того, що будь-яка команда в програмі може вплинути на поведінку дуже віддаленої її частини.

Ефективність застосування об'єктної декомпозиції програм можна пояснити обмеженням неконтрольованих впливів усередині програми. Зв'язки всередині об'єкта сильніші, ніж зв'язки поза ним. Цей постулат об'єктного програмування, який підтримує прийнята нами концепція закритих атрибутів, допомагає приховати значну частину складності класу всередині об'єкта, виносячи назовні лише зв'язки з іншими об'єктами.

На фізичному рівні окремого класу складністю керують, поділяючи клас на визначення та реалізацію. Визначення зазвичай розміщують у заголовному файлі, а реалізацію переносять до окремого файлу. Це дає змогу не обтяжувати визначення деталями реалізації там, де вони несуттєві. Особливе місце у визначенні класу належить його відкритій частині, яка містить *інтерфейс* класу, оскільки саме від нього залежить, якими способами клієнти класу використовуватимуть його.

Наступний крок у керуванні складністю переводить нас на рівень взаємодії об'єктів. Потрібно відрізняти динамічну взаємодію, яка може ви-

никати між конкретними об'єктами в певні періоди їхнього життєвого циклу, від сталої взаємодії, властивої всім об'єктам певного класу впродовж усього життєвого циклу. Називатимемо її взаємодією класів. Вона має статичний характер у тому розумінні, що виникає до виконання програми та підтримується впродовж усього часу виконання. Взаємодію як окремих об'єктів, так і цілих класів доцільно організувати ієрархічно – у вигляді ієрархій об'єктів та ієрархій класів. Це дає змогу керувати складністю програмних систем.

7.1. Ієрархія об'єктів

Перший крок у керуванні складністю ми вже, власне, зробили, застосувавши вкладення об'єктів – агрегацію та композицію (див. підрозділ 5.3). Цей тип ієрархії називають *ієрархією об'єктів*.

Об'єктно-орієнтоване програмування за своєю суттю ієрархічне. Уже самі поняття класу й об'єкта можна вважати проявами програмної ієрархії, яка задає підпорядкованість нестатичних членів-даних класу (атрибутів) – своєму об'єкту, а функцій-членів класу (методів), як і статичних членів-даних, – самому класу. Якщо атрибути самі виявляються об'єктами якихось класів, то підпорядкованість атрибутів своєму об'єкту зумовлює ієрархію об'єктів. Саме її ми вивчатимемо тепер, а підпорядкованістю методів класам займемося пізніше.

Отже, розглянемо детальніше ієрархію об'єктів, а саме деякі її властивості, суттєві з погляду вивчення інших типів ієрархій.

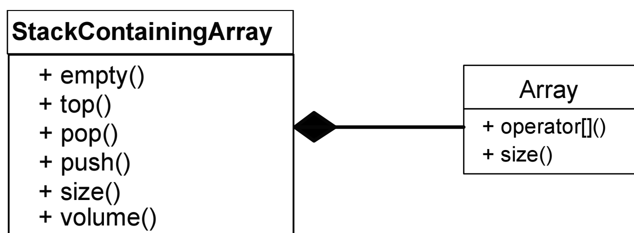


Рис. 7.1. Діаграма класу стеків із вкладеним масивом

Візьмемо пару об'єктів із попереднього розділу, а саме стек і вкладений у нього масив. Із діаграми об'єктів (рис. 7.1) видно, що екземпляр `_stackArray` масиву `Array` – прихована частина об'єкта класу `StackContainingArray`, оскільки відповідно до загальної домовленості щодо прав доступу атрибут міститься в закритій частині класу.

Через це клієнти класу `StackContainingArray` не мають доступу до масиву `_stackArray`. Тому їм недоступна навіть відкрита частина масиву,

наприклад операція індексування. Така організація прав доступу, про що вже йшлося раніше, захищає стек від несанкціонованого втручання в його вміст поза дозволеними операціями доступу.

Наведений далі код класу стеків значною мірою повторює визначення стека з лістингу 6.39. Обробку аварійних ситуацій пропустимо, пропонуючи читачеві самостійно внести відповідні доповнення.

Лістинг 7.1

```
// Стек із вкладеним до нього масивом
template <class Elem>
class StackContainingArray {
private:
   static const size_t _bos; // Дно стека
   size_t _top; // Верхівка стека
   Array<Elem> _stackArray; // Масив для розміщення стека
public:
// Конструктор, закритий від конвертування типів
   explicit StackContainingArray(const size_t size):
      _stackArray(size), _top(_bos) {}
   ~StackContainingArray() {} // Деструктор
   bool empty() const {return _top==_bos;} // Чи порожній стек?
// Чи не переповнений контейнер?
   bool full() const {return _top==_stackArray.size()-1;}
   size_t size() const {return _top+1;} // Видати поточний розмір стека
// Видати ємність контейнера
   size_t volume() const {return _stackArray.size();}
// Керування стеком
// Видати верхівку стека
   const Elem& top() const {return _stackArray[_top];}
   void pop() {_top--;}  // Виштовхнути верхівку стека
   void push(Elem value) {_stackArray[++_top]=value;} // Помістити в стек
};
template <class Elem> const size_t StackContainingArray<Elem>::_bos=-1;
```

Отже, масив `_stackArray` – член стека з тими самими правами, що й інші члени цього класу, зокрема маркер верхівки стека `_top`. Важливо розуміти, що права доступу до об'єкта і його частин залежать від того, де міститься цей об'єкт. Позаяк масив `_stackArray` розміщено в закритій частині класу `StackContainingArray`, то доступ до нього ззовні стека закрито. Нагадаємо визначення класу масивів у трохи скороченому вигляді, знову ж пропускаючи деталі обробки особливих ситуацій.

Лістинг 7.2

```
// Простий масив елементів
template <class ArrayElem>
```

```
class Array {
public:
// Конструктор, закритий для конвертування типів
   explicit Array (const size_t sz): _size(sz),
      _array(new ArrayElem[sz]) {}
// Деструктор
   ~Array() {delete [] _array;}
// Селектор елемента масиву: операція сталого індексування
   const ArrayElem& operator[] (const size_t index) const
      {return _array[index];}
// Селектор-модифікатор елемента масиву: операція індексування
   ArrayElem& operator[] (const size_t index) {return _array[index];}
// Видати розмірність масиву (селектор)
   size_t size() const {return _size;}
private:
   size_t _size;   // Розмірність масиву
   ArrayElem* _array;   // Вміст масиву
// Операції та функції, заборонені для використання в масивах
   Array (const Array&); Array& operator=(const Array& a);
};
```

Зробимо перший висновок щодо ієрархії об’єктів. Композит (у розглядуваному випадку – стек) містить у собі як члена класу об’єкт – екземпляр компонента (тут – масив). Говорячи про відношення між компонентом і композитом, розрізнятимемо право власності та право доступу. Компонент перебуває у власності композита. Утім володіння компонентом не дає композиту виняткових прав доступу до нього. Композит поводиться як звичайний клієнт свого компонента. Права доступу композита стосуються лише відкритої частини компонента. Так, стек має доступ до конструктора й деструктора масиву, операторів індексування та функції обчислення розміру масиву.

Разом із цим композит має права виняткової власності на компонент, приховуючи його від інших об’єктів. Клієнти композита не мають доступу ні до самого компонента, ні навіть до його відкритої частини. Водночас композит може відкрити доступ до свого компонента повністю або частково за допомогою вже відомого засобу делегування. Зокрема, стек делегує право на виконання своєї відкритої функції-члена `volume()` масиву, доручивши це функції-члену масиву `size()`.

Вправа 7.1

Складіть тестову програму та виконайте тестування класу `StackContainingArray`.

Вправа 7.2

Доповніть класи механізмами обробки аварійних ситуацій, зумовлених спробами прочитати верхівку чи виштовхнути елемент порожнього стека, додати елемент до переповненого стека.

Вправа 7.3

За зразком класу стеків із лістингу 7.1 розробіть і реалізуйте параметризований клас `StackAggregatingArray` обмежених стеків, атрибут якого `_pstackArray` був би приєднаним за допомогою указника

```
Array<Elem>* _pstackArray;
```

Порівняйте класи `StackAggregatingArray` і `StackContainingArray` з точки зору можливості розширення ємності стека.

Вправа 7.4

Сформуйте параметризований клас `UnboundedStackOnArray` необмежених стеків на базі масиву.

Вправа 7.5

Розробіть параметризований клас `UnboundedStackOnList` необмежених стеків на базі списку.

Як відомо, вимоги до програм уточнюються та змінюються в процесі їх використання, тому для підтримки працездатності програм доводиться вносити до них зміни та навіть удаватися до перепрограмування. Цю діяльність зазвичай називають *супроводом* (maintenance) програми. Особливо цінують легкі в супроводі програми, легко налаштовувані на *нові застосування* (reuse) без суттєвого перепрограмування. (На думку автора, узвичаєний у літературі переклад терміна «reuse» російською мовою як «повторное использование» не зовсім відповідає суті поняття.)

Припустімо, що в процесі експлуатації виникла потреба доповнити стек засобом підглядання, тобто можливістю прочитати вміст його внутрішньої частини. Цього можна було б досягти за допомогою утиліти `peekback()`, застосувавши допоміжний стек:

Лістинг 7.3

```
// Утиліта підглядання за вмістом стека
template <typename T>
bool peekback(StackContainingArray<T>& s, size_t n, T& res) {
   StackContainingArray<T> ss(n); // Допоміжний стек
   bool found=false;
   for(size_t i=0; i<n; ++i) {   // Цикл підходу до шуканого елемента
```

```
      T x=s.top();
      if(res==x) {
         found=true; // Шуканий елемент знайдено
         break;
      }
   // Перекладаємо проміжний елемент до допоміжного стека
      s.pop();
      ss.push(x);
   }
   while(!ss.empty()) {   // Цикл відновлення вмісту стека
      T x=ss.top();
      ss.pop();
      s.push(x);
   }
   return found;
}
```

Аналізуючи утиліту peekback(), візьмемо до уваги одну очевидну недоладність: стек, до якого ми підглядаємо, реалізовано на базі масиву – структури даних із прямим доступом, а утиліта використовує послідовний доступ. Прямий доступ під час підглядання реалізуємо в новому класі, який назвемо стеком із підгляданням PeekBackStackContainingArray. Значна його частина повторює звичайний стек.

Лістинг 7.4

```
// Стек із підгляданням на базі масиву
template <class Elem>
class PeekBackStackContainingArray {
// Спільна зі стеком частина
private:
   static const size_t _bos;  // Дно стека
   size_t _top;  // Верхівка стека
   Array<Elem> _stackArray;   // Масив для розміщення стека
public:
// Конструктор, закритий від конвертування типів
   explicit PeekBackStackContainingArray (const size_t size):
      _stackArray(size), _top(_bos) {}
   ~PeekBackStackContainingArray() {}  // Деструктор
   bool empty() const {return _top==_bos;}  // Чи не порожній стек?
   // Чи не переповнений контейнер?
   bool full() const {return _top==_stackArray.size()-1;}
   size_t size() const {return _top+1;} // Видати поточний розмір стека
// Видати ємність контейнера
   size_t volume() const {return _stackArray.size();}
```

```
// Керування стеком:
// Видати верхівку стека
   const Elem& top() const {return _stackArray[_top];}
   void pop() {_top--;}  // Виштовхнути верхівку стека
// Помістити в стек
   void push(Elem value) {_stackArray[++_top]=value;}
// Додано функцію підглядання
   bool peekback(const size_t index, Elem& elem) const {
      if (index>_top) return false;
      elem=_stackArray[index];
      return true;
   }
};
```

Вправа 7.6

Складіть тестову програму та протестуйте клас `PeekBackStackContainingArray`.

Проаналізуємо зроблене. На рис. 7.2 показано склад класів і відношення між ними.

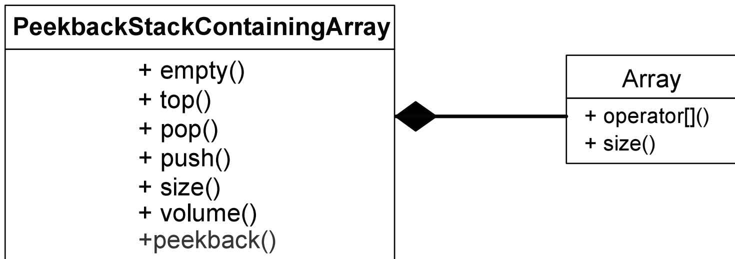


Рис. 7.2. Діаграма класу стеків з підгляданням

Досягненням можна вважати те, що готовий програмний текст класу стеків можна використовувати як значну частину нового класу: власне, до нього дописано одну нову функцію підглядання. Вада ж полягає в дублюванні коду цих двох класів. Зазвичай його вважають небажаним, оскільки дублювання породжує потребу постійно відстежувати відповідність кодів упродовж усього життєвого циклу програмної системи. Дублювання спричиняє ускладнення супроводу, тому що доводиться повторювати кожну правку, унесену до класу `StackContainingArray`, у тій частині `PeekBackStackContainingArray`, яка його повторює. Вихід полягає у використанні об’єктного, а не текстового коду базового класу `StackCo-`

ntainingArray. Це породжує принципово відмінний тип ієрархії – ієрархію класів, до розгляду якої ми перейдемо в наступному підрозділі. Але спочатку зробимо два зауваження.

У такий само спосіб, як і для стека з вбудованим масивом, можна задати стек із підгляданням, створений на агрегованому масиві.

Вправа 7.7

Створіть клас PeekBackStackAggregatingArray стеків із підгляданням, базуючись на класі StackAggregatingArray.

Для стека на базі списків підглядання реалізувати складніше. Звісно, можна скористатися запропонованою вище утилітою. Інший спосіб – включити до стека додаткову структуру даних для реалізації прямого доступу.

Вправа 7.8

Створіть клас PeekBackStackOnList стеків із підгляданням на базі класу UnboundedStackOnList.

7.2. Ієрархія класів: відкрите успадкування

Перейдемо тепер до вивчення найважливішої концепції об'єктно-орієнтованого програмування, яка, власне, і зумовлює виникнення ієрархії класів. Вона одержала назву *успадкування* (inheritance), а точніше – *відкритого успадкування* (public inheritance).

Варто почати з простого прикладу. Візьмемо клас прямокутників Rectangle:

Лістинг 7.5

```
typedef double length_t;    // Тип довжини сторони
// Клас прямокутників
class Rectangle {
private:
   length_t _height; // Висота
   length_t _width; // Ширина
public:
// Конструктор
   Rectangle (const length_t height, const length_t width);
   ~Rectangle(); // Деструктор
   Rectangle (const Rectangle&); // Копіювальний конструктор
   Rectangle& operator=(const Rectangle&); // Присвоєння
// Селектори сторін
   length_t getA() const; length_t getB() const;
```

```
// Модифікатори сторін
   Rectangle& setA(const length_t); Rectangle& setB(const length_t);
// Функції обчислення характеристик:
   length_t area() const; // площі
   length_t diagonal() const; // діагоналі
};
// Операція виведення прямокутника
ostream& operator<<(ostream&, const Rectangle&);
```

Клас Rectangle має звичайний набір членів: закриті атрибути довжини та ширини, зрозумілі конструктори, деструктор, оператор присвоєння, селектори та модифікатори, а також функції обчислення площі та діагоналі прямокутника. Наявність серед членів класу всіх функцій, окрім двох останніх, зрозуміла. Функція обчислення площі могла б бути утилітою – наприклад, із сигнатурою

```
length_t area(const Rectangle&);
```

як і функція обчислення діагоналі:

```
length_t diagonal(const Rectangle&);
```

Трохи пізніше ми проаналізуємо аргументи на користь того чи іншого рішення. Поки що залишимо обидві функції – утиліту й метод, реалізувавши одну через іншу:

Лістинг 7.6

```
// Реалізація утиліт через метод:
// обчислення площі
length_t area(const Rectangle& rec) {return rec.area();}
// обчислення діагоналі
length_t diagonal(const Rectangle& rec) {return rec.diagonal();}
```

Обмежившись лише утилітою, для обчислення площі чи діагоналі довелося б скористатися селекторами довжини та ширини, тоді як методи можуть звертатися до атрибутів напряму (що, можливо, не завжди добре):

Лістинг 7.7

```
// Реалізація функції обчислення площі
length_t Rectangle::area() const {return _height*_width;}
#include <cmath>
// Реалізація функції обчислення діагоналі
length_t Rectangle::diagonal() const {
   return sqrt(_height*_height+_width*_width);
}
```

Реалізація класу очевидна, але наведемо її повністю, щоб мати цілісну картину ієрархії, до якої ми рухаємося:

Лістинг 7.8

```
// Реалізація конструктора
Rectangle::Rectangle(const length_t height, const length_t width):
      _height (height), _width (width) {}
// Реалізація деструктора
Rectangle::~Rectangle() {}
// Реалізація копіювального конструктора
Rectangle::Rectangle (const Rectangle& rec):
      _height (rec._height), _width (rec._width) {}
// Реалізація присвоєння
Rectangle& Rectangle::operator=(const Rectangle& source) {
   _height=source._height; _width=source._width;
   return *this;
}
// Реалізація селекторів
length_t Rectangle::getA() const {return _height;}
length_t Rectangle::getB() const {return _width;}
Rectangle& Rectangle::setA(const length_t a) {_height=a; return *this;}
// Реалізація модифікаторів
Rectangle& Rectangle::setB(const length_t b) {_width=b; return *this;}
// Реалізація операції виведення прямокутника
ostream& operator<<(ostream& os, const Rectangle& rec) {
   os<<rec.getA()<<'x'<<rec.getB();
   return os;
}
```

Тепер виділимо важливий різновид прямокутника – квадрат. Зрозуміло, що кожен квадрат можна розглядати як прямокутник, але не навпаки. Узявши довільний прямокутник `source`, завжди можна перевірити, чи він не квадрат, порівнявши довжини сторін: `source.getA()==source.getB()`. Проте ця перевірка відбуватиметься вже на етапі виконання програми, тоді як ми прагнемо якнайбільше перевірок (передусім щодо типів об'єктів) зробити ще на етапі компіляції.

Властивості, які об'єкти зберігають упродовж усього життєвого циклу, називатимемо *статичними*, а інші, тобто ті, що можуть змінюватися під час виконання програми, – *динамічними*. «Бути прямокутником» – статична властивість, «мати певну довжину сторони» – зазвичай динамічна, оскільки довжину можна змінити, застосувавши відповідний модифікатор. Якщо квадрат розуміти як прямокутник із рівними сторонами, то властивість «бути квадратом» теж стає динамічною. Це добре чи погано?

Проведемо аналогію з числами. Візьмемо, наприклад, цілий і дійсний типи. Властивість належати до них статична. У цілого числа не може

з'явитися дробової частини, тоді як дуже легко одержати дійсне число, дробова частина якого – нуль. Однак від цього воно не змінює свого типу в тому розумінні, що воно зберігається в форматі дійсного числа і до нього й далі застосовуватимуться операції та функції з дійсними параметрами. Їх узагалі-то можна застосувати й до цілого числа, але попередньо перетворивши тип.

Подібно до того, як із дійсних чисел було виділено тип цілих чисел, хотілося б виділити клас квадратів, задавши його як окремий клас, наприклад

Лістинг 7.9

```
// Незалежний клас квадратів
class Square {
private:
   length_t _side; // Сторона
public:
   Square (const length_t); // Конструктор
   ~Square(); // Деструктор
   Square (const Square&); // Копіювальний конструктор
   Square& operator=(const Square&); // Присвоєння
   length_t getA() const; // Селектор сторони
   Square& setA(const length_t); // Модифікатор сторони
// Функції обчислення характеристик:
   length_t area() const; // площі
   length_t diagonal() const; // діагоналі
};
// Операція виведення квадрата
ostream& operator<<(ostream&, const Square&);
```

Однак із такого визначення не видно зв'язку між квадратами та прямокутниками. Щоб він виник, приєднаємо один клас до іншого, застосувавши механізм *відкритого успадкування* (public inheritance), яке полягає у визначенні нового класу квадратів `DerivedSquare` не окремо, а на основі базового класу `Rectangle` як похідного від нього:

Лістинг 7.10

```
// Клас квадратів, похідний від класу прямокутників
class DerivedSquare: public Rectangle {
public:
   DerivedSquare (const length_t side=10); // Конструктор
   ~DerivedSquare() {} // Деструктор
   DerivedSquare (const DerivedSquare&); // Копіювальний конструктор
   DerivedSquare& operator=(const DerivedSquare&); // Присвоєння
};
```

Що це означає? Передусім те, що об'єкт класу квадратів – водночас повноцінний об'єкт класу прямокутників. Заголовок

```
class DerivedSquare: public Rectangle
```

читають так: кожен об'єкт класу `DerivedSquare` є похідним від об'єкта базового класу `Rectangle`. Об'єкт «квадрат» створено з об'єкта «прямокутник», тому квадрат має всі властивості базового класу. При цьому зникає потреба дублювати функції, задані для прямокутників, у класі квадратів. Клієнти класу квадратів одержать доступ не лише до його відкритої частини, але й до відкритої частини базового класу прямокутників. Цей зв'язок однобічний: базовий клас (тут – прямокутник), не має й не може мати інформації про утворені від нього похідні класи.

Говорять, що похідний об'єкт – *уточнення*, чи *спеціалізація*, базового, тоді як базовий об'єкт називають *узагальненням* похідного.

Створення об'єкта будь-якого похідного класу складається з двох кроків. Спочатку працює конструктор базового класу, який створює базовий об'єкт. Після цього виконується тіло конструктора похідного класу. Природно, що для створення квадрата потрібно передбачити коректну ініціалізацію атрибутів прямокутника. Типова помилка може полягати в передаванні конструкторові базового класу неправильних параметрів.

Порівняємо два варіанти конструктора:

Лістинг 7.11

```
// Коректний конструктор квадрата: у списку
// ініціалізації конструктору прямокутника передано довжину сторони
DerivedSquare::DerivedSquare(const length_t side):
                                   Rectangle(side, side) {}
```

і

Лістинг 7.12

```
// Некоректний конструктор квадрата: список
// ініціалізації пропущено; буде створено прямокутник
// із заданими за замовчуванням сторонами
DerivedSquare::DerivedSquare (const length_t side) {}
```

Під час кожного створення об'єкта похідного класу конструктор базового класу виконується *незалежно* від того, чи є базовий клас у списку ініціалізації конструктора. Від його наявності залежить спосіб, яким йому буде передано параметри. Якщо його немає, то система програмування підставить виклик конструктора без параметрів або зі значеннями заданих за замовчуванням параметрів. Оскільки такого конструктора в класі прямокутників немає, спроба компілювати функцію з лістингу 7.12 призведе до синтаксичної помилки. Можна вважати, що нам пощастило: якби в прямокутника були задані за замовчуванням параметри, наприклад

```
// Конструктор
Rectangle (const length_t heigh=10, const length_t width=20);
```

то непомітно для користувача система програмування створила б замість квадрата із заданою під час виклику стороною прямокутник зі сторонами 10 і 20. Ось такий вийшов би «квадрат»!

Аналогічно можна реалізувати копіювальний конструктор квадрата і його оператор присвоєння:

Лістинг 7.13

```
// Реалізація копіювального конструктора квадрата
DerivedSquare::DerivedSquare (const DerivedSquare& source):
   Rectangle(source.getA(), source.getB()) {}
// Реалізація присвоєння квадратів
DerivedSquare& DerivedSquare:: operator=(const) DerivedSquare& source {
   return setA(source.getA());
}
```

У списку його ініціалізації знову бачимо прямокутник, сторони якого взято з оригіналу для копіювання.

До речі, копіювальний конструктор прямокутника можна застосувати до квадрата. Оскільки кожен квадрат – повноцінний прямокутник, його можна використовувати всюди, де передбачено використання прямокутника, зокрема передати як параметр у вигляді відсилки чи указника:

Лістинг 7.14

```
DerivedSquare square(5);
Rectangle copy(square); cout<<"Copied rec: "<<copy<<endl;
```

Результатом виконання стане прямокутник `copy` – точна копія квадрата `square`, який, утім, перестане бути екземпляром класу квадратів. Порівняйте з аналогічною ініціалізацією числових типів:

```
double d(1);
```

Поки що обмежимося цим зауваженням про використання похідних класів замість базових, відклавши детальний розгляд до ознайомлення з динамічним поліморфізмом.

Вправа 7.9

Складіть тестову програму та виконайте тестування класів `Rectangle` і `DerivedSquare`.

Вправа 7.10

a) Доповніть клас квадратів копіювальним конструктором, який перетворюватиме прямокутник з рівними сторонами на відповідний йому квадрат.

b) Реалізуйте метод присвоєння для класу квадратів.

Розглянемо детальніше будову класу квадратів. Як видно з його визначення в лістингу 7.10, власними елементами квадрата є лише конструктори, деструктор і присвоєння. Усе інше квадрат успадковує від базового класу прямокутників. Це означає, що квадрат має дві (але рівні) сторони, виражені атрибутами `_height` і `_width`, селектори та модифікатори цих атрибутів, функції обчислення площі та діагоналі, а також утиліту виведення.

Звісно, успадкування атрибутів і методів звільняє від потреби дублювати код, яке довелося зробити в лістингу 7.9. Проте наслідки успадкування не цілком відповідають нашим уявленням про коректну поведінку квадрата. Справді, подивимося, що станеться в разі спроби застосувати до квадрата модифікатори прямокутника. Виявляється, що при цьому квадрат може бути спотворено:

Лістинг 7.15

```
DerivedSquare square(10); // Спотворення квадрата
cout<<square<<endl; // Показуємо квадрат зі стороною 10
square.setB(20); // Модифікація лише однієї зі сторін
cout<<square<<endl; // Показуємо «дивний квадрат»: одна сторона - 10,
                    // друга - 20
```

Частковий вихід із ситуації некоректної поведінки методів базового класу стосовно об'єктів похідного полягає в тому, що слід відмовитися від певної частини методів базового класу та замінити їх на власні методи похідного класу. Для цього є дві можливості. Перша – перевизначити модифікатори сторін у класі квадратів:

Лістинг 7.16

```
// Квадрат, доповнений власними модифікаторами
class DerivedSquare: public Rectangle {
public:
   DerivedSquare (const length_t side=10); // Конструктор
   ~DerivedSquare() {} // Деструктор
   DerivedSquare (const DerivedSquare&); // Копіювальний конструктор
   DerivedSquare& operator=(const DerivedSquare&); // Присвоєння
// Власні модифікатори сторони квадрата
   DerivedSquare& setA(const length_t);
   DerivedSquare& setB(const length_t);
};
```

реалізувавши кожен із методів так, щоб зі зміною довжини однієї зі сторін базового прямокутника водночас відповідно змінювалася довжина другої сторони, залишаючи його квадратом:

```
// Одночасна модифікація обох сторін прямокутника
DerivedSquare& DerivedSquare::setA(const length_t a) {
// Потрібно замінити довжини обох сторін базового прямокутника
   Rectangle::setB(a); Rectangle::setA(a);
   return *this;
}
DerivedSquare& DerivedSquare::setB(const length_t b) {
   Rectangle::setB(b); Rectangle::setA(b);
   return *this;
}
```

Лістинг 7.17

Зверніть увагу на те, що в тілі методу `setA()` класу квадратів буде викликано модифікатори `setA()` та `setB()` класу прямокутників. Про це свідчить застосування операторів визначення області дії – `Rectangle::setA(a)` та `Rectangle::setB(a)`.

Проте це рішення певною мірою нейтралізує переваги успадкування, призводячи до часткового дублювання коду. Тому привабливішим може видатись інший спосіб – перевизначити лише один із модифікаторів, закривши доступ до іншого:

```
// Квадрат, доповнений власним модифікатором сторони A
// із закритим доступом до модифікатора сторони B
class DerivedSquare: public Rectangle {
public:
   DerivedSquare (const length_t side=10); // Конструктор
   ~DerivedSquare() {} // Деструктор
   DerivedSquare (const DerivedSquare&); // Копіювальний конструктор
   DerivedSquare& operator=(const DerivedSquare&); // Присвоєння
// Власний модифікатор однієї сторони квадрата
   DerivedSquare& setA(const length_t);
private:
   Rectangle::setB; // Закритий доступ до модифікатора сторони B
};
```

Лістинг 7.18

Реалізація залишається такою самою, як її подано в лістингу 7.17. Проте жодне із запропонованих рішень не розв’язує проблеми повністю. У разі як заміни обох модифікаторів (лістинг 7.17), так і закритого доступу до одного з них (лістинг 7.18), у квадрата залишається можливість доступу до модифікатора прямокутника за допомогою застосування оператора визначення області дії (перевірте).

```
// Спотворення квадрата через доступ до
// модифікатора базового прямокутника
DerivedSquare square(10); cout<<square<<endl;
```

Лістинг 7.19

```
// Знову модифікація лише однієї зі сторін
square.Rectangle::setB(20);
// Виводимо «дивний квадрат»: одна сторона - 10, друга - 20
cout<<square<<endl;
```

Виклик модифікатора `setB()`, як і раніше, спотворить квадрат, і його варто було б уважати синтаксично некоректним, адже доступ до методу було закрито. Однак завдяки гнучкості, властивій мові C++, похідний клас може закрити лише безпосередній доступ до відкритого методу базового класу за допомогою власного інтерфейсу, але не здатен заборонити кваліфікований (із застосовуванням кваліфікатора) доступ за допомогою оператора визначення області дії.

Для повного закривання методів базового класу використовують інші прийоми, наприклад закрите успадкування, яке буде розглянуто пізніше. Є також методи побудови ієрархії за допомогою абстрактних інтерфейсних класів, і вони позбавлені проблем, спричинених перевизначенням методів. Про це йтиметься в підрозділі 7.9.

Ми ж продовжимо налаштовувати клас квадратів на якнайкоректніше співіснування з базовим класом.

Якщо модифікатор сторони квадрата потребував скасування відповідного модифікатора прямокутника, то інші методи, наприклад, обчислення площі чи діагоналі, можуть бути коректно використані як базовим прямокутником, так і похідним від нього квадратом. Те саме стосується утиліт.

Лістинг 7.20

```
// Виклик квадратом утиліти й методу прямокутника
DerivedSquare square; cout<<square<<endl;
cout<<square<<" area "<<area(square)<<'('<<square.area()<<')'<<endl;
```

Похідні класи здатні модифікувати код базового класу шляхом заміщення його методів. Це стосується будь-якого методу базового класу. Наприклад, для методів обчислення площі `area()` та діагоналі `diagonal()` у класі квадратів розробник міг би прийняти рішення про ефективнішу реалізацію обчислень характеристик квадрата. Тоді це матиме такий вигляд:

Лістинг 7.21

```
// Квадрат, доповнений методами обчислення
// площі та діагоналі, власним модифікатором сторони A
// й із закритим доступом до модифікатора сторони B
class DerivedSquare: public Rectangle {
public:
   DerivedSquare (const length_t side=10); // Конструктор
   ~DerivedSquare() {}  // Деструктор
```

```
   DerivedSquare (const DerivedSquare&); // Копіювальний конструктор
   DerivedSquare& operator=(const DerivedSquare&); // Присвоєння
// Власний модифікатор однієї сторони квадрата
   DerivedSquare& setA(const length_t);
// Власні функції обчислення характеристик
   length_t area() const; length_t diagonal() const;
private:
   Rectangle::setB; // Закритий доступ до модифікатора сторони B
};
```

Як тільки у визначенні похідного класу з'явиться заміщена функція, безпосереднє використання базової функції буде скасовано. Тому заміщенню потрібно надати власну реалізацію:

Лістинг 7.22

```
// Реалізація функції обчислення площі квадрата
length_t DerivedSquare::area() const {return getA()*getA();}
// Реалізація функції обчислення діагоналі квадрата
const double sqrt2=sqrt(2.); //1.4142135623730950;
length_t DerivedSquare::diagonal() const {return getA()*sqrt2;}
```

Фрагмент програми з лістингу 7.20 тепер виконуватиметься інакше – з використанням утиліти прямокутника, але методу квадрата:

Лістинг 7.23

```
// Виклик квадратом утиліти прямокутників і методу квадратів
DerivedSquare square(10); cout<<square<<endl;
cout<<square<<" area "<<area(square)<<'('<<square.area()<<')'<<endl;
```

Можна заміщити й утиліти, задавши їх окремо з параметрами похідного класу:

Лістинг 7.24

```
// Реалізація утиліти обчислення площі квадрата
length_t area(const DerivedSquare& square) {
   return square.getA()*square.getA();
}
// Реалізація утиліти обчислення діагоналі квадрата
length_t diagonal(const DerivedSquare& square) {
   return square.getA()*sqrt2;
}
```

Іще раз наведемо попередній фрагмент коду з лістингу 7.20. Тепер як метод, так утиліта вибиратимуться з класу квадратів:

Лістинг 7.25

```
// Виклик квадратом власних утиліти й методу
DerivedSquare square(10); cout<<square<<endl;
cout<<square<<" area "<<area(square)<<'('<<square.area()<<')'<<endl;
```

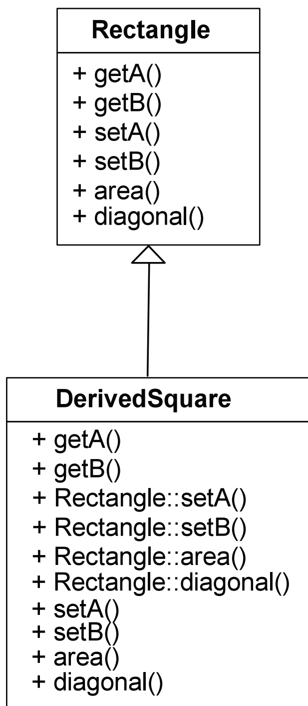


Рис. 7.3. Діаграма ієрархії класів прямокутників і квадратів

Заміщення методу базового класу власною реалізацією в похідному класі не зумовлює повної відмови від методу базового класу. Він залишається доступним для похідного класу, утім із застосуванням операції визначення області дії. Як ви бачили, визначення області дії дає змогу похідному класу не лише використовувати заміщений метод, але й відкрити доступ до методу, відкритого в базовому класі, але закритого в похідному.

Відношення двох розглянутих класів проілюстровано на рис. 7.3.

Із нього видно, що до об'єктів похідного класу можна безпосередньо застосовувати методи базового класу, якщо вони не конфліктують за іменами. У розглядaному випадку це селектори `getA()` і `getB()`. У разі наявності однойменних методів виклики методів похідного класу записуються звичним способом, а виклики методів базового класу потребують додаткового кваліфікатора.

Отже, відкрите успадкування дає похідному класу змогу повністю успадкувати поведінку базового, розширити її додатковими власними функціональними можливостями, у разі потреби уточнюючи або навіть скасовуючи поведінку (тобто методи) базового класу. Після детального вивчення можливостей успадкування ми дамо рекомендації щодо їх застосування.

Вправа 7.11

Задайте клас паралелограмів. Побудуйте на його основі ієрархію класів паралелограмів і прямокутників.

7.3. Ієрархія класів: типізація об'єктів

Коли не було ієрархії класів, поняття класу й типу перебували в однозначній відповідності: кожному класу відповідав рівно один тип, визначений цим класом. Якщо не брати до уваги наявності так званих *простих*

(старих) типів (plane old type), то кожен програмований тип задано якимось класом. Ієрархія класів (точніше – відкрите успадкування) зумовлює порушення цієї взаємної однозначності типів і класів. Тепер кожному похідному класу можна приписати принаймні два типи: його власний і визначений базовим класом. Така неоднозначність потребуватиме додаткових уточнень під час виклику функцій, які допускають заміщення в похідних класах.

Розглянемо детальніше деякі аспекти заміщення та визначимо його межі. Важливо розрізняти заміщення функцій і заміщення об'єктів, схоже на перетворення типів. Почнемо з останнього.

Як відомо з підрозділу 4.9, перетворення типів виконує конвертор. Ним може бути конструктор чи оператор перетворення типу. Зазвичай ці перетворення неявні, але вони завжди зумовлюють створення нового об'єкта потрібного типу, використовуючи початковий об'єкт як параметр. Якихось особливих обмежень на перетворення типів немає. Важливо лише, щоб було виконано одну з умов: або цільовий клас містить конструктор із параметром початкового класу, або початковий клас містить оператор перетворення свого об'єкта в об'єкт цільового класу.

Перетворення, які ми розглядали досі, не стосувались ієрархії. Так ми перетворювали дійсне число на комплексне чи об'єкт, що виражав довжину в британських мірах, в об'єкт у давньоруських. Наявність ієрархії класів надає поняттю перетворення типів нового змісту. Як щойно було продемонстровано, кожен квадрат водночас можна вважати прямокутником, тобто скрізь, де маємо прямокутник, дозволено використовувати квадрат, але не навпаки. Так само прямокутник або квадрат можна використати замість паралелограма, про що йшлось у вправі 7.11. Отже, використання успадкувань надає додатковий вимір для перетворень типів угору за ієрархією успадкувань, від похідного типу до базового. Тому кожен об'єкт похідного класу має багато типів, оскільки належить кожному зі своїх базових класів, яких може бути кілька.

У зв'язку з цим виникають дві можливості: перетворити об'єкт похідного типу до базового типу чи використовувати похідний об'єкт як базовий. Останнє завжди можливо, адже кожен похідний об'єкт містить у собі екземпляр базового, доповнений властивостями похідного. Застосування тієї чи іншої можливості залежить від способу доступу до об'єкта: одержуючи його за допомогою імені, указника чи відсилки, ми надаємо об'єкту тип відповідного іменування.

Розглянемо це детальніше. У разі визначення звичайного імені екземпляру класу, яке передбачає виклик конструктора, цим конструктором

буде створено об'єкт саме того типу, який приписано імені, що визначається. Конструктор, який це робить, однозначно визначається типом імені. Так, визначення

```
Rectangle rec(10, 20);
```

зумовлює створення прямокутника, а

```
DerivedSquare square(30);
```

створює квадрат.

Для створення нового об'єкта можна використати копіювальний конструктор. Так, визначення імені

```
Rectangle copyrec(rec);
```

створює копію `copyrec` прямокутника `rec`. Водночас визначення

```
Rectangle copysquare(square);
```

спричинить створення нового об'єкта перетворенням квадрата `square` на прямокутник `copysquare`.

Визначення ж відсилки чи указника не приводять до створення нових об'єктів. Відсилку буде пов'язано з якимось наявним об'єктом. Наприклад, визначення

```
Rectangle& refrec(rec);
```

зумовить появу альтернативного імені `refrec` для вже наявного прямокутника `rec`. Так само після виконання команди

```
Rectangle& refrecInSquare(square);
```

відсилка `refrecInSquare` стане альтернативним ім'ям для прямокутника, що є базовою частиною квадрата `square`.

Те саме стосується указників. Указником на базовий клас можна ідентифікувати як окремий об'єкт базового класу, так і базову частину об'єкта похідного класу:

Лістинг 7.26

```
Rectangle* ptrrec(&rec);
Rectangle* ptrrecInSquare(&square);
```

Головна відмінність опосередкованого іменування за допомогою відсилок або указників порівняно з прямим іменуванням полягає в тому, що один і той самий об'єкт без будь-яких перетворень можна іменувати іменами кількох підхожих типів. Крім власного типу об'єкта підхожим для

нього буде будь-який базовий тип в ієрархії успадкувань. Це називають доступом до одного й того самого об'єкта за допомогою різних інтерфейсів, кожен з яких задано відповідним класом. Можна сказати, наприклад, що відсилка `refrecInSquare`, як і указник `ptrrecInSquare`, використовують для доступу до квадрата `square` інтерфейс прямокутника, оскільки їх було задано як відсилку й указник на прямокутник.

Розглянемо простий ілюстративний приклад. Візьмемо клас банківських рахунків зі скромним набором атрибутів: номер рахунка, його власник і залишок на рахунку:

Лістинг 7.27

```
// Банківський рахунок
class Account {
private:
   static unsigned int _freeID; // Поточний вільний номер рахунка
   const unsigned int _number; // Власний номер рахунка
   int _balance; // Залишок коштів на рахунку
   string _owner; // Власник рахунка
public:
// Конструктор рахунка
   Account(const string owner, const unsigned int balance):
      _owner(owner), _balance(balance), _number(++_freeID) {}
   ~Account() {} // Деструктор рахунка
   string getName() const {return _owner;} // Селектор власника рахунка
   int getBalance() const {return _balance;} // Селектор балансу
   unsigned int getNumber() const {return _number;} // Селектор номера
                                                    // рахунка
   void setBalance(int balance) {_balance=balance;} // Модифікатор
                                                    // балансу
   int state() {return getBalance();} // Обчислення стану рахунка
};
```

Найважливішою для цього прикладу стане функція стану рахунка `state()`, яка в найпростішому випадку просто показує залишок коштів на рахунку, але в принципі може бути складнішою, нараховуючи преміальні різного типу.

Далі визначимо похідний клас ощадних рахунків. Він характеризуватиметься значенням запроваджених у банку ощадних відсотків `_savingRate` і розміром прибутку за відсотками `_interest`.

Лістинг 7.28

```
// Ощадний рахунок: спеціалізація банківського рахунка
class SavingAccount: public Account {
private:
```

```
   unsigned int _savingRate; // Норма банківського відсотка
   unsigned int _interest; // Прибуток за відсотками
public:
// Конструктор
   SavingAccount(const string owner, const unsigned int balance,
   unsigned int savingRate):
             Account(owner, balance), _savingRate(savingRate) {}
   ~SavingAccount() {}    // Деструктор
// Селектор банківського відсотка
   unsigned int getSavingRate() const {return _savingRate;}
// Модифікатор прибутку за відсотками
   void setInterest(const int interest) {_interest=interest;}
// Селектор прибутку за відсотками
   int getInterest() const {return _interest;}
// Обчислення стану рахунка
   int state() {
     setInterest(getBalance()*static_cast<double>(getSavingRate())/100);
     return Account::state()+getInterest();
   }
};
```

Тепер функція стану рахунка `state()` даватиме залишок, збільшений на розмір прибутку за відсотками. Відбулося заміщення функції `state()` із класу рахунків власною функцією. Однак при цьому сама функція базового класу нікуди не поділася: щоб звернутися до неї, достатньо лише зайти до відповідного простору. Це можна зробити за допомогою оператора визначення області дії, як показано в тексті програми (`Account::state()`), або ж використати імена з областю значень у базовому класі, як це буде зроблено далі.

І, нарешті, третій тип рахунка – бонусний ощадний. На ньому крім відсотків передбачено нарахування фіксованого бонусу.

Лістинг 7.29

```
// Бонусний ощадний рахунок: спеціалізація
// банківського рахунка
class BonusSavingAccount: public SavingAccount {
private:
   unsigned int _bonus; // Особливий бонус
public:
// Конструктор
   BonusSavingAccount(const string owner, int balance,
         const unsigned int savingRate, const unsigned int bonus):
      SavingAccount(owner, balance, savingRate), _bonus(bonus) {}
   ~BonusSavingAccount() {} // Деструктор
// Селектор бонусу
   unsigned int getBonus() const {return _bonus;}
```

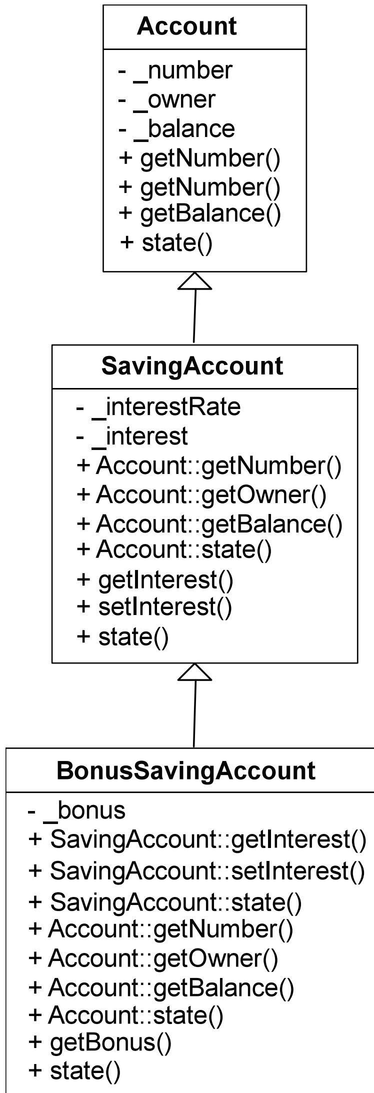


Рис. 7.4. Діаграма класів рахунків

```
// Обчислення стану рахунка
   int state()
      {return SavingAccount::state()+
         getBonus();}
};
```

Функція стану рахунка тепер додає до залишку та відсотків ще й бонус.

Конструктори та деструктори посідають особливе місце в ієрархії класів: вони не можуть бути заміщені. Під час кожного створення похідного об’єкта відпрацьовує весь ланцюжок конструкторів у напрямі, протилежному до успадкування. Це означає, що першим буде створено базовий об’єкт, потім – похідний об’єкт першого рівня, за ним – другого й так далі. Такий порядок виконання конструкторів цілком природний: адже базовий клас нічого не знає про свої похідні класи, тоді як похідний об’єкт може скористатися базовим уже безпосередньо в конструкторі.

Вправа 7.12

Доповніть конструктори та деструктори класів рахунків операціями виведення і простежте за послідовністю їх виконання. Ось приклад, що демонструє використання базового об’єкта в процесі створення похідного:

Лістинг 7.30

```
SavingAccount::
SavingAccount(const string owner,
   const unsigned int balance,
   const unsigned int savingRate):
   Account(owner, balance),
      _savingRate(savingRate) {
         cout<<"A saving account:"<<
         getNumber()<<endl;
}
```

Вправа 7.13

Доповніть клас ощадних рахунків `SavingAccount` нарахуванням за відкриття рахунка одноразового бонусу, що нараховується на баланс базового рахунка `Account`.

Вправа 7.14

Доповніть визначення класів `Account` деструкторами та проаналізуйте порядок їх викликів.

Структуру успадкувань зображено на рис. 7.4. Із нього видно, що ощадний рахунок має всі властивості звичайного і додає до них власні. Так само бонусний ощадний рахунок доповнює ощадний власним атрибутом і методом.

Розглянемо приклад, що демонструє способи вибору типу об'єкта в разі доступу до нього за допомогою указника. Тип об'єкта й інтерфейс доступу до нього визначаються під час компіляції програми.

Лістинг 7.31

```
// Тестова програма для банківських рахунків
int main () {
// 1. Створимо базовий рахунок
   Account ac("Ivanenko", 100);
   Account* pAcc=&ac;   // Непрямий доступ до базового рахунка
   cout<<pAcc->getNumber()<<':'<<pAcc->state()<<endl;
// 2. Створимо ощадний рахунок
   SavingAccount save("Petrenko", 200, 10);
   cout<<save.getNumber()<<':'<<save.state()<<endl;
// Доступ до ощадного рахунка за допомогою інтерфейсу базового рахунка
   pAcc=&save;
   cout<<pAcc->getNumber()<<':'<<pAcc->state()<<endl;
// 3. Створимо бонусний рахунок
   BonusSavingAccount bonus("Pavlenko", 300, 20, 100);
   cout<<bonus.getNumber()<<':'<<bonus.state()<<endl;
// Доступ до бонусного рахунка за допомогою інтерфейсу ощадного рахунка
   SavingAccount* pSave=&bonus;
   cout<<pSave->getNumber()<<':'<<pSave->state()<<endl;
// Доступ до бонусного рахунка за допомогою інтерфейсу базового рахунка
   pAcc=&bonus;
   cout<<pAcc->getNumber()<<':'<<pAcc->state()<<endl;
}
```

На першому кроці створено об'єкт базового класу. Установлений на нього указник такого ж типу не додає нічого нового.

На другому кроці буде створено об'єкт `save` похідного класу `SavingAccount`. Доступ до нього можна організувати за допомогою указника базового типу `pAcc` присвоєнням `pAcc=&save`, але безпосереднє виведення рахунка `save` і виведення за допомогою указника будуть принципово відрізнятися. У першому випадку відбудеться виклик функції `save.state()` похідного класу та нарахування відсотків, а в другому буде викликано функцію базового класу `pAcc->state()`, оскільки застосовано його інтерфейс.

Третій крок – створення бонусного рахунка – проаналізуйте самостійно.

Як висновок зазначимо, що указник базового класу можна встановити на об'єкт будь-якого похідного від нього класу. При цьому не відбувається перетворення типу, але указник має доступ лише до свого базового об'єкта, а всі похідні залишаються для нього недоступними.

Те саме можна сказати й про відсилки. Як приклад розглянемо функцію виведення рахунка

Лістинг 7.32

```
ostream& operator<<(ostream& os, const Account& ac) {
   os<<ac.getName()<<', '<<ac.getNumber()<<endl;
   return os;
}
```

Другий її параметр може набувати значення як базового рахунка, так і ощадного чи бонусного.

Лістинг 7.33

```
Account ac("Ivanenko", 100); // Базовий рахунок
cout<<ac; // Виведення базового рахунка
SavingAccount save("Petrenko", 200, 10); // Ощадний рахунок
cout<<save; // Виведення ощадного рахунка як базового
BonusSavingAccount bonus("Pavlenko", 300, 20, 100); // Бонусний рахунок
cout<<bonus; // Виведення бонусного рахунка як базового
```

Як і раніше, відсилці базового типу доступні лише засоби базового класу.

Властивість об'єктів бути доступними за допомогою різних інтерфейсів або, інакше кажучи, можливість використовувати об'єкти одного типу замість об'єктів іншого типу – чи не найважливіша риса об'єктно-орієнтованого програмування. Однак застосувати її слід особливо акуратно й обережно.

Тепер проаналізуємо проблеми, що виникають у разі застосування різних інтерфейсів до імен. Якщо відсилку при її визначенні пов'язують з уже наявним об'єктом, і тому для її значення не потрібно відводити окремої пам'яті, то визначення імені спричиняє створення нового

об'єкта. Природно, що визначення імені базового типу зумовлює виділення місця, потрібного саме для такого об'єкта. У наведеному далі прикладі один за іншим буде створено об'єкти базового та двох похідних типів. Після застосування до похідних об'єктів конструкторів копіювання базового типу вони перетворяться на об'єкти базових типів.

Лістинг 7.34

```
// 1. Створюємо базовий рахунок: ім'я ас має тип Account
   Account ac("Ivanenko", 100);
// 2. Створюємо ощадний рахунок: ім'я save має тип SavingAccount
   SavingAccount save("Petrenko", 200, 10);
// Забутливий копіювальний конструктор утрачає властивості
// ощадного рахунка: ім'я forgetful має тип Account і копіється
// лише базова частина save
   Account forgetful(save);
// 3. Створюємо бонусний рахунок
   BonusSavingAccount bonus("Pavlenko", 300, 20, 100);
// Забутливе присвоєння втрачає властивості бонусного рахунка: ім'я
// forgetful має тип Account і копіється лише базова частина bonus
   forgetful=bonus;
```

Копіювальний конструктор, застосований до об'єкта похідного типу під час створення базового, називають *забутливим* (forgetful). Так само поводяться копіювальні присвоєння імені базового класу об'єктові похідного класу, оскільки воно призводить до втрати всієї власної частини похідного об'єкта.

Проаналізуємо, як забутливість узгоджується з існуванням стандартних (згенерованих системою програмування) конструкторів і присвоєнь, а також програмованих (визначених розробником) класів.

Розглянемо можливі варіанти. Спочатку візьмемо клас `ShadowMakerNoAssign`, вільний від копіювального конструктора та програмованого присвоєння. Це означає, що до його об'єктів буде застосовано стандартні засоби. Наявність серед атрибутів класу указника дозволить стежити за коректністю поведінки його об'єктів. Об'єкт вважатимемо коректним, якщо його указник `_pChar` встановлено на власний атрибут `_myChar`.

Лістинг 7.35

```
// Базовий клас без власного програмованого присвоєння
class ShadowMakerNoAssign {
private:
   char _myChar; char* _pChar;
public:
   ShadowMakerNoAssign(const char c): _myChar(c), _pChar(&_myChar) {}
```

```
    const char& getChar() const {return _myChar;}
    void* const getPChar() const {return _pChar;}
};
```

Визначимо два об'єкти nshma й nshmb:

```
ShadowMakerNoAssign nshma('a'), nshmb('b');
```

Для спостереження за станом об'єктів цього класу запрограмуємо оператор виведення:

Лістинг 7.36

```
ostream& operator<<(ostream& os,
        const ShadowMakerNoAssign& shm) {
    os<<&shm<<':'<<shm.getChar()<<hex<<', '<<shm.getPChar()<<endl;
    return os;
}
```

Проаналізуємо конструктор класу ShadowMakerNoAssign. Як бачимо, він установлює указник, заданий у другому атрибуті, на значення свого першого атрибута. Результат виконання конструкторів зображено на рис. 7.5.

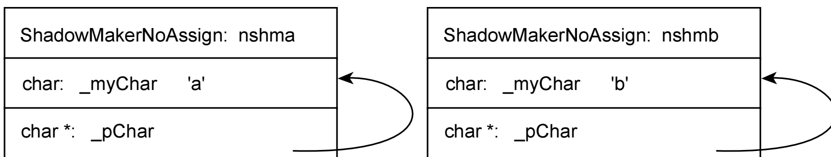


Рис. 7.5. Діаграма об'єктів nshma й nshmb до присвоєння

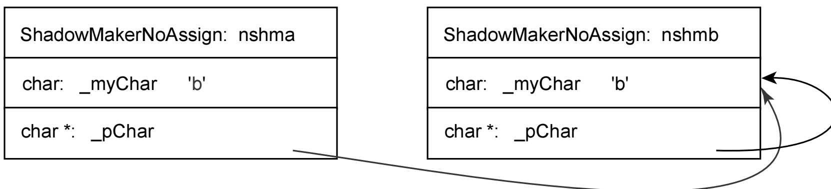


Рис. 7.6. Діаграма об'єктів nshma й nshmb після присвоєння: некоректний результат

Виконаємо присвоєння

```
nshma=nshmb;
```

Його результат, власне, не зовсім коректний, як бачимо з рис. 7.6. Символьний атрибут _myChar в об'єкті nshma присвоєно коректно за допо-

могою почленного присвоєння, проте його указник _pChar показує тепер на чужий атрибут замість власного.

Тепер визначимо клас ShadowMakerDerivedNoAssign, похідний від класу ShadowMakerNoAssign, з тією ж властивістю атрибутів і знову ж із виконаним за замочуванням присвоєнням:

Лістинг 7.37

```
// Похідний клас без власного програмованого присвоєння
class ShadowMakerDerivedNoAssign: public ShadowMakerNoAssign {
private:
   int _myInt; int* _pInt;
public:
   ShadowMakerDerivedNoAssign(const char c, const int i):
      ShadowMakerNoAssign(c), _myInt(i), _pInt(&_myInt) {}
   const int& getInt() const {return _myInt;}
   void* const getPInt() const {return _pInt;}
};
```

Розглянемо тепер два об'єкти похідного класу

```
ShadowMakerDerivedNoAssign dnshma('a', 1), dnshmb('b', 2);
```

зображені на рис. 7.7.

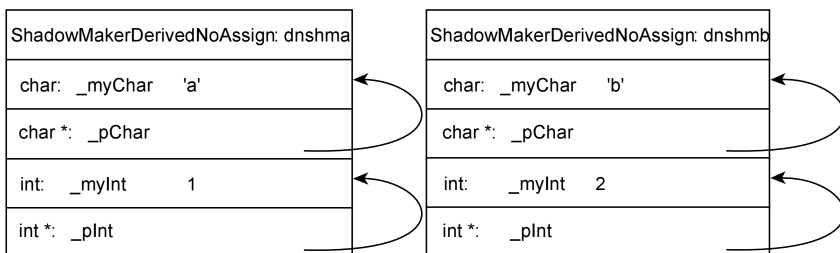


Рис. 7.7. Діаграма об'єктів dnshma *й* dnshmb *до присвоєння*

Іще раз виконаємо присвоєння – тепер похідних об'єктів:

```
dnshma=dnshmb;
```

Як бачимо з рис. 7.8, воно торкнулося як базової, так і похідної частини об'єкта некоректно, як і в попередньому випадку для базового класу.

Вправа 7.15

Задайте оператор виведення для похідного класу та протестуйте виконання присвоєння.

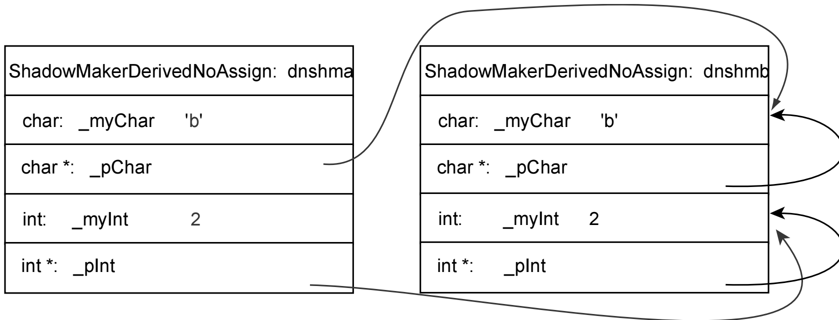


Рис. 7.8. Діаграма об'єктів `dnshma` *й* `dnshmb` *після присвоєння: некоректний результат*

Власне, поки що тут немає нічого нового: зрозуміло, що почленне присвоєння указників змістовно некоректне. Подивимося, як поводитимуться схожі класи, коли в них задано програмоване присвоєння. Розглянемо базовий клас `ShadowMakerAssign`, наділений власним присвоєнням:

Лістинг 7.38

```
// Базовий клас із власним програмованим присвоєнням
class ShadowMakerAssign {
private:
   char _myChar; char* _pChar;
public:
   ShadowMakerAssign(const char c): _myChar(c), _pChar(&_myChar) {}
// Програмоване присвоєння
   ShadowMakerAssign& operator= (const ShadowMakerAssign& tar) {
      _myChar=tar._myChar;
      return *this;
   }
   const char& getChar() const {return _myChar;}
   void* const getPChar() const {return _pChar;}
};
```

Візьмемо два його об'єкти й виконаємо присвоєння:

Лістинг 7.39

```
ShadowMakerAssign shma('a'), shmb('b');
shma=shmb;
```

Після цього матимемо ситуацію, проілюстровану на рис. 7.9. Як бачимо, не лише змінилося значення символьного атрибута `_myChar` в об'єкті `shma`, але й указник `_pChar` указує на атрибут у власному, а не чужому об'єкті.

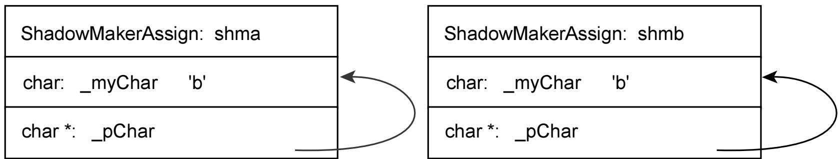


Рис. 7.9. Діаграма об'єктів `shma` й `shmb` після присвоєння: коректний результат

Похідний клас `ShadowMakerDerivedNoAssign` поки що залишимо без програмованого присвоєння.

Лістинг 7.40

```
// Похідний від класу з програмованим присвоєнням
// клас без власного програмованого присвоєння
class ShadowMakerDerivedNoAssign: public ShadowMakerAssign {
private:
   int _myInt; int* _pInt;
public:
   ShadowMakerDerivedNoAssign(const char c, const int i):
      ShadowMakerAssign(c), _myInt(i), _pInt(&_myInt) {}
   const int& getInt() const {return _myInt;}
   void* const getPInt() const {return _pInt;}
};
```

Тепер присвоєння в похідному класі виконуватиметься за тим самим принципом, але з іншим результатом, порівняно з рис. 7.8: спочатку задане за умовчанням присвоєння в похідному класі, а потім – програмоване в базовому.

Лістинг 7.41

```
ShadowMakerDerivedNoAssign dnshma('a', 1), dnshmb('b', 2);
dnshma=dnshmb;
```

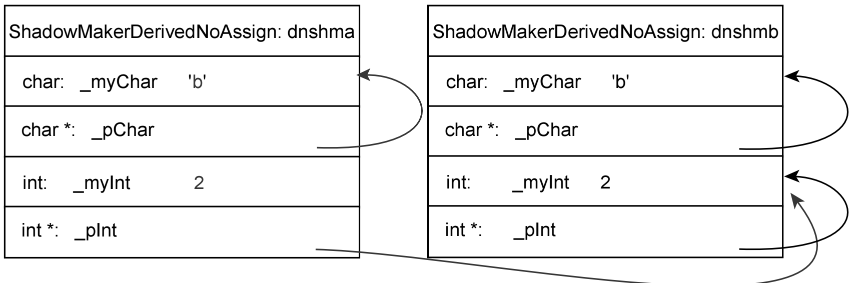


Рис. 7.10. Діаграма об'єктів `dnshma` й `dnshmb` після присвоєння: коректний результат в базовому об'єкті та некоректний в похідному

Результат бачимо на рис. 7.10. Базовий об'єкт присвоєно коректно, а похідну частину – усе ще за правилом стандартного, а тому почленного присвоєння.

Цікаво, що просто визначивши програмоване присвоєння в похідному класі, проблемі не зарадити. Якщо програмоване присвоєння задано в похідному класі, то відбудеться заміщення присвоєння з базового класу присвоєнням похідного: усю відповідальність буде перекладено на нього, і справа не дійде до присвоєння в базовому класі.

Лістинг 7.42

```
// Похідний клас із власним присвоєнням
class ShadowMakerDerivedAssign: public ShadowMakerAssign {
private:
   int _myInt; int* _pInt;
public:
   ShadowMakerDerivedAssign(const char c, const int i):
      ShadowMakerAssign(c), _myInt(i), _pInt(&_myInt) {}
// Програмоване присвоєння
   ShadowMakerDerivedAssign& operator=
         (const ShadowMakerDerivedAssign& tar) {
      cout<<"derived assignment"<<endl; _myInt=tar._myInt;
      return *this;
   }
   const int& getInt() const {return _myInt;}
   void* const getPInt() const {return _pInt;}
};
```

Знову розглянемо схожий приклад:

Лістинг 7.43

```
ShadowMakerDerivedAssign dnshma('a', 1), dnshmb('b', 2);
dnshma=dnshmb;
```

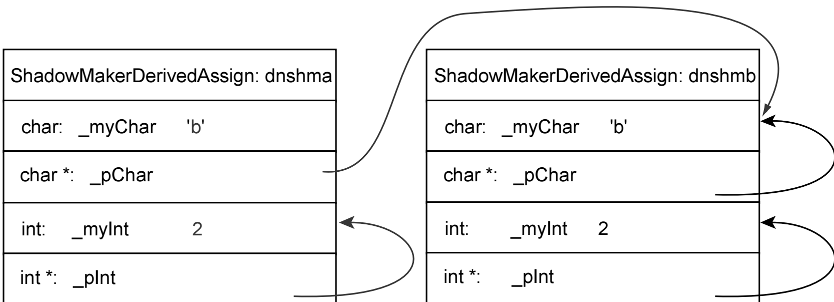


Рис. 7.11. Діаграма об'єктів `dnshma` *й* `dnshmb` *після присвоєння: некоректний результат в базовому об'єкті та коректний в похідному*

Одержаний результат відображено на рис. 7.11. Похідний клас використовує власне програмоване присвоєння, яке «затінило» програмоване присвоєння з базового класу. Присвоєння базового об'єкта знову відбулось як задане за умовчанням.

Нарешті, запишемо повне розв'язання проблеми, у якому задано коректне присвоєння як у базовому, так і в похідному класах:

Лістинг 7.44

```
// Узгоджене присвоєння похідного та базового класів
class ShadowMakerDerivedAssign: public ShadowMakerAssign {
private:
   int _myInt; int* _pInt;
public:
   ShadowMakerDerivedAssign(const char c, const int i):
      ShadowMakerAssign(c), _myInt(i), _pInt(&_myInt) {}
// Програмоване присвоєння похідного класу
   ShadowMakerDerivedAssign& operator=
         (const ShadowMakerDerivedAssign& tar) {
// Виклик програмованого присвоєння базового класу
      this->ShadowMakerAssign::operator=(tar);
      cout<<"derived assignment"<<endl; _myInt=tar._myInt;
      return *this;
   }
   const int& getInt() const {return _myInt;}
   void* const getPInt() const {return _pInt;}
};
```

Тепер у базовому класі явно викликатиметься потрібне присвоєння, задане в ньому сигнатурою

```
ShadowMakerAssign& ShadowMakerAssign::operator=
   (const ShadowMakerAssign& tar);
```

Отримуємо правильний результат (рис. 7.12).

Отже, стає повністю зрозумілою природа забутливого присвоєння: виконуючи присвоєння значення імені об'єкта базового класу, буде взято присвоєння, наявне у базовому класі, навіть якщо джерелом присвоєнням служить об'єкт похідного класу. У цьому випадку частину об'єкта, що не входить до базового класу, буде втрачено (забуто). За місцем призначення до лівої частини присвоєння потрапить лише об'єкт базового класу. Така поведінка цілком зрозуміла: адже за місцем розміщення значення імені об'єкта базового класу не передбачено пам'яті для зберігання атрибутів його похідних класів.

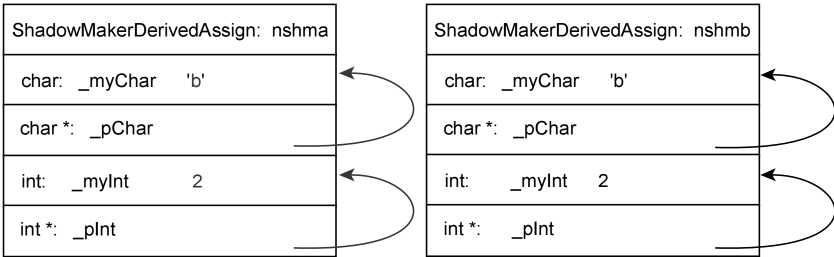


Рис. 7.12. Діаграма об'єктів `dnshma` *й* `dnshmb` *після присвоєння: повністю коректний результат в базовому і похідному об'єктах*

Як висновок скажемо, що, використовуючи ієрархію класів, слід або ***повністю заборонити присвоєння, поміщаючи його в закритій частині, або програмувати в кожному похідному класі всю ієрархію присвоєнь***.

7.4. Уточнення привілеїв доступу до членів базового класу

Поділ класу на відкриту й закриту частини не надає похідному класу жодних привілеїв стосовно доступу до базового. Проте за своїм становищем похідні класи якогось базового класу мали б відрізнятися від усіх інших класів, які його використовують. Бажання наділити похідний клас особливими правами доступу до певної частини базового класу зумовило появу ще однієї (окрім відкритої та закритої) частини класу, яку назвали *захищеною* (protected).

Повернімося до класу прямокутників і змінимо статус його атрибутів із закритого на захищений.

Лістинг 7.45

```
// Тип довжини сторін
typedef double length_t;
// Клас прямокутників із захищеними сторонами
class Rectangle {
// Захищена частина класу
protected:
   length_t _height; // Висота
   length_t _width; // Ширина
public:
   Rectangle (const length_t heigh, const length_t width); // Конструктор
   ~Rectangle(); // Деструктор
```

```
   Rectangle (const Rectangle&); // Копіювальний конструктор
   Rectangle& operator=(const Rectangle&); // Присвоєння
// Селектори сторін
   length_t getA() const;
   length_t getB() const;
// Модифікатори сторін
   Rectangle& setA(const length_t); Rectangle& setB(const length_t);
// Функції обчислення характеристик:
   length_t area() const; // площі
   length_t diagonal() const; // діагоналі
};
// Операція виведення прямокутника
ostream& operator<<(ostream&, const Rectangle&);
```

Захищена частина недоступна нікому ззовні класу, крім його похідних класів. Тепер можна переписати реалізацію методів квадрата, скориставшись замість селекторів прямим доступом до атрибутів прямокутника, що не так вже й добре. Саме визначення класу квадратів залишається без змін таким, як у лістингу 7.10.

Лістинг 7.46

```
// Реалізація копіювального конструктора квадрата
DerivedSquare::DerivedSquare (const DerivedSquare& source):
// Замість Rectangle(source.getA(), source.getB())
   Rectangle(source._height, source._width) {}
// Реалізація функції обчислення площі квадрата
length_t DerivedSquare::area() const {
   return _height*_height; // Замість return getA()*getA();
}
// Реалізація функції обчислення діагоналі квадрата
const double sqrt2=sqrt(2.); // 1.4142135623730950;
length_t DerivedSquare::diagonal() const {
   return _height*sqrt2; // Замість return getA()*sqrt2;
}
```

Ззовні похідного класу захищена частина базового поводить як закрита. Тому утиліти обчислення площі та діагоналі залишаються незмінними. Вони, як і раніше, використовуватимуть селектори. Для перевірки транзитивності доступу до захищеної частини класу вздовж ланцюжка успадкувань достатньо пересвідчитися в тому, що клас, похідний від класу квадратів, зберігає за собою права доступу до захищеної частини прямокутників. Це можна зробити, скажімо, за допомогою такого простого класу:

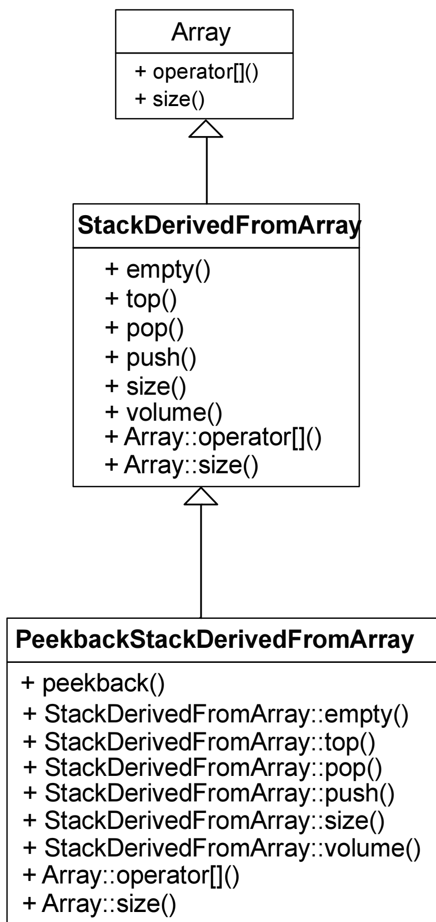


Рис. 7.13. Діаграма успадкувань стеку з підгляданням (відкритий варіант)

Лістинг 7.47

```
// Похідний клас
// другого рівня від прямокутника
class ProtectionChecker:
      public DerivedSquare {
// Захищена частина класу
// прямокутників залишається
// доступною
   length_t getA()
      {return _width;}
};
```

Але насправді захищена частина класу – серйозна загроза інкапсуляції, оскільки дозволяє неконтрольоване розширення кола класів із винятковими правами доступу до неї: адже кожен похідний клас, у принципі, має право відкрити захищену частину свого базового класу.

Тепер проаналізуємо, як відкрите успадкування можна застосувати до практичної розробки ієрархії класів, знову розглянувши, зокрема, задачу підглядання до стека з підрозділу 7.1, розв'язання якої засобами композиції виявилося недосконалим. Замінимо ієрархію об'єктів стека та масиву ієрархією відповідних класів, як це зображено на рис. 7.13.

Далі наведено визначення стека, похідного від масиву (порівняйте його з текстом лістингу 7.1). Зверніть увагу на особливості звертань до масиву.

Лістинг 7.48

```
// Стек як спеціалізація масиву або гібрид масиву зі стеком
template <class Elem>
class StackDerivedFromArray: public Array<Elem> {
private:
```

```
   static const size_t _bos; // Дно стека
   size_t _top; // Верхівка стека
// Більше немає масиву як атрибута
public:
// Конструктор, закритий від конвертування типів
   explicit StackDerivedFromArray(const size_t size): _top(_bos),
      Array<Elem>(size) {}   // Масив, ініціалізований як базовий клас
   ~StackDerivedFromArray() {}    // Деструктор
// Чи не порожній стек?
   bool empty() const {return _top==_bos;}
// Чи не переповнений контейнер?
   bool full() const {
// Тепер звертаємося до функції базового класу,
// використовуючи оператор визначення області дії
      return _top==Array<Elem>::size()-1;
   }
// Видати поточний розмір стека
   size_t size() const {return _top+1;}
// Видати ємність контейнера
   size_t volume() const {
// Звертаємося до функції базового класу, адаптуючи її ім'я
      return Array<Elem>::size();
   }
// Керування стеком
// Видати верхівку стека
   const Elem& top() const {
// До стека як представника класу, похідного від
// масиву, застосовуємо його операцію індексування
      return (*this)[_top];
   }
// Виштовхнути верхівку стека
   void pop() {_top--;}
// Помістити до стека
   void push(const Elem& value) {
      (*this)[++_top]=value; // Знову застосовуємо операцію індексування
   }
};
template <class Elem>
const size_t StackDerivedFromArray<Elem>::_bos=-1;
```

Проаналізуємо деякі важливі особливості нової організації ієрархії.

У разі застосування ієрархії об’єктів масив належав до закритої частини стека. Коли стек надавав своїм клієнтам доступ до масиву, було за-

стосовано делегування. Так, у лістингу 7.1 функції методу `volume()` було делеговано масиву, де їх забезпечував метод `size()`.

Ієрархія класів, реалізована у вигляді відкритого успадкування, надає клієнтам стека повний доступ до відкритої частини масиву, зокрема до функції `size()`. Тому делегування стає непотрібним. Утім стек уже містить іншу функцію з цим самим іменем. У такому разі говорять, що функція похідного класу *затіняє* своїм клієнтам однойменну функцію базового класу. Звісно, до неї можна звернутися, використовуючи відповідний кваліфікатор, як це зроблено в наступному прикладі:

Лістинг 7.49

```
// Визначення стека
StackDerivedFromArray<char> s(100);
cout<<s.size()<<endl; // Виведення розміру стека: буде показано 0
// Виведення розміру масиву: буде показано 100
cout<<s.Array<char>::size()<<endl;
```

Інша можливість полягає в адаптації наявного базового класу під установлений інтерфейс похідного. Тому метод `volume()` похідного класу стеків став адаптером для методу `size()` класу масивів:

```
// Виведення розміру масиву за допомогою адаптера: буде показано 100
cout<<s.volume()<<endl;
```

Крім делегування й адаптації існує ще кілька інших важливих прийомів, відомих під загальною назвою *проектних взірців*, або *патернів* (design pattern). Рекомендуємо ознайомитися з ними самостійно [14].

Завдяки відкритому успадкуванню операція індексування стала тепер успадкованою від масиву, а тому її можна застосовувати до самого стека. У тілі стека це потребує явного звертання до указника поточного об'єкта, наприклад `(*this)[_top]`.

Просте на перший погляд розв'язання проблеми організації взаємодії класу стеків із класом масивів за допомогою ієрархії класів має серйозні наслідки. Так, операція індексування стала тепер доступною також і клієнтам стека, що повністю порушує дисципліну зберігання й обробки даних стеком.

Лістинг 7.50

```
// Некоректна поведінка стека, успадкованого від масиву
StackContainingArray<char> s(100);
// Беремо з порожнього стека
cout<<s[0]<<endl;
// Кладемо безсистемно
s[10]='a'; s[20]='b';
```

```
// Беремо не звідти, куди клали
cout<<s[15]<<endl;
```

Поміркуймо, як зарадити цьому, використовуючи відомі нам засоби. Можна зробити так, щоб стек закривав доступ до операції індексування, але, як побачимо, це лише половинчасте розв’язання проблеми.

Лістинг 7.51

```
// Стек з умовно закритим доступом до операції індексування
template <class Elem>
class StackDerivedFromArray: public Array<Elem> {
private:
   static const size_t _bos; // Дно стека
   size_t _top; // Верхівка стека
   Array<Elem>::operator[]; // Закриття доступу до операції індексування
public:
// Конструктор, закритий від конвертування типів
   explicit StackDerivedFromArray(const size_t size): _top(_bos),
// Масив, ініціалізований як базовий клас
      Array<Elem>(size) {}
// Деструктор
   ~StackDerivedFromArray() {}
// Чи не порожній стек?
   bool empty() const {return _top==_bos;}
// Чи не переповнений контейнер?
   bool full() const {
// Тепер звертаємося до функції базового класу,
// використовуючи оператор визначення області дії
      return _top==Array<Elem>::size()-1;
   }
// Видати поточний розмір стека
   size_t size() const {return _top+1;}
// Видати ємність контейнера
   size_t volume() const {
// Звертаємося до функції базового класу, адаптуючи її ім'я
      return Array<Elem>::size();
   }
// Керування стеком
// Видати верхівку стека
   const Elem& top() const {
// До стека як представника класу, похідного від
// масиву, застосовуємо його операцію індексування
      return (*this)[_top];
   }
// Виштовхнути верхівку стека
```

```
   void pop() {_top--;}
// Помістити до стека
   void push(const Elem& value) {
      (*this)[++_top]=value; // Знову застосовуємо операцію індексування
   }
};
```

Утім цей розв'язок, як і раніше, неповний, бо клієнтові стека відмовлено лише в безпосередньому використанні оператора індексування, але можливий кваліфікований доступ до цього оператора за допомогою оператора визначення області дії. Некоректне застосування стека, приклад якого було наведено в лістингу 7.50, усе ще можливе, якщо його переписати, наприклад, так:

Лістинг 7.52

```
// Некоректна поведінка стека, успадкованого від масиву,
// у разі закритого доступу до індексування
StackContainingArray<char> s(100);
// Беремо з порожнього стека
cout<<s.Array<char>::operator[](0)<<endl;
// Кладемо безсистемно
s.Array<char>::operator[](10)='a'; s.Array<char>::operator[](20)='b';
// Беремо не звідти, куди клали
cout<<s.Array<char>::operator[](15)<<endl;
```

Зверніть увагу на те, як записано оператор індексування. Варіант запису `s[i]` стає тепер недозволеним. Потрібно застосувати кваліфікатор `Array<char>::`, для чого в розгляданому контексті доведеться вжити повну, а не скорочену форму оператора індексування `operator[]`.

Успадкування стека від масиву невдале рішення. Залишається лише сподіватися, що клієнт дотримуватиме дисципліни коректного використання стека.

Вправа 7.16

Складіть тестову програму та протестуйте клас `StackDerivedFromArray`.

Подивимося, які можливості надає стек, похідний від масиву, з погляду визначення на його основі інших класів, наприклад уже відомого нам стека з підгляданням. Тепер можна за допомогою успадкування просто додати підглядання до наявної функціональності стека. Справді, кваліфікований доступ до індексування доступний будь-якому клієнтові стека, зокрема похідному від нього класу.

```
// Стек із підгляданням, похідний від стека, який      Лістинг 7.53
// успадковано від масиву: доступ до індексування опосередковано
// через операцію визначення області дії
template <class Elem>
class PeekbackStackDerivedFromArray: public StackDerivedFromArray<Elem> {
public:
// Конструктор, закритий від конвертування типів
   explicit PeekbackStackDerivedFromArray (const size_t size):
          StackDerivedFromArray(size) {}
// Деструктор
   ~PeekbackStackDerivedFromArray() {}
// Функція підглядання
   bool peekback(const size_t i, Elem& elem) const {
      if (i>=size()) {elem=top();return false;}
      elem=this->Array<Elem>::operator[](i);
      return true;
   }
};
```

Варто звернути увагу на особливу форму виклику операції індексування у функції `peekback()`:

```
elem=this->Array<Elem>::operator[](i);
```

Вона зумовлена тим, що клас стеків закрив доступ до операції індексування всім, зокрема своїм похідним класам. Тому варіант запису `(*this)[i]`, як у реалізації операцій над стеком із лістингу 7.48, знову став недозволеним, і потрібно застосувати кваліфікатор `Array<Elem>::` до повної форми оператора індексування `operator[]`. Як бачимо, можливість кваліфікованого доступу залишилась, і її застосування виправдане.

Вправа 7.17

Складіть тестову програму та протестуйте клас `PeekbackStackDerivedFromArray`.

Вправа 7.18

Визначте й реалізуйте операцію виведення вмісту стека з підгляданням.

Спробуємо точніше призначити права доступу до операції індексування. Закриємо її від клієнтів стека, але зробимо доступною для похідних класів, адже саме для цього й призначено захищену область класу.

```
// Стек з операцією індексування, умовно закритою      Лістинг 7.54
// для клієнтів, але доступною для похідних класів
```

```
template <class Elem>
class StackDerivedFromArray: public Array<Elem> {
private:
   static const size_t _bos; // Дно стека
   size_t _top; // Верхівка стека
protected:
   Array<Elem>::operator[]; // Обмеження доступу до операції індексування
public:
// Конструктор, закритий від конвертування типів
   explicit StackDerivedFromArray(const size_t size): _top(_bos),
      Array<Elem>(size) {}   // Масив, ініціалізований як базовий клас
   ~StackDerivedFromArray() {}  // Деструктор
// Чи не порожній стек?
   bool empty() const {return _top==_bos;}
// Чи не переповнений контейнер?
   bool full() const {
// Тепер звертаємося до функції базового класу,
// використовуючи оператор визначення області дії
      return _top==Array<Elem>::size()-1;
   }
// Видати поточний розмір стека
   size_t size() const {return _top+1;}
// Видати ємність контейнера
   size_t volume() const {
// Звертаємося до функції базового класу, адаптуючи її ім'я
      return Array<Elem>::size();
   }
// Керування стеком
// Видати верхівку стека
   const Elem& top() const {
// До стека як представника класу, похідного від
// масиву, застосовуємо його операцію індексування
      return (*this)[_top];
   }
// Виштовхнути верхівку стека
   void pop() {_top--;}
// Помістити до стека
   void push(const Elem& value) {
      (*this)[++_top]=value; // Знову застосовуємо операцію індексування
   }
};
```

Тепер у стеку з підгляданням не має бути кваліфікатора, а індексування має виконуватися більш звичним способом `(*this)[i]`:

```
// Стек із підгляданням, похідний від стека, який        Лістинг 7.55
// успадковано від масиву: безпосередній доступ до індексування
template <class Elem>
class PeekbackStackDerivedFromArray: public StackDerivedFromArray<Elem> {
public:
// Конструктор, закритий від конвертування типів
   explicit PeekbackStackDerivedFromArray
        (const size_t size): StackDerivedFromArraysize) {}
// Деструктор
   ~PeekbackStackDerivedFromArray() {}
// Функція підглядання
   bool peekback(const size_t i, Elem& elem) const {
      if (i>=size()) {elem=top(); return false;}
   elem=(*this)[i];
   return true;
   }
};
```

Вправа 7.19

Означте й реалізуйте операції порівняння стеків із підгляданням на рівність і нерівність їхнього вмісту.

Знайдений розв'язок кращий за попередній, але все одно незадовільний. Перевагою слід вважати те, що клієнти стека втратили можливість доступу (навіть кваліфікованого) до операції індексування, невластивої для стека. Але «підводні камені» все ще залишилися. Проаналізуємо їх.

Передусім, помилковою була сама концепція відкритого успадкування стека від масиву. У відкритого успадкування має бути лише одне призначення, яке залежно від напрямку руху слід розуміти як спеціалізацію базового класу в похідному чи як узагальнення похідного класу до базового. У такому розумінні ієрархія «стек – стек із підгляданням» цілком коректна. Справді, стекові з підгляданням властива вся поведінка стека, до того ж він має ще один власний метод, у чому й полягає його спеціалізація.

Якщо ж порівняти стек із масивом, то нічого подібного ми не виявимо. Стек не тільки не схожий на масив, але й може існувати цілком незалежно від нього (як, наприклад, стек на базі списку), залишаючись при цьому повноцінним стеком. Оскільки стек у жодному розумінні не слугує прикладом масиву, він не має успадковувати його поведінки. Тому застосування відкритого успадкування стека від масиву слід вважати невиправданим, навіть коли можна закрити доступ до небажаної поведінки.

Окрім того, усе ще залишається можливість відкрити приховане. Виявивши певну кмітливість, можна примусити стек поводитися так, ніби він є масивом, що, узагалі кажучи, неприпустимо.

Лістинг 7.56

```
// «Несерйозний» стек
// Відкритий доступ до індексування
template <class Elem>
class NaughtyStackDerivedFromArray:
   public StackDerivedFromArray<Elem> {
public:
// Конструктор, закритий від конвертування типів
   explicit NaughtyStackDerivedFromArray
       (const size_t size): StackDerivedFromArray<Elem> (size) {}
// Деструктор
   ~NaughtyStackDerivedFromArray () {}
// Відкривання доступу до операції індексування
   Array<Elem>::operator[];
};
```

Тепер над «несерйозним» стеком можна повністю виконати експеримент із несанкціонованого доступу з лістингу 7.52 за таким зразком:

Лістинг 7.57

```
// Несподівана поведінка «несерйозного» стека
NaughtyStackDerivedFromArray<char> s(100);
s.push('A'); s.push('B'); s.push('C');
cout<<s[0]<<endl; cout<<s[1]<<endl; cout<<s[2]<<endl;
```

Зроблений аналіз застосування відкритого успадкування приведе до пошуків інших способів організації успадкувань, які ми почнемо із захищеного успадкування.

7.5. Захищене успадкування

Маніпуляції з правами доступу для відкритого успадкування, власне, свідчать про невдалі спроби застосувати його в неадекватній ситуації. Ми виходили з того, що відкрите успадкування дає змогу віднести об'єкти похідного класу до типу, заданого базовим класом. Так, квадрат і прямокутник мають спільний тип прямокутника, а стек і стек із підгляданням обидва належать до стеків. Інша справа – масив і стек. Стек не має й не може мати поведінки масиву. Його відношення з масивом принципово інші: стек використовує масив для власної реалізації, хоча для цього придатна й інша структура даних, наприклад список. Від цього тип стека не змінився б: він би й надалі залишався стеком у разі дотримання принципу доступу «останнім прийшов – першим обслужили».

Для взаємодії стека з масивом ми вже використовували композицію й агрегацію. Кожна з них була б цілком прийнятним типом ієрархії, якби не потреба передати класам, похідним від стека, права доступу до методів масиву, чого нам не вдалося забезпечити.

Розглянемо розв'язання цієї проблеми згідно з концепцією *захищеного успадкування* (protected inheritance). Таке успадкування перетворює статус функцій, відкритих у базовому класі, на захищений у похідному. У разі застосовування цього виду успадкування похідним класам буде передано права доступу як до відкритої, так і до захищеної частин базового класу. Проте для клієнтів похідних класів доступ до базового класу закрито. Порівнявши наведене далі визначення стека з текстом лістингу 7.51, побачимо, що змінився тип успадкування та зникла потреба закривати доступ до операції індексування.

Лістинг 7.58

```
// Стек закриває доступ до масиву своїм клієнтам,
// але відкриває доступ до нього своїм похідним класам
template <class Elem>
class StackDerivedFromArray: protected Array<Elem> {
private:
   static const size_t _bos; // Дно стека
   size_t _top; // Верхівка стека
public:
// Конструктор, закритий від конвертування типів
   explicit StackDerivedFromArray(const size_t size): _top(_bos),
      Array<Elem>(size) {}  // Масив, ініціалізований як базовий клас
   ~StackDerivedFromArray() {}  // Деструктор
// Чи не порожній стек?
   bool empty() const {return _top==_bos;}
// Чи не переповнений контейнер?
   bool full() const {
// Тепер звертаємося до функції базового класу,
// використовуючи оператор визначення області дії
      return _top==Array<Elem>::size()-1;
   }
// Видати поточний розмір стека
   size_t size() const {return _top+1;}
// Видати ємність контейнера
   size_t volume() const {
// Звертаємося до методу базового класу size(),
// делегуючи йому функції методу volume()
      return Array<Elem>::size();
   }
```

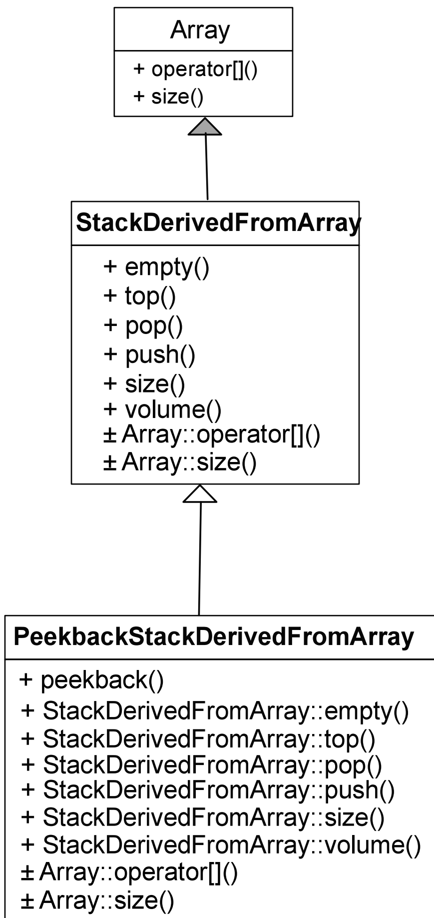


Рис. 7.14. Діаграма успадкувань стеку з підгляданням (захищений варіант)

```
// Керування стеком
// Видати верхівку стека
   const Elem& top() const {
// До стека як представника
//класу, похідного від масиву,
// застосовуємо його операцію
//індексування
      return (*this)[_top];
   }
// Виштовхнути верхівку стека
   void pop() {_top--;}
// Помістити до стека
   void push(const Elem& value) {
      (*this)[++_top]=value;
// Знову застосовуємо операцію
// індексування
   }
};
```

Залишивши ієрархію стека та стека з підгляданням без змін у вигляді відкритого успадкування, що, як бачимо, адекватно відтворює суть зв'язку між ними, одержимо трирівневу ієрархію, зображену на рис. 7.14. Захищене успадкування тут позначено сірою стрілкою. Видно, що відкриті методи класу масивів `Array` перетворюються на захищені методи класу стеків `StackDerivedFromArray`, тому стають доступними (знову ж як захищені методи) похідному класу стеків із підгляданням `PeekbackStackDerivedFromArray`.

Останнє визначення стека з підгляданням із попереднього підрозділу залишається без змін. Як і раніше, це повноцінний стек із додатковою функціональністю – методом підглядання.

Вправа 7.20

Складіть тестову програму та виконайте тестування класів `StackDerivedFromArray` і `PeekbackStackDerivedFromArray`.

Тепер стає неможливим некоректне використання клієнтами похідних класів операції індексування. Спроби добратися до елемента масиву як зі звичайного стека, так і зі стека з підгляданням призводять до синтаксичних помилок:

Лістинг 7.59

```
StackDerivedFromArray<char> stachar(100);
stachar.Array<char>::operator[](30)='A';   // ERROR!
PeekbackStackDerivedFromArray<char> pbstachar(100);
pbstachar.Array<char>::operator[](30)='A'; // ERROR!
```

Наведене розв'язання цієї проблеми у формі захищеного успадкування має суттєву перевагу порівняно зі спробами закрити доступ у разі відкритого успадкування. Вона полягає в послідовному дотриманні відповідних прав доступу до об'єктів: навіть застосування операції визначення області дії не надає клієнтам доступу до операції індексування.

Вправа 7.21

Задайте ієрархію з трьох класів `Father`, `Son` і `Grandson`, пов'язавши `Father` і `Son` захищеним, а `Son` і `Grandson` – відкритим успадкуванням. Означте один відкритий

```
void Father::open(void);
```

й один захищений метод

```
void Father::protect(void);
```

у базовому класі. Проаналізуйте можливість доступу до них як у кожному з похідних класів, так і з боку клієнтів цих класів. Відкрийте доступ до захищеного методу почергово в усіх похідних класах і простежте за доступом клієнтів до методу, що став відкритим.

7.6. Закрите успадкування

Перш ніж розглянути ще один тип успадкувань, порівняємо два попередні. Відкрите успадкування наділяє всі похідні класи типом базового класу, а тому відкриває доступ до відкритої частини базового класу клієнтам усіх його похідних класів. Захищене успадкування не поширює тип базового класу на похідні, закриваючи від їхніх клієнтів відкриту частину базового класу, але похідні класи далі передають доступ до відкритої частини базового класу всім класам, одержаним із них за допомогою відкритого успадкування.

Тепер похідному класові залишилося закрити базовий клас не лише від клієнтів, але й від похідних від нього самого класів. Цю функцію виконує *закрите успадкування* (private inheritance), яке перетворює відкриту частину базового класу на закриту частину похідного.

Задамо стек як результат закритого успадкування від масиву:

Лістинг 7.60

```
// Стек закриває доступ до масиву як своїм клієнтам,
// так і похідним класам
template <class Elem>
class StackDerivedFromArray: private Array<Elem> {
private:
   static const size_t _bos; // Дно стека
   size_t _top; // Верхівка стека
public:
// Конструктор, закритий від конвертування типів
   explicit StackDerivedFromArray(const size_t size): _top(_bos),
      Array<Elem>(size) {} // Масив, ініціалізований як базовий клас
   ~StackDerivedFromArray() {} // Деструктор
// Чи не порожній стек?
   bool empty() const {return _top==_bos;}
// Чи не переповнений контейнер?
   bool full() const {
      return _top==Array<Elem>::size()-1; // Тепер звертаємося до функції
         // базового класу, використовуючи оператор визначення області дії
   }
// Видати поточний розмір стека
   size_t size() const {
      return _top+1;
   }
// Видати ємність контейнера
   size_t volume() const {
      return Array<Elem>::size(); // Звертаємося до методу базового
               // класу size(), делегуючи йому функції методу volume()
   }
// Керування стеком
// Видати верхівку стека
   const Elem& top() const {
      return (*this)[_top]; // До стека як представника класу,
       // похідного від масиву, застосовуємо його операцію індексування
   }
// Виштовхнути верхівку стека
   void pop() {_top--;}
// Помістити до стека
```

```
   void push(const Elem& value) {
      (*this)[++_top]=value; // Знову застосовуємо операцію індексування
   }
};
```

Якщо тепер знову спробувати задати стек із підгляданням як клас, похідний від класу стеків, то не буде доступу до масиву, а отже, і до його операції індексування. Щоб уникнути цього, відкриємо похідному класу доступ до операції індексування вже відомим способом – уточненням прав доступу через захищену частину стека:

Лістинг 7.61

```
// Стек закриває доступ до масиву як своїм клієнтам,
// так і похідним класам, але похідним класам відкриваємо доступ
// до операції індексування, помістивши її в захищеній частині
template <class Elem>
class StackDerivedFromArray: private Array<Elem> {
private:
   static const size_t _bos; // Дно стека
   size_t _top; // Верхівка стека
protected:
// Відкриваємо похідним класам доступ до операції індексування
   Array<Elem>::operator[];
public:
// Конструктор, закритий від конвертування типів
   explicit StackDerivedFromArray(const size_t size): _top(_bos),
      Array<Elem>(size) {}  // Масив, ініціалізований як базовий клас
   ~StackDerivedFromArray() {}  // Деструктор
// Чи не порожній стек?
   bool empty() const {return _top==_bos;}
// Чи не переповнений контейнер?
   bool full() const {
      return _top==Array<Elem>::size()-1; // Тепер звертаємося до
// функції базового класу, використовуючи оператор визначення області дії
   }
// Видати поточний розмір стека
   size_t size() const {return _top+1;}
// Видати ємність контейнера
   size_t volume() const {
      return Array<Elem>::size(); // Звертаємося до методу базового
                  // класу size(), делегуючи йому функції методу volume()
   }
// Керування стеком
// Видати верхівку стека
   const Elem& top() const {
```

```
        return (*this)[_top];
// До стека як представника
// класу, похідного від масиву,
// застосовуємо його операцію
// індексування
    }
// Виштовхнути верхівку стека
    void pop() {_top--;}
// Помістити до стека
    void push(const Elem& value) {
        (*this)[++_top]=value;
// Знову застосовуємо операцію
// індексування
    }
};
```

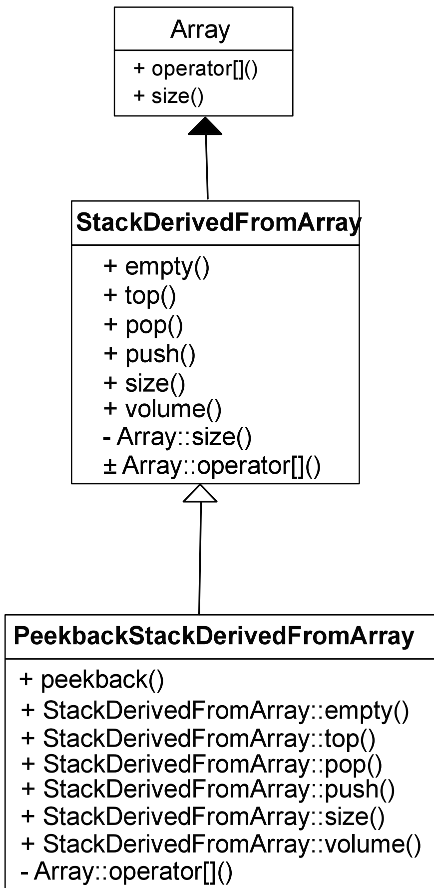


Рис. 7.15. Діаграма успадкувань стеку з підгляданням (закритий варіант)

Визначення стека з підгляданням знову ж залишається без змін. Одержану ієрархію показано на рис. 7.15. Захищене успадкування позначено чорною стрілкою. Функція `size()` класу масивів стає закритою функцією класу стеків, тоді як оператору індексування в ньому надано статус захищеного.

Якщо знову утворити стек із підгляданням за допомогою відкритого успадкування, то операція індексування стане в ньому закритою. Це означає, що методи стека з підгляданням можуть використовувати її для власної реалізації, але ця операція закрита від клієнтів стека з підгляданням.

Підіб'ємо підсумки та спробуємо охарактеризувати виявлені властивості різних видів успадкувань. Для організації ієрархії стека відносно масиву ми почергово спробували застосувати кожен із трьох видів успадкування: відкрите, захищене та закрите. Розглянемо їхні переваги та вади.

Відкрите успадкування не відповідає характеру залежності стека від масиву, оскільки в разі його застосування стек перетворюється на різновид масиву. Доводиться штучно забороняти клієнтам стека доступ до ма-

сиву, що, власне, зводить нанівець принципи відкритого успадкування. До того ж класи, похідні від стека з підгляданням, усе одно можуть відкрити своїм клієнтам доступ до функцій масиву.

Захищене успадкування закриває масив від клієнтів стека, що усуває проблему неадекватної поведінки стека, властиву відкритому успадкуванню. Тепер стек сам використовує масив, проте не надає цієї послуги своїм клієнтам. Окрім того, захищене успадкування відкриває похідним класам доступ до масиву, що дає змогу легко реалізувати стек із підгляданням як похідний від масиву клас. Щоправда, класи, похідні від стека з підгляданням, можуть знову ж таки відкрити доступ до масиву.

Закрите успадкування, можливо, найточніше відповідає ідеї залежності стека від масиву. Стек використовує масив для власної реалізації та закриває його від своїх клієнтів і похідних класів. Якщо потрібно надати власному похідному класу доступ до масиву, стек може уточнити права доступу до нього, замінивши його статус із закритого на захищений. Щоправда, тоді знову будь-який похідний клас зможе відкрити своїм клієнтам доступ до операції індексування, і це порушить принципи використання стека.

Повністю адекватним розв'язанням задачі організації ієрархії стека виявилося відкрите успадкування стека з підгляданням від стека. Справді, незалежно від способу організації самого стека відкрите успадкування дає змогу кожному клієнтові стека з підгляданням виконувати всі дії, задані над стеками, а також функцію підглядання. У такому разі очевидно, що стек із підгляданням являє собою спеціалізацію стека, а тому успадковує його тип, тобто сигнатуру його методів, і поведінку – їх реалізацію.

Успадкування типу зазвичай називають *успадкуванням інтерфейсу*, а успадкування поведінки – *успадкуванням реалізації*. Об'єднавши їх, можна говорити про *повне успадкування*. Зазначимо, що для похідних класів можна як скасувати успадкування інтерфейсу, закривши доступ до будь-якого з методів, так і відмовитися від успадкованої реалізації, замінивши її власною. Усе це потребує дотримання чіткої дисципліни в організації ієрархій.

Вправа 7.22

Задайте ієрархію з трьох класів `Father`, `Son` і `Grandson`, пов'язавши `Father` та `Son` закритим успадкуванням, а `Son` і `Grandson` –відкритим. Означте в базовому класі відкритий метод

```
void Father::open(void);
```

Проаналізуйте можливість доступу до нього як у кожному з похідних класів, так і з боку клієнтів усіх похідних класів. Відкрийте доступ до закритого методу почергово в кожному похідному класі та простежте за доступом клієнтів похідних класів до щойно відкритого методу.

7.7. Комбінована ієрархія

Розглянувши два фундаментальні типи ієрархії, варто порівняти результати їх застосування. Спільною рисою успадкування та композиції залишається вкладення базового об'єкта до похідного (у разі успадкування) та компонента до композита (у разі композиції). Принципова відмінність композиції від застосування відкритого успадкування полягає в тому, що клієнти компонента не мають жодного доступу до композита (у разі композиції), тоді як клієнти об'єктів похідного класу мають доступ до відкритої частини базового класу.

Якщо ж порівняти закрите успадкування з композицією, то спільна риса залишається тією самою, а відмінність повністю зникає, оскільки клієнти об'єктів похідного класу тепер утрачають доступ до базового.

Розглянемо приклад. Доповнимо визначення стека з лістингу 7.1 делегуванням операції індексування.

Лістинг 7.62

```
// Стек із вкладеним до нього масивом
template <class Elem>
class StackContainingArray {
private:
   static const size_t _bos; // Дно стека
   size_t _top; // Верхівка стека
   Array<Elem> _stackArray; // Масив для розміщення стека
protected:
// Делегування масиву операції індексування: забезпечення доступу
// до неї в похідних класах
   const Elem& operator[] (const size_t index) const {
      return _stackArray[index];
   }
public:
// Конструктор, закритий від конвертування типів
   explicit StackContainingArray(const size_t size):
      _stackArray(size), _top(_bos) {}
   ~StackContainingArray() {}  // Деструктор
// Чи не порожній стек?
   bool empty() const {return _top==_bos;}
```

```
// Чи не переповнений контейнер?
   bool full() const {return _top==_stackArray.size()-1;}
// Видати поточний розмір стека
   size_t size() const {return _top+1;}
// Видати ємність контейнера
   size_t volume() const {return _stackArray.size();}
// Керування стеком
// Видати верхівку стека
   const Elem& top() const {return _stackArray[_top];}
// Виштовхнути верхівку стека
   void pop() {_top--;}
// Помістити до стека
   void push(const Elem& value) {_stackArray[++_top]=value;}
};
template <class Elem>
const size_t StackContainingArray<Elem>::_bos=-1;
```

Далі утворимо стек із підгляданням звичним способом – за допомогою відкритого успадкування.

Лістинг 7.63

```
// Стек із підгляданням, похідний від стека
// з вбудованим масивом. Застосування делегованого індексування
template <class Elem>
class PeekbackStackDerivedFromArray:
      public StackContainingArray <Elem> {
public:
// Конструктор, закритий від конвертування типів
   explicit PeekbackStackDerivedFromArray
       (const size_t size): StackDerivedFromArraysize) {}
   ~PeekbackStackDerivedFromArray() {}  // Деструктор
// Функція підглядання
   bool peekback(const size_t i, Elem& elem) const {
      if (i>=size()) {elem=top(); return false;}
   elem=(*this)[i];
   return true;
   }
};
```

Відношення між класами показано на рис. 7.16.

Вправа 7.23

Складіть тестову програму та протестуйте щойно заданий стек із підгляданням.

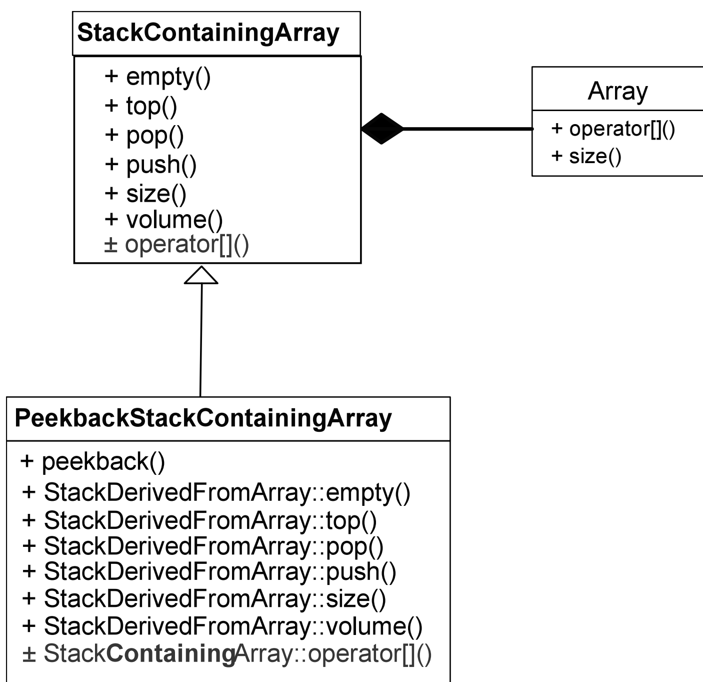


Рис. 7.16. Діаграма успадкувань стеку з підгляданням (вкладений масив)

Побудована тут ієрархія має принципову перевагу над попередніми. Вона полягає в точнішому виборі операції, доступ до якої відкрито похідному класу. Якщо у варіантах із попередніх підрозділів був відкритим доступ до всіх операцій індексування, то застосований тут метод делегування надає доступ тільки до сталої операції індексування, здатної функціонувати лише як селектор, але не як модифікатор. Тому немає небезпеки несанкціонованого втручання у внутрішність стека.

Вправа 7.24

У стеку, утвореному за допомогою закритого успадкування з масиву (лістинг 7.61), реалізуйте делегування до масиву захищеної сталої операції індексування для її використання в стеку з підгляданням. Виконайте тестування побудованих класів.

Вправа 7.24 демонструє, що вирішення проблеми організації доступу до закритої частини базового класу (зокрема, до сталої операції індексу-

вання з боку стека з підгляданням) за допомогою закритого успадкування та композиції рівноцінні. Це свідчить про те, що ієрархія класів та ієрархія об'єктів можуть певною мірою заміняти одна одну. Зауважимо, що замість композиції для вирішення зазначеної проблеми може бути успішно використана і агрегація масиву за допомогою указника.

Вправа 7.25

У стеку, до якого агреговано масив (вправа 7.3), реалізуйте делегування до захищеної сталої операції індексування для її використання в стеку з підгляданням. Протестуйте побудовані класи.

Порівнюючи вкладення (тобто композицію чи агрегацію) як ієрархію об'єктів із закритим успадкуванням як ієрархією класів, зазвичай перевагу надають вкладенню, передусім агрегації. У розглянутому прикладі застосування агрегації послабить залежність одного класу від іншого: реалізація масиву стане прихованою, адже масив із частини класу перетворюється на приєднаний до стека об'єкт. Це зумовлює більшу незалежність стека від масиву: наприклад, зміна структури масиву не спричинятиме повторної компіляції стека та похідних від нього класів. Наголосимо, що в разі застосування успадкувань залежність між класами залишається найсильнішою.

7.8. Кратне успадкування

Особливо обережно слід застосовувати *кратне успадкування* (multiple inheritance). Знову розглянемо приклад. Візьмемо класи кішок, хижаків і кімнатних тварин. На перший погляд, походження класу кішок `Cat` як від класу хижаків `Predator`, так і від класу кімнатних тварин `Pet` цілком коректне, однак тут криється певна суперечливість.

Почнемо з класу хижаків, представник якого має кличку та характеризується улюбленою здобиччю:

Лістинг 7.64

```
// Клас хижаків
class Predator {
private:
   string _name; // Прізвисько
   string _favoritePrey; // Переважна здобич
public:
   Predator(const string& prey, const string& name); // Конструктор
   ~Predator(); // Деструктор
   void setFavotitePray (const string&); // Модифікатор здобичі
```

```
   const string& getFavotitePray() const; // Селектор здобичі
   void setName (const string&); // Модифікатор прізвиська
   const string& getName() const; // Селектор прізвиська
};
```

А це клас кімнатних тварин, які також мають прізвиська, але характеризуються улюбленою іграшкою.

Лістинг 7.65

```
// Клас кімнатних тварин
class Pet {
private:
   string _name; // Прізвисько
   string _favoriteToy; // Улюблена іграшка
public:
   Pet (const string& prey, const string& name); // Конструктор
   ~Pet (); // Деструктор
   void setFavotiteToy (const string&); // Модифікатор іграшки
   const string& getFavotiteToy () const; // Селектор іграшки
   void setName (const string&); // Модифікатор прізвиська
   const string& getName() const; // Селектор прізвиська
};
```

Кожен із цих класів задає сукупність його повноцінних об’єктів – хижаків і кімнатних тварин. Окрім того, можливий варіант *змішування* (mixing) цих класів у одному похідному, наприклад у класі кішок `Cat` (або песиків, пацюків тощо). Схему успадкувань у цьому разі зображено на рис. 7.17.

Клас кішок може мати такий вигляд:

Лістинг 7.66

```
// Клас кішок: хижаки та кімнатні тварини водночас
class Cat: public Predator, public Pet {
public:
// Конструктор
   Cat(const string& prey, const string& toy, const string& name);
   ~Cat();  // Деструктор
};
```

Зверніть увагу: клас кішок не має власної закритої частини. Це не можна вважати великою несподіванкою: власної закритої частини не було й у класу квадратів із підрозділу 7.2 (див., наприклад, лістинг 7.10). Похідний клас успадковує закриту частину базового класу, а доступ до неї в такому разі надають селектори та модифікатори. Інша річ, що клас кішок утворено змішуванням двох базових класів, а тому

він успадкував дві закриті частини – по одній від кожного класу. Тепер атрибут прізвиська є в об’єкті класу кішок двічі, а тому виникає запитання: як забезпечити узгодженість цих імен або уникнути надлишкового кодування?

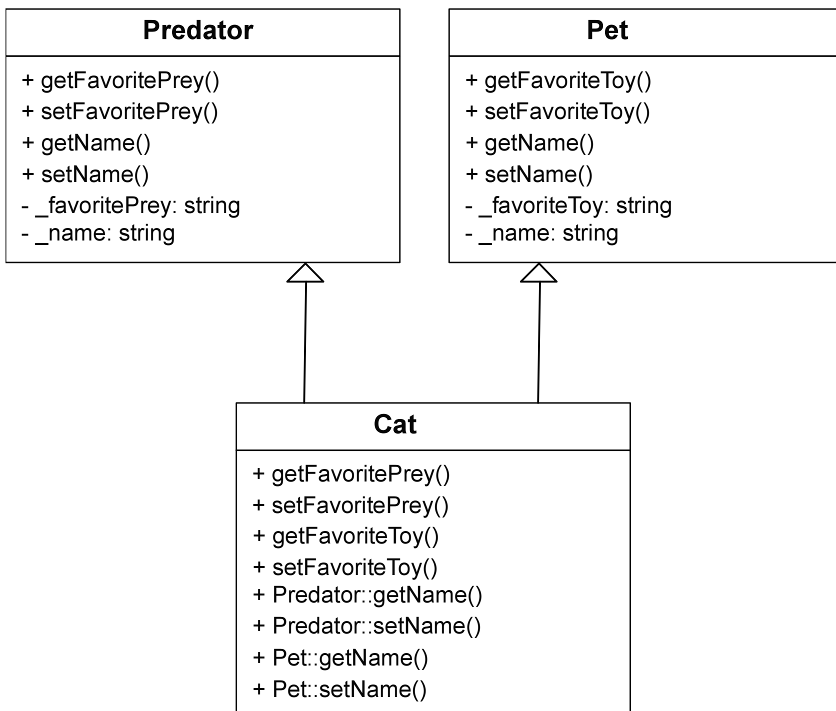


Рис. 7.17. Кратне успадкування

Розглянемо приклад, у якому використано кратне успадкування. Створимо об’єкт похідного класу, а потім змінимо атрибут одного з базових класів (лістинг 7.67).

Лістинг 7.67

```
Cat myCat("мишка", "клубок", "Мурка");
myCat.Predator::setName("Багіра");
```

Яку кличку матиме тепер кішка `myCat`? Як бачимо, застосовувати кратне успадкування слід украй обережно, тому що є суперечності між однойменними атрибутами та функціями в базових класах.

Вправа 7.26

Як коректно задати утиліти виведення в кожному з класів хижаків, кімнатних тварин і кішок? Означте й обґрунтуйте.

Проблема стає ще гострішою, якщо ланцюжок кратних успадкувань починається зі спільного базового класу. Подивимося, як відбувається успадкування в такому разі.

Розглянемо класи прямокутників і ромбів, які походять від спільного базового класу паралелограмів і, крім того, є базовими класами для квадратів.

Лістинг 7.68

```
typedef double length_t;
typedef double angle_t;
// Клас паралелограмів
class Parallelogram {
protected:
   length_t _height; length_t _width; // Сторони паралелограма
   angle_t _angle;   // Кут між сторонами
public:
   Parallelogram(const length_t h, const length_t w, const length_t a):
// Конструктор
                 _height (h), _width (w), _angle (a) {}
   ~Parallelogram() {}   // Деструктор
};
const double pi=3.141592653589793;
// Клас прямокутників
class Rectangle: public Parallelogram {
public:
   Rectangle(const length_t h, const length_t w):  // Конструктор
      Parallelogram (h, w, pi/2) {}
   ~Rectangle() {} // Деструктор
};
// Клас ромбів
class Rhombus: public Parallelogram {
public:
   Rhombus(const length_t side, angle_t a): // Конструктор
      Parallelogram (side, side, a) {}
   ~Rhombus() {} // Деструктор
};
// Квадрат походить водночас від прямокутника та ромба
class Square: public Rectangle, public Rhombus {
public:
```

```
   Square(short side): // Конструктор
      Rectangle(side, side), Rhombus (side, pi/2) {}
   ~Square() {} // Деструктор
};
```

Тепер у кожному екземплярі квадрата буде два екземпляри паралелограмів: один – від базового класу прямокутників, другий – від ромбів.

Вправа 7.27

Доповніть класи паралелограма, прямокутника, ромба та квадрата потрібними селекторами й модифікаторами, функціями обчислення площі та периметра, а також утилітою виведення. Складіть тестову програму та проаналізуйте виклики конструкторів під час створення об'єктів похідних класів.

На рис. 7.18 показано ієрархічну залежність створених класів.

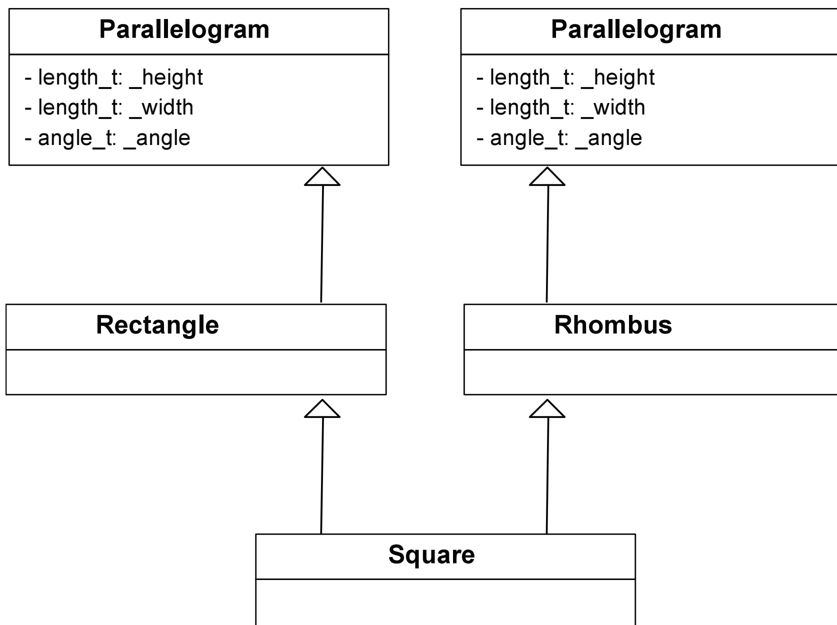


Рис. 7.18. Кратне успадкування із спільного базового класу

Однак нашому уявленню про взаємозв'язок між розглянутими видами чотирикутників більше відповідає рис. 7.19.

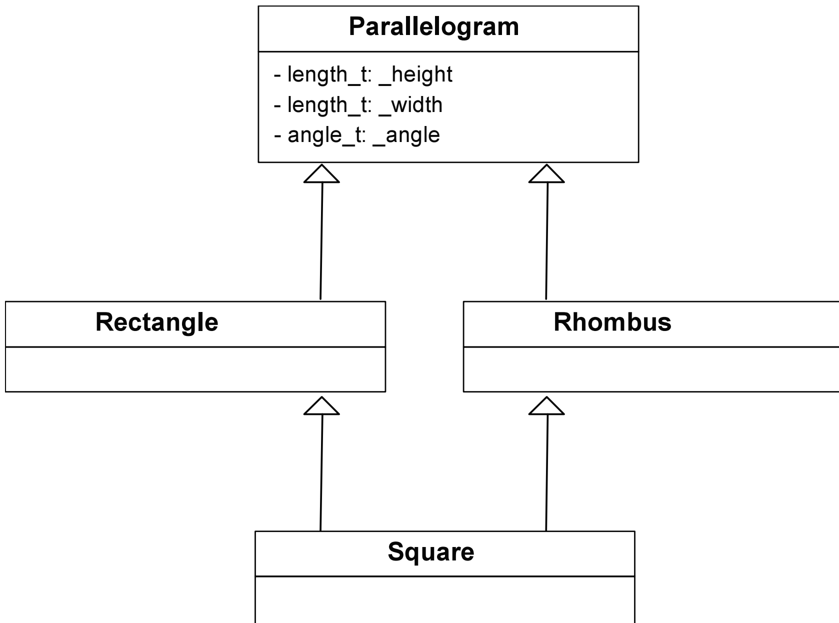


Рис. 7.19. Ієрархія успадкувань із спільного базового класу

Повністю досягти цього не вдається, однак є штучний прийом, який дає змогу скасувати стандартний порядок виклику конструкторів. Він має назву *віртуального успадкування* (virtual inheritance). Таке успадкування делегує виклик конструктора базового класу на рівень нижче.

Підправимо визначення класів так:

Лістинг 7.69

```
typedef double length_t;
typedef double angle_t;
// Клас паралелограмів (залишається без змін)
class Parallelogram {
protected:
   length_t _height; length_t _width; // Сторони паралелограма
   angle_t _angle;                    // Кут між сторонами
public:
// Конструктор
   Parallelogram(const length_t h, const length_t w, const length_t a):
               _height (h), _width (w), _angle (a) {}
   ~Parallelogram() {} // Деструктор
};
```

```
const double pi=3.141592653589793;
// Клас прямокутників: віртуальне успадкування
class Rectangle: virtual public Parallelogram {
public:
   Rectangle(const length_t h, const length_t w): // Конструктор
      Parallelogram (h, w, pi/2) {}
   ~Rectangle() {} // Деструктор
};
// Клас ромбів: віртуальне успадкування
class Rhombus: virtual public Parallelogram {
public:
   Rhombus(const length_t side, angle_t a): // Конструктор
      Parallelogram (side, side, a) {}
   ~Rhombus() {} // Деструктор
};
// Квадрат походить водночас від прямокутника та ромба;
// додано ініціалізацію паралелограма
class Square: public Rectangle, public Rhombus {
public:
   Square(short side): // Конструктор
      Parallelogram (side, side, pi/2),
      Rectangle(side, side),
      Rhombus (side, pi/2) {}
   ~Square() {} // Деструктор
};
```

Однак тут не повністю реалізовано схему з рис. 7.19. Означенню з лістингу 7.69 більше відповідає рис. 7.20.

Розглянемо створення прямокутника та ромба як самостійних об'єктів. У цьому разі віртуальність успадкування не проявляється. Створенню як прямокутника, так і ромба передує створення базових прямокутників.

Вправа 7.28

Виконайте трасування наведеного далі коду й простежте за послідовністю викликів конструкторів прямокутника та паралелограма, ромба та паралелограма.

Лістинг 7.70

```
// Створення ромба та базового паралелограма
   Rhombus(10, pi/3);
// Створення прямокутника та базового паралелограма
   Rectangle(10, 20);
```

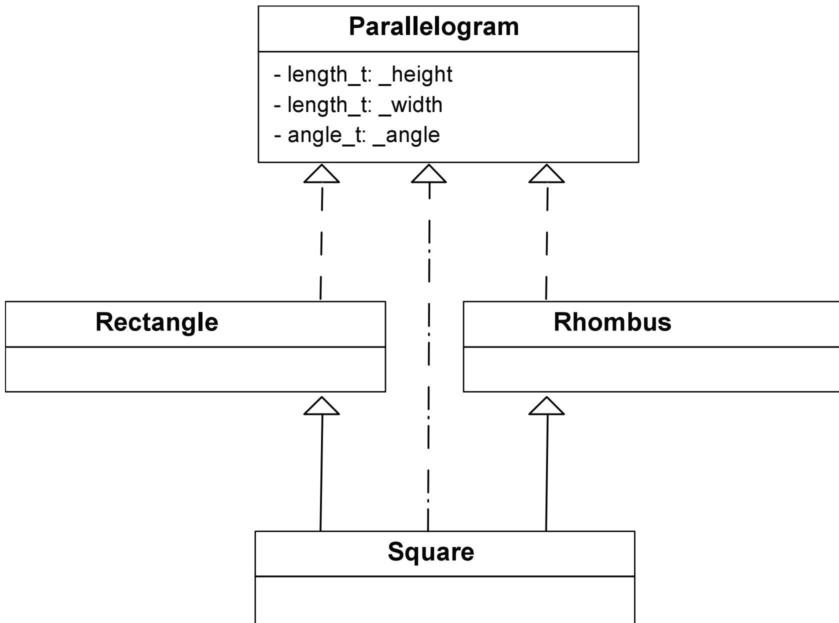


Рис. 7.20. Кратне віртуальне успадкування із спільного базового класу

Тепер розглянемо створення квадрата. Коли конструктор квадрата викликає конструктор прямокутника, спрацьовує віртуальне успадкування. Прямокутник не відповідає більше за створення паралелограма, як і ромб. За виклик конструктора паралелограма тепер відповідає квадрат, а тому в його конструкторі записано три ініціалізації.

Вправа 7.29

Виконайте трасування коду

```
// Створення квадрата, паралелограма, ромба та прямокутника
   Square(10);
```

і простежте за послідовністю викликів конструкторів прямокутника, ромба й паралелограма. Поміняйте порядок ініціалізації конструкторів базових класів у конструкторі квадрата й порівняйте послідовність виклику конструкторів із попереднім випадком. Видаліть ініціалізацію паралелограма з конструктора прямокутника та прокоментуйте результат компіляції.

На рис. 7.20 звичайні виклики конструкторів прямокутника та ромба позначено суцільною лінією, додатковий виклик конструктора паралелограма конструктором квадрата – штрих-пунктирною лінією, а скасовані виклики конструктора паралелограма з конструкторів прямокутника та ромба (віртуальне успадкування) – пунктирними.

Проблеми, виявлені під час вивчення кратного успадкування, – вагома підстава для того, щоб не використовувати його в практиці програмування. Точніше, не варто застосовувати повне кратне успадкування. Кратне успадкування ефективне в разі успадкування інтерфейсів.

7.9. Успадкування інтерфейсу й успадкування реалізації; абстрактні класи

Приклади класу стеків, розглянуті в цьому розділі, подають кожну конкретну реалізацію стека як окремий тип, не пов'язаний з іншими типами стеків. Водночас вони являють собою втілення однієї й тієї самої абстракції стека. Щоб запровадити цю абстракцію на рівні програми, використовують поняття *абстрактного інтерфейсу*, тобто інтерфейсу, реалізацію якого відкладено чи делеговано іншим класам. У мові C++ цього досягають за допомогою *абстрактних класів* (abstract class).

Абстрактний клас може містити лише сигнатури методів, які можуть не мати реалізації. Об'єкт цього класу не можна створити конструктором інакше, як через його похідні класи. Тому екземпляри абстрактних класів можуть з'явитися лише як базова частина конкретних екземплярів похідних класів. Виникає питання, чи можна вживати імена, типізовані абстрактними класами. Нехай `class Abstract` абстрактний клас, а `class Concrete: public Abstract` конкретний клас, похідний від нього. Тоді визначення `Abstract nothing;` некоректне, оскільки його виконання вимагало б осібного виклику конструктора абстрактного класу, що неможливо, інакше в програмі з'явився б об'єкт з невизначеними методами. Але цілком можливим стає визначення указника `Abstract* p = new Concrete` так само, як відсилки абстрактного класу до конкретного об'єкта `Abstract& r = p`.

Як ми побачимо далі, основне застосування указників і відсилок до абстрактних класів полягає у передачі параметрів. Якщо `Stack` – це абстрактний клас, то його можна використати у функції як тип параметра, який передано за допомогою відсилки, наприклад так:

```
template <type Elem>
void process(Stack<Elem>& stackToProcess);
```

або як тип указника, значення якого буде конкретизовано під час створення:

```
Stack<char>* myStack= new StackContainingArray<char>(100);
```

Тепер, щоправда, виникне проблема типізації. Згідно з домовленостями щодо використання об'єктів похідних типів замість об'єкта базового типу, у цьому разі над об'єктами похідних класів виконуються функції базового класу. Однак можуть виникнути проблеми використання цих функцій, якщо доступ до об'єкта забезпечує базовий абстрактний клас, бо, як щойно було сказано, його методи можуть узагалі не мати реалізації. Вихід полягає в застосуванні *динамічного зв'язування* (dynamic binding) і базованого на ньому *поліморфізму* (polymorphism), що забезпечується механізмом *віртуальних функцій* (virtual function).

Наведемо та прокоментуємо приклад визначення абстрактного класу із віртуальними функціями.

Лістинг 7.71

```
// Інтерфейс стеків
template <class T>
class Stack {
public:
   virtual ~Stack() {}
   virtual bool empty() const=0; virtual const T& top() const=0;
   virtual void pop()=0; virtual void push(const T& value)=0;
};
```

У цьому визначенні використано кілька домовленостей.

Перша домовленість стосується регулювання механізмів *заміщення* (overriding) методів. Досі вибір функції для виклику ми задавали статично типом імені (псевдоніма, указника), використаного для доступу до об'єкта. Цей механізм називають статичним зв'язуванням імен. Таке зв'язування можна скасувати й замінити його динамічним, якщо оголосити функцію віртуальною. Виклик віртуального методу відбувається не за типом імені, а за його значенням. Тому у віртуальних методах можливе динамічне зв'язування залежно від типу значення, наявного на момент виклику: функція базового класу заміщується відповідною функцією найнижчого в ієрархії похідного класу, якому належить об'єкт.

Динамічне зв'язування дає змогу застосовувати в базовому класі віртуальні методи, що не матимуть реалізації. Їх називають *чисто віртуальними* (pure virtual). Наявність таких методів свідчить про належність класу до категорії абстрактних.

Друга зі згаданих раніше домовленостей стосується самого віднесення класу до категорії абстрактних. Про абстрактність класу свідчить наявність хоча б одного чисто віртуального методу, позначеного за допомогою виразу «=0».

Наведемо кілька прикладів використання інтерфейсу стеків.

Лістинг 7.72

```
// Обмежений стек із вбудованим масивом, похідний
// від абстрактного стека
template <class Elem>
class StackContainingArray: public Stack<Elem> {
private:
   static const size_t _bos; // Дно стека
   size_t _top; // Верхівка стека
   Array<Elem> _stackArray; // Масив для розміщення стека
public:
// Конструктор, закритий від конвертування типів
   explicit StackContainingArray(const size_t size):
      _stackArray(size), _top(_bos) {}
   ~StackContainingArray() {} // Деструктор
// Чи не порожній стек?
   bool empty() const {return _top==_bos;}
// Чи не переповнений контейнер?
   bool full() const {return _top==_stackArray.size()-1;}
// Видати поточний розмір стека
   size_t size() const {return _top+1;}
// Видати ємність контейнера
   size_t volume() const
      {return _stackArray.size();}
// Керування стеком
// Видати верхівку стека
   const Elem& top() const
      {return _stackArray[_top];}
// Виштовхнути верхівку стека
   void pop() {_top--;}
// Помістити до стека
   void push(Elem value)
      {_stackArray[++_top]=value;}
};
template <class Elem>
const size_t StackContainingArray<Elem>::
   _bos=-1;
```

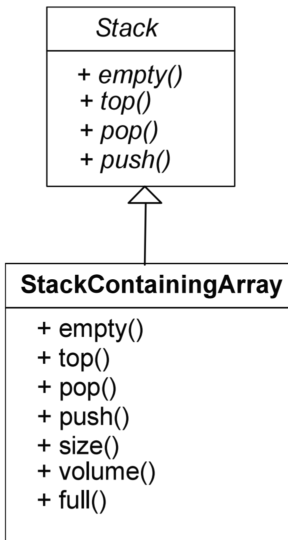


Рис. 7.21. Успадкування базового абстрактного класу

Схему класів зображено на рис. 7.21. Абстрактні класи та чисто віртуальні методи, як заведено, виділено курсивом.

Зверніть увагу на те, що власні функції стека на базі масиву, наприклад `size()` або

volume(), стануть недоступними через інтерфейс абстрактного класу. Тому, реалізуючи абстрактний клас, не варто додавати до інтерфейсу нові функції, оскільки вони виявляться недоступними через базовий інтерфейс.

Вправа 7.30

Базуючись на абстрактному класі стеків із лістингу 7.71, визначте абстрактний клас обмежених стеків так, щоб через нього був би доступний повністю весь інтерфейс як стеків із вбудованим масивом (лістинг 7.1), так і стека, до якого масив агреговано (вправа 7.3).

Подивимося тепер, на що перетвориться ієрархія класів, якщо реалізувати абстрактний клас стеків через стек, похідний від масиву.

Лістинг 7.73

```
// Обмежений стек, похідний від масиву та від
// абстрактного стека
template <class Elem>
class StackDerivedFromArray: public Stack<Elem>, protected Array<Elem> {
private:
   static const size_t _bos; // Дно стека
   size_t _top; // Верхівка стека
public:
// Конструктор, закритий від конвертування типів
   explicit StackDerivedFromArray(const size_t size): _top(_bos),
      Array<Elem>(size) {}  // Масив, ініціалізований як базовий клас
   ~StackDerivedFromArray() {}   // Деструктор
// Чи не порожній стек?
   bool empty() const {return _top==_bos;}
// Чи не переповнений контейнер?
   bool full() const {
// Виклик функції базового класу
      return _top==Array<Elem>::size()-1;
   }
// Видати поточний розмір стека
   size_t size() const {return _top+1;}
// Видати ємність контейнера
   size_t volume() const {
      return Array<Elem>::size();  // Звертаємося до функції базового
                                   // класу, адаптуючи її ім'я
   }
// Керування стеком
// Видати верхівку стека
```

```
   const Elem& top() const {
      return (*this)[_top]; // До стека як представника класу,
       // похідного від масиву, застосовуємо його операцію індексування
   }
```

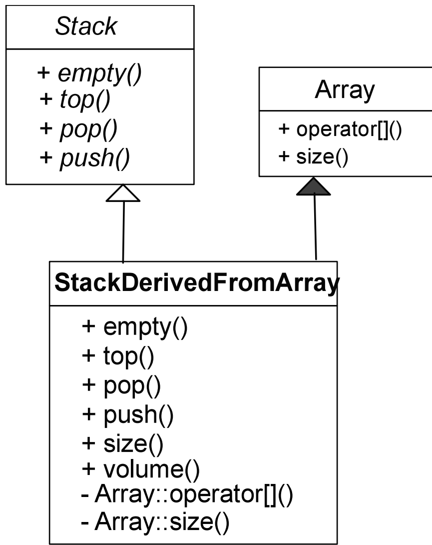


Рис. 7.22. Кратне успадкування інтерфейсу і реалізації

```
// Виштовхнути верхівку стека
   void pop() {_top--;}
// Помістити до стека
   void push(Elem value) {
// Знову застосовуємо операцію
// індексування
      (*this)[++_top]=value;
   }
}
template <class Elem>
const size_t
   StackDerivedFromArray<Elem>::
   _bos=-1;
```

Побудовану ієрархію зображено на рис. 7.22.

Тепер у контексті оголошення функції

```
template <typename Elem>
   int process(Stack<Elem>&
   stackToProcess);
```

можна записати її виклик до стека, похідного від масиву:

Лістинг 7.74

```
StackDerivedFromArray<int> stack(100);
process(stack);
```

або до стека з вбудованим масивом:

Лістинг 7.75

```
StackContainingArray<int> stack(100);
process(stack);
```

Ось простий приклад реалізації функції обробки стека, яка полягає в обчисленні суми його елементів:

Лістинг 7.76

```
// Функція підрахунку суми елементів стека,
// визначена з використанням абстрактного стека
int process(Stack<int>& stackToProcess) {
   int sum=0;
```

```
   while(!stackToProcess.empty()) {
      sum+=stackToProcess.top(); stackToProcess.pop();
   }
   return sum;
}
```

Вправа 7.31

Реалізуйте варіант функції підрахунку суми елементів стека, яка залишає його вміст незмінним.

Функцію, яка використовує абстрактні класи, максимально налаштовано на нові застосування. Адже вона не містить зайвих припущень про реалізацію класу, до якого її застосовують (у розгляданому випадку – стека), способу його реалізації, його обмеженості тощо, а тому придатна для обробки стеків будь-яких типів, зокрема нових, що можуть з'явитися вже після її створення й навіть компіляції. Система, яка виконуватиме таку функцію, має бути здатна до відкладеного зв'язування виклику функції залежно від конкретного типу фактичного параметра (наперед не відомого).

Розглянемо ще один простий приклад. Звернімося до схеми, зображеної на рис. 7.23.

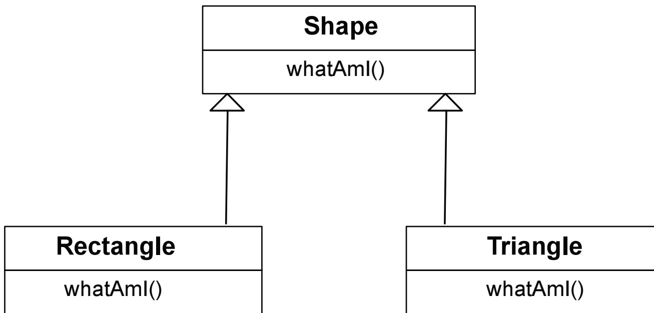


Рис. 7.23. Ієрархія геометричних фігур

Тут немає абстрактних класів, але використано віртуальні функції. На лістингу 7.77 подано можливий варіант визначення класів зі схеми рис. 7.23 в разі застосування раннього (статичного) зв'язування.

Лістинг 7.77

```
// Клас фігур невідомого типу
class Shape {
public:
// Виведення типу фігури
   void whatAmI() {
```

```
      cout<<"I don't know what kind of shape I am! "<<endl;
   }
}
// Клас прямокутників
class Rectangle: public Shape {
public:
// Виведення типу фігури
   void whatAmI() {cout<<"I'm a rectangle! "<<endl;}
};
// Клас трикутників
class Triangle: public Shape {
public:
// Виведення типу фігури
   void whatAmI() {cout<<"I'm a triangle! "<<endl;}
};
```

Запропоновану реалізацію важко назвати вдалою. Як видно з наступного лістингу, правильну відповідь про тип фігури вдається одержати не завжди, позаяк один і той самий об'єкт – прямокутник `s` – по-різному реагує на спробу застосувати до нього метод `whatAmI()`.

Лістинг 7.78

```
// Використання раннього зв'язування
int main() {
   Shape* p; Rectangle s(10, 20);
   s.whatAmI();  // Виклик функції прямокутника
   p=&s;
   p->whatAmI(); // Виклик функції фігури невідомого типу
   return 0;
}
```

Справді, статичне зв'язування не може адекватно реагувати на тип значення указника. Надамо тепер функції виведення типу фігури статус віртуальної:

Лістинг 7.79

```
// Клас фігур невідомого типу
class Shape {
public:
// Віртуальна функція виведення типу фігури
   virtual void whatAmI() {
      cout<<"I don't know what kind of shape I am! "<<endl;
   }
};
// Клас прямокутників
class Rectangle: public Shape {
```

```
public:
// Віртуальна функція виведення типу фігури
   void whatAmI() {cout<<"I'm a rectangle! "<<endl;}
};
// Клас трикутників
class Triangle: public Shape {
public:
// Віртуальна функція виведення типу фігури
   void whatAmI() {cout<<"I'm a triangle! "<<endl;}
};
```

Ось простий приклад, який демонструє результат викликів віртуальної функції `whatAmI()` засобами пізнього (динамічного) зв'язування:

Лістинг 7.80

```
// Використання пізнього зв'язування
int main() {
   Shape *s1, *s2, *s3;
   s1=new Rectangle; s2=new Triangle; s3=new Shape;
   s1->whatAmI(); // Виклик функції виведення прямокутника
   s2->whatAmI(); // Виклик функції виведення трикутника
   s3->whatAmI(); // Виклик функції фігури невідомого типу
   return 0;
}
```

Зверніть увагу, що в усіх трьох випадках виклику методу `whatAmI` використано указник одного й того самого типу, але буде викликано різні методи залежно від типу значення, на яке встановлено указник.

Вправа 7.32

Запрограмуйте й випробуйте меню створення фігури на замовлення користувача, виражене в діалозі з програмою.

У розглянутих прикладах бачимо прояви поліморфізму – можливості використання об'єктів одного типу (наприклад, прямокутників або трикутників) замість об'єктів іншого типу (фігур). Конкретно поліморфізм проявляється як можливість викликати функції похідного класу за допомогою інтерфейсу базового класу. Для реалізації поліморфізму потрібен механізм динамічного розпізнавання типу об'єкта. Справді, у наведеному прикладі кожен із трьох указників має один і той самий тип `Shape`, але їхні значення відрізняються типами: для `s1` це `Rectangle`, для `s2` – `Triangle` та, нарешті, для `s3` – сам тип `Shape`. Отже, виклик віртуальної функції має не прямо виконуватись, а бути схожим на виклик функції за указником, який відсилає до методу в потрібному класі. Для реалізації такого доступу кожен клас із

віртуальними функціями наділено спеціальною таблицею віртуальних функцій, що вже відсилає до самих функцій.

Тому виклик віртуальної функції має складатися з двох кроків: 1) на етапі виконання програми перед викликом функції слід визначити клас, якому належить об'єкт, до котрого застосовано цю функцію; 2) потрібно звернутися до таблиці віртуальних функцій цього класу та взяти з неї потрібну функцію. Статично (до виконання програми) цього зробити не можна, тому що одна й та сама команда виклику функції може виконуватися багаторазово та щоразу використовувати об'єкт іншого типу.

Наступний приклад демонструє, що навіть один і той самий програмний код може залежно від стану процесу виконання програми зумовлювати виклики різних функцій.

Лістинг 7.81

```
// Гра з фігурами
int main() {
   Shape* s[10]; char chr;
// Створення фігур на замовлення
   for (int i=0; i<10; ++i) {
      cout<<"what to place to s["<<i<<"]? Aswer s/r/t"; cin>>chr;
      switch (chr) {
      case 's': s[i]=new Shape; break;
      case 'r': s[i]=new Rectangle; break;
      case 't': s[i]=new Triangle; break;
      default: --i;
      };
   }
   for(i=0; i<10; ++i)     // Виведення створених фігур
      s[i]->whatAmI(); // У цій команді кожного разу може виконуватись
                       // інший код
   return 0;
}
```

Особливу роль у поліморфізмі відіграють *віртуальні деструктори*. Вони призначені для встановлення правильної послідовності виклику деструкторів у разі поліморфного використання об'єктів. Розглянемо простий приклад успадкування, у якому як базовий, так і похідний класи використовують динамічну пам'ять.

Лістинг 7.82

```
// Базовий клас
class Person {
   char* _name;
public:
```

```
// Конструктор
   Person (const char* name): _name(new char[strlen(name)+1]) {
      strcpy(_name, name);
   }
// Віртуальний деструктор
   virtual ~Person() {delete [] _name;}
};
// Похідний клас
class Student: public Person {
   char* _inmatriculation;
public:
// Конструктор
   Student (const char* name, const char* m):
      Person(name), _inmatriculation(new char[strlen(m)+1]) {
      strcpy(_inmatriculation, m);
   }
// Віртуальний деструктор
   virtual ~Student() {delete [] _inmatriculation;}
};
```

Створимо об'єкт похідного класу за указником:

```
Person* p=new Student("Qwerty", "123456789");
```

Зрозуміло, що виклик конструктора `Student("Qwerty", "123456789")` передбачає виконання обох конструкторів як похідного, так і базового класів. Виникає запитання: як бути з викликом деструктора командою **delete** p? Ось тут виявляється потрібним віртуальний деструктор: у цьому разі завдяки поліморфізму буде викликано деструктор об'єкта класу студентів, який видалить свою частину динамічної пам'яті, а саме значення указника `_inmatriculation`, а потім на загальних підставах відпрацювання деструкторів уздовж ланцюжка успадкувань спрацює деструктор базового класу, який видалить значення указника `_name`. Отже, для застосовування поліморфізму потрібні віртуальні деструктори.

Вправа 7.33

Протестуйте видалення об'єктів класів `Person` і `Student` у разі використання віртуальних і невіртуальних деструкторів.

7.10. Архітектура ієрархій об'єктів і класів

У попередньому розділі було з'ясовано, що відкриту віртуальну функцію можна розглядати водночас не тільки як частину інтерфейсу класу, який надано зовнішньому коду, але і як засіб налаштувати його по-

ведінку, наданий похідним класам. Використання цієї, здавалось би, переваги призводить до суттєвого зниження якості програмного забезпечення. Доцільно поділити частини базового класу за цільовим призначенням, виділивши ту, яку призначено для успадкування інтерфейсу, і відділивши її від частини, що забезпечуватиме налаштування поведінки. Пам'ятаймо, що успадкування інтерфейсу призначене для клієнтів класу, а налаштування поведінки – для його похідних класів. Якість архітектури підвищиться, якщо заборонити користувачам різного типу (клієнтам класу та похідним класам) застосовувати один і той самий засіб з різною метою. Оскільки інтерфейсна функція відкрита, то вона не повинна вживатися для налаштування поведінки, а тому не варто робити її віртуальною.

Тому рекомендовано визначати інтерфейсні функції як відкриті невіртуальні. Тоді всі похідні класи використовують спільний інтерфейс, не призначений для налаштування поведінки. Це правило відоме під назвою *ідіоми невіртуального інтерфейсу NVI* (Non Virtual Interface).

Отже, завдяки ідіомі невіртуального інтерфейсу у відкритій частині класу не може з'явитися жодної віртуальної функції. Проте саме це правило ще не гарантує стійкої стосовно використаного інтерфейсу поведінки об'єктів. Як відомо з попереднього розділу, неоднозначність поведінки виявляється в разі перевизначення відкритих невіртуальних базових функцій у похідних класах.

Тому запровадимо ще одне правило невіртуальності інтерфейсу, а саме: інтерфейсні функції не можна перевизначати в похідних класах. Його називають *ідіомою вільного від заміщень інтерфейсу*.

Іще раз наголосимо, що введені нами норми визначення структури класів виділяють з усіх можливих синтаксично правильно побудованих класів ті, будова яких коректна з прагматичних позицій. Із синтаксичного погляду відкрита функція може з'явитись у будь-якому класі як така, котру довільний клієнт класу може використати з будь-якою метою. У контексті ідіоми невіртуального інтерфейсу відкрита функція – це така, яку призначено клієнтові для визначення спільної точки входу до всіх різновидів поліморфного типу.

Аналогічно, реалізацію будь-якої відкритої функції базового класу можна синтаксично змінити в похідному класі. Із погляду ідіоми вільного від заміщень інтерфейсу така структура успадкувань перестає відповідати прагматичним вимогам до їхньої архітектури.

Застосуємо ідіому NVI до ієрархії «прямокутник – квадрат».

Лістинг 7.83

```
typedef unsigned int length_t;
// Клас прямокутників, розділений на інтерфейс і налаштування
class NVIRectangle {
// Відкритий інтерфейс NVI
public:
// Конструктор
   NVIRectangle(const length_t a, const length_t b): _a(a), _b(b) {}
   NVIRectangle() {}  // Деструктор
// Селектори:
// реалізований у базовому класі
   length_t getA() const {return _a;}
// призначений для налаштування
   length_t getB() const {return do_getB();}
// Модифікатори:
// реалізований у базовому класі
   void setA(const length_t a) {_a=a;}
// призначений для налаштування
   void setB(const length_t b) {return do_setB(b);}
// Обчислення площі прямокутника
   length_t area() const {return do_area();}
private:
   length_t _a, _b;  // Сторони прямокутника
// Закрите налаштування
// Налаштування селектора другої сторони
   virtual length_t do_getB() const {return _b;}
// Налаштування модифікатора другої сторони
   virtual void do_setB(const length_t b) {return _b=b;}
// Налаштування функції обчислення площі
   virtual length_t do_area() const {return getA()*getB();}
};
// Клієнтський код
void increase_a(NVIRectangle& r, const unsigned int m) {
   r.setA(r.getA()*m);
}
```

Тепер похідний клас квадратів здобуває право налагоджувати віртуальні функції `do_getB()`, `do_setB()` та `do_area()`, але не може заміщувати невіртуальні функції `getA()`, `setA()` й `area()`, як і утиліту `increase_a()`, яку, з огляду на її відкритий статус, прирівнюють до інтерфейсу.

Лістинг 7.84

```
// Клас квадратів
// Успадкування інтерфейсу та налаштування реалізації
class NVISquare: public NVIRectangle {
```

```
public:
// Конструктор
   NVISquare(const length_t x): NVIRectangle(x, x) {
// Якщо це судження не справджується, в селекторах припущено помилки
      assert(getA()==getB());
   }
   ~NVISquare() {} // Деструктор
private:
// Налаштування поведінки прямокутника на поведінку, властиву квадрату
// Налаштування селектора другої сторони
   virtual length_t do_getB() const {return getA();}
// Налаштування модифікатора другої сторони
   virtual void do_setB(const length_t b) {return setA(b);}
// Налаштування функції обчислення площі
   virtual length_t do_area() const {return getA()*getA();}
};
```

Проаналізуємо поведінку кожного з класів.

Застосування ідіоми невіртуального інтерфейсу гарантує, що в контексті як прямокутника, так і квадрата буде використано один і той самий інтерфейс. Справді, візьмемо код, який оперуватиме безпосередньо з квадратом –

```
NVISquare sqr(10);
```

і викличемо модифікатор сторони `a`:

```
sqr.setA(15);
```

Оскільки квадрат не має власних відкритих функцій, окрім конструктора та деструктора, то буде викликано функцію базового класу `NVIRectangle::setA(15)`, яку в ньому ж і реалізовано. Тепер обчислюємо площу:

```
cout<<sqr.area()<<endl;
```

Це знову ж функція класу прямокутників `NVIRectangle::area()`, але, на відміну від попередньої, її не реалізовано в ньому, а призначено для налаштування за допомогою закритої віртуальної функції. У виклику `do_area()` закрита віртуальна функція класу прямокутників `NVIRectangle::do_area()` заміщується такою самою функцією класу квадратів `NVISquare::do_area()`. Якщо тепер викликати не метод, а утиліту

```
increase_a(sqr, 2);
```

то відповідно збільшиться сторона a прямокутника. У разі ж спроби вивести іншу сторону –

```
cout<<sqr.getB()<<endl;
```

виклик do_getB() знову зумовить її заміщення в класі квадратів, а тому врешті-решт буде виконано функцію NVIRectangle:: getA(), яка забезпечить потрібний результат.

Нічого не зміниться, якщо відразу застосувати інтерфейс прямокутника:

Лістинг 7.85

```
NVIRectangle* rec=new NVISquare(10);
rec->setA(15); cout<<rec->area()<<endl;
increase_a(rec, 2); cout<<rec->getB()<<endl;
```

Буде викликано всі ті самі функції в такому самому порядку, оскільки всі зовнішні виклики завжди були викликами методів інтерфейсного класу прямокутників.

Вправа 7.34

Застосуйте ідіому невіртуального вільного від налаштування інтерфейсу до класу фігур і його похідних класів.

Отже, віртуальні функції краще закрити. Тоді похідні класи можуть пропонувати власну реалізацію закритої й тому недоступної для клієнтського коду частини базового класу. Приходимо до *ідіоми закритих віртуальних налаштувань*.

У найзагальнішому вигляді цей важливий *архітектурний принцип успадкувань інтерфейсу* можна сформулювати так: потрібно використовувати невіртуальний вільний від заміщень інтерфейс у поєднанні із закритими віртуальними налаштуваннями.

Вправа 7.35

Застосуйте архітектурний принцип успадкувань інтерфейсу до ієрархій «паралелограм – прямокутник», «ромб – квадрат».

Із правила закритих віртуальних налаштувань є один виняток, який стосується деструкторів: вони хоч і відкриті, але мають бути віртуальними, щоб вивільнялася пам'ять, зайнята похідними об'єктами.

До запропонованої ієрархії все ще можна зробити зауваження. Перевага об'єктно-орієнтованого підходу, як уже було зазначено, полягає в побудові таких ієрархій, які сприятимуть зменшенню, а точніше впорядкуванню складності великих програмних систем. Розглядаючи ієрархію в цьому ас-

пекті, можна помітити, що структура похідних класів зовсім не впливає на структуру базових, тоді як самі похідні класи суттєво залежать від базових. Це означає, що надмірна конкретизація базових класів звужує сферу застосування їхніх похідних класів, а спроба модифікувати базовий клас (наприклад, із метою узагальнення, щоб розширити сферу його застосувань) зумовить потребу переглянути всі базові класи. У цьому розумінні модифікація базових класів у будь-якій програмній системі – найвитратніша процедура, оскільки вона стосується всієї архітектурної піраміди.

У кожній ієрархії класів можна виділити *кореневий клас* (краще, коли він один), який породжує всі інші класи в ієрархії, і *термінальні класи*, які не мають похідних. Як кореневий доцільно використовувати абстрактний клас із невіртуальним інтерфейсом. Виявляється, усі інші нетермінальні класи теж мають бути абстрактними.

Виходячи зі сказаного, будемо намагатися задавати нетермінальні класи якомога неконкретними. Як міру конкретності класу запропонуємо взяти його реальність, тобто здатність класу до створення власних екземплярів. Говорячи формально, наявність у класі хоча б однієї чисто віртуальної функції відразу перетворює його на абстрактний, який не може створювати самостійні екземпляри. Оскільки деструктор кожного базового інтерфейсного класу має бути віртуальним, то, формально перетворивши деструктор на абстрактний метод, можна зробити будь-який клас абстрактним. Зрозуміло, що таким способом не вдасться досягнути підвищення змістовної абстрактності класу, але принаймні стане неможливим існування його незалежних екземплярів.

Приходимо до ідіоми застосування *абстрактного деструктора*: у нетермінальному класі має бути абстрактний деструктор (тобто віртуальний, позначений нулем).

У кожному ланцюжку успадкувань можна виділити класи, від яких не успадковано жодного похідного класу. Ми назвали їх термінальними. Термінальність – це не властивість класу, адже будь-коли хто завгодно може формально створити з нього похідний. Щоб усунути можливість подальших спадкувань необхідно закрити його конструктор. Приходимо до ідіоми *примусової термінальності класу*: примусово термінальним називатимемо клас, у якому функції конструкторів виконують так звані *псевдоконструктори* (pseudo constructor), які також називають *функціями клонування* (clone).

Наведемо приклад структури типового примусово термінального класу, наділеного псевдоконструктором:

```
// Термінальний клас, у якого не може бути похідних класів   Лістинг 7.86
class Terminal {
public:
   static Terminal* clone(); // Псевдоконструктор
   static Terminal* clone(const Terminal&); // Псевдоконструктор копіювання
private:
// Закриті, а тому недоступні потенційним похідним класам:
   Terminal();  // Конструктор
   Terminal(const Terminal&);  // Конструктор копіювання
};
// Реалізація псевдоконструкторів
Terminal* Terminal::clone() {return new Terminal();}
Terminal* Terminal::clone(const Terminal& a) {return new Terminal(a);}
```

Наявність неабстрактних класів на ланцюжку спадкувань приводить по появи об'єктів з різних його рівнів. Так поява екземпляру класу NVIRectangle одночасно з NVISquare приводить до виникнення проблем забутливого присвоєння (підрозділ 7.3). Тому ієрархію прямокутник-квадрат варто розділити на дві частини: абстрактний інтерфейс ний клас і конкретні термінальні класи.

Поєднання ідіом невіртуального інтерфейсу та нетермінальної абстрактності зумовлюють виникнення поняття *стабільного базового інтерфейсу* – невіртуального, вільного від заміщень інтерфейсу абстрактного базового класу з віртуальним деструктором.

Стабільний базовий інтерфейс гарантує можливість *нових застосувань* (reuse) лише інтерфейсу, а не реалізації. Тому немає сенсу наповнювати базовий клас атрибутами та реалізаціями методів. У цьому разі його типовий склад – чисто віртуальні закриті функції для налаштування відкритого інтерфейсу. Це мінімізує припущення про базовий клас, а отже, уможливлює якнайширше коло застосувань.

```
// Стабільний базовий інтерфейс прямокутників         Лістинг 7.87
class InterfaceRectangle {
public:
   InterfaceRectangle() {}  // Конструктор
   virtual ~InterfaceRectangle() {}  // Деструктор
// Стабільний невіртуальний інтерфейс
   unsigned int& a() {return do_a();}
   unsigned int& b() {return do_b();}
   const unsigned int& a() const {return do_a();}
   const unsigned int& b() const {return do_b();}
   unsigned int area() const {return do_area();}
```

```
private:
// Налаштування реалізації
   virtual unsigned int& do_a()=0;
   virtual const unsigned int& do_a() const=0;
   virtual unsigned int& do_b()=0;
   virtual const unsigned int& do_b() const=0;
   virtual unsigned int do_area() const=0;
};
// Функція масштабування сторони a
void increase_a(InterfaceRectangle& r, const unsigned int m) {
   r.a()*=m;
}
// Функція масштабування сторони b
void increase_b(InterfaceRectangle& r, const unsigned int m) {
   r.b()*=m;
}
```

Тепер класи як прямокутників, так і квадратів стають повноцінними різновидами поліморфного типу, заданого стабільним базовим інтерфейсом: у них немає власних відкритих функцій (крім стандартного набору конструкторів, деструктора та присвоєнь). Досягнута гармонія в ієрархії полягає в гарантованому використанні єдиного інтерфейсу незалежно від способів доступу до похідних об'єктів, а закритий характер віртуальних функцій перетворює налаштування поведінки на внутрішню, закриту від клієнта, проблему реалізації похідних класів.

Лістинг 7.88

```
// Термінальний клас прямокутників
class Rectangle: public InterfaceRectangle {
public:
// Конструктор
   Rectangle(const unsigned int a, const unsigned int b): _a(a), _b(b) {}
private:
   unsigned int _a, _b; // Сторони
// Закрите налаштування
   virtual unsigned int& do_a() {return _a;}
   virtual const unsigned int& do_a() const  {return _a;}
   virtual unsigned int& do_b() {return _b;}
   virtual const unsigned int& do_b() const {return _b;}
   virtual unsigned int do_area() const {return _a*_b;}
};
// Термінальний клас квадратів
class Square: public InterfaceRectangle {
```

```
public:
   Square(const unsigned int x): _a(x) {}  // Конструктор
private:
   unsigned int _a; // Сторона
// Закрите налаштування
   virtual unsigned int& do_a() {return _a;}
   virtual const unsigned int& do_a() const {return _a;}
   virtual unsigned int& do_b() {return _a;}
   virtual const unsigned int& do_b() const {return _a;}
   virtual unsigned int do_area() const {return _a*_a;}
};
```

Вправа 7.36

Доповніть класи ієрархії прямокутників програмованими конструкторами, деструкторами та присвоєннями.

Вправа 7.37

Застосуйте до класів ієрархії прямокутників ідіому нетермінальної абстрактності, заборонивши можливість створення класів, похідних від неінтерфейсних.

Застосуємо стабільний базовий інтерфейс до класу стеків. Визначимо абстрактний інтерфейсний клас (порівняйте з лістингом 7.71):

Лістинг 7.89

```
// Стабільний базовий інтерфейс стека елементів типу Elem
template <class Elem> class Stack {
public:
   Stack() {}  // Конструктор стека
   virtual ~Stack() {}  // Деструктор стека
   bool empty() const {return doEmpty();} // Перевірити, чи порожній стек
   const Elem& top() const {return doTop();} // Показати верхівку стека
   void pop() {doPop();}  // Виштовхнути верхівку стека
   void push(const Elem& el) {doPush(el);} // Проштовхнути елемент у стек
private:
// Налаштування поведінки стеків
   virtual bool doEmpty() const=0;
   virtual const Elem& doTop() const=0;
   virtual void doPop()=0;
   virtual void doPush(const Elem&)=0;
// Закриті нереалізовані операції
   Stack operator=(const Stack&);
};
```

Відкрите успадкування як засіб підтримки типізації всім похідним класам надає доступ до стабільного спільного інтерфейсу. Для налаштування базового класу використовують його закриті віртуальні методи. Головна перевага стабільного базового інтерфейсу полягає в тому, що можливі різні його реалізації, без жодних обмежень. Можна реалізувати стек на вбудованому контейнері, наприклад масиві чи списку, або застосувати закрите чи захищене успадкування від контейнера (рис. 8.9). Утім найголовніше те, що абстрактний клас не лише не має обмежень на нові застосування для самого себе, але й гарантує нові застосування будь-якому з клієнтів, які використовують його як інтерфейс.

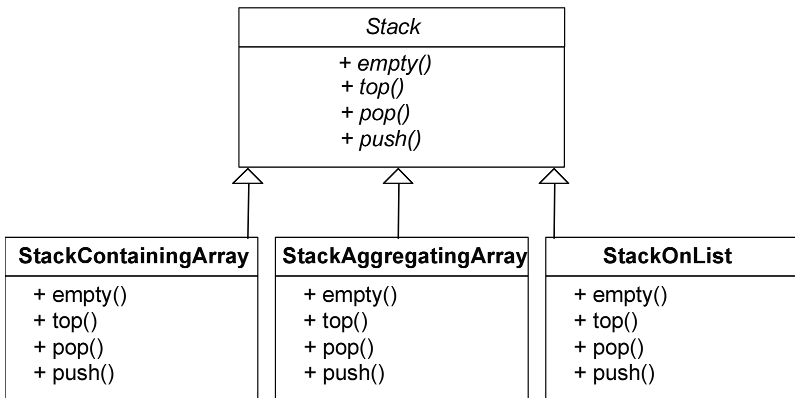


Рис. 7.24. Діаграма стеків, похідних від абстрактного класу

Вправа 7.38

Реалізуйте стабільний базовий інтерфейс стека для стеків, наведених на схемі класів (рис. 7.24): із вбудованим масивом, з агрегованим масивом і на базі списку.

Завдання до розділу 7

1. Визначте абстрактний клас трикутників і дві його реалізації: за трьома вершинами і за трьома сторонами.
2. Доповніть клас трикутників конструктором за основами його трьох медіан. Реалізуйте метод обчислення довільної медіани трикутника.
3. За довільною стороною трикутника знайдіть паралельну їй середню лінію.
4. Визначте ієрархію трикутник-прямокутний трикутник. Реалізуйте функції обчислення довжин радіусів вписаного і описаного кіл.

5. Визначте абстрактний клас трикутних призм `Prism`, передбачивши функції доступу до його складових: основ `Base`, бічних граней `Face`, ребер `Edge` і вершин `Vertex`.
6. Визначте абстрактний клас трикутних пірамід `Pyramid` і побудуйте ієрархію: піраміда, правильна піраміда, зрізана піраміда.
7. Визначте клас двосторонніх черг `DeqList` на базі списків.
8. Визначте клас двосторонніх черг `DeqArray` на базі масивів.
9. Організуйте абстрактний клас інтерфейсу черг `Queue` для обох типів черг.
10. Включіть до інтерфейсу і реалізуйте в кожному з класів функцію інвертування черги.
11. Визначте ієрархію черг, що міститиме абстрактну чергу, чергу з підгляданням `PeekBackQueue` і чергу з шахраюванням `CheatQueue` (остання дає змогу не лише підглянути, але й замінити заданий елемент черги).
12. Визначте абстрактний клас виробів `Goods`. Кожен виріб має інвентарний номер (штрих-код), дату виготовлення, вартість, назву і додаткові характеристики. Вироби можна виводити. Виводяться штрих-код, дата виготовлення, вартість, назва і додаткові характеристики.
13. На базі класу `Goods` визначте конкретний клас продуктів харчування `Product`. Додатковою характеристикою є термін придатності.
14. На базі класу `Goods` визначте конкретний клас предметів одягу `Clothes`. Додатковою характеристикою є розмір.
15. Визначте комору `Storehouse` для зберігання виробів, відсортованих за штрих-кодами. В кожній програмі існує лише одна комора. Визначте функцію виведення інвентарної відомості комори.
16. Реалізуйте метод `find()` підрахунку за даним штрих-кодом кількості виробів у коморі.
17. Реалізуйте метод `include()` занесення виробу до комори.
18. Реалізуйте метод `exclude()` вилучення виробу з комори.
19. Запрограмуйте створення і обробку аварійних ситуацій а) `Lack`, спричиненої наявністю критичної кількості виробів з даним штрих-кодом в коморі; б) `ShelfLife` – критичністю терміну зберігання.
20. Визначте клас касових апаратів `CashRegister` для підрахунку вартості та створення чеку покупки за прейскурантами комори.
21. Запропонуйте засоби підрахунку скидок залежно від: а) терміну придатності продукту; б) кількості одиниць виробів в одному чеку; в) загальної вартості всього чеку.
22. Визначте клас плоских фігур `Shape` на координатній площині, доповнивши типову структуру класу мультиконструктором копіювання, вір-

туальними утилітами введення, виведення, дзеркального відображення відносно осей координат, центрально-симетричного відображення відносно початку координат, повороту, паралельного переносу.

23. Клас многокутників `Polygon` є варіантом класу плоских фігур. Запропонуйте дві реалізації класу `Polygon` на базі масиву і на базі списку. Реалізуйте утиліту дзеркального відображення многокутника за довільною віссю координат.
24. В коло заданого радіуса r з центром у початку координат вписано правильний многокутник. Реалізуйте клас `RegPolygon` правильних вписаних многокутників, спираючись на поведінку класу `Shape`. Конструктор будує його вершини в напрямку, протилежному руху годинникової стрілки, починаючи з вершини $(r,0)$.
25. Реалізуйте в класі `RegPolygon` віртуальні утиліти введення, виведення, дзеркального відображення відносно осей координат, центрально-симетричного відображення відносно початку координат, повороту, паралельного переносу.
26. Визначте абстрактний ітератор та запрограмуйте з його допомогою злиття двох впорядкованих списків довільного виду в один.
27. Визначте клас `Functor` функторів – параметричних елементарних функцій-методів класу виду $y=f(a,x)$. Параметр `a` реалізувати атрибутом класу, а аргумент `x` – формальним параметром методу. Передбачити методи `myexp` для a^x, `mylog` для $\log_a x$, `mysin` для $\sin ax$, `mycos` для $\cos ax$.
28. Визначте клас `Oper` калькулятора арифметичних операцій над елементарними функціями, передбачивши в ньому метод `execute` виконання довільної арифметичної операції над довільною парою функцій в заданій точці x.
29. Реалізуйте функцію `tabulate` для табулювання значень довільної елементарної функції або результату арифметичної операції над функціями класу `Functor` на інтервалі $[x_0, x_1]$.

Навчальні проекти

1. Задача побудови опуклої оболонки множини точок на площині та обчислення її числових характеристик: площі і периметра.
   Опукла оболонка визначається рекурсивно:

   Convex(Ø) = Ø, де Ø – порожня множина точок;
   Convex($\{a\}$) = $\{a\}$, де a – точка площини;
   Convex($\{a, b\}$) = $\{a, b\}$, де a і b – точки площини;
   Convex($\{a_1, a_2, \ldots, a_n, a_{n+1}\}$) = Convex($\{$Convex($\{a_1, a_2, \ldots, a_n\}$), $a_{n+1}\}$), n=2,3,…, де $a_1, a_2, \ldots, a_n, a_{n+1}$ – точки площини.

Визначте ієрархії об’єктів і класів для реалізації опуклої оболонки довільної послідовності точок площини. Реалізуйте проект.

2. Компілятор арифметичних формул. Арифметичні формули – це вирази, побудовані за допомогою арифметичних операцій з числових змінних і сталих з використанням дужок для визначення порядку виконання операцій. Вони визначаються індуктивно. Змінна *x* – це формула, стала *c* – теж формула. Якщо *a* і *b* – формули, то (*a*),+*a*, –*a*, *a*+*b*, *a*–*b*, *a**b*, *a*/*b* – теж формули. На вхід калькулятора арифметичних формул надходять арифметична формула і набір значень змінних, на виході – значення цієї формули при заданому наборі значень аргументів. Визначте ієрархію об’єктів і класів для реалізації калькулятора арифметичних формул. Реалізуйте проект.
3. Квазіузагальнений стек. З метою уникнення розбухання коду замовник попросив розробника зберігати в контейнері стеку безтипові указники об’єктів `void*`. Спроектуйте абстрактний параметризований клас стеків, спираючись на клас стеку безтипових указників. Потурбуйтесь про збережність об’єктів, указники яких будуть заноситися до стеку, до моменту їх вилучення зі стеку. Захистіть стек безтипових указників від спроби його створення і використання поза інтерфейсним параметризованим класом. Обґрунтуйте обраний спосіб організації ієрархії.
4. На умовах проекту квазіузагальненого стеку (3) спроектуйте архітектуру параметризованих класів черг з пріоритетами (завдання 1, 3, 18 до розділу 6) на основі класу циклічного масиву безтипових указників. Додайте до черги операцію підглядання.
5. Виконайте рефакторинг класів банківських рахунків з підрозділу 7.3 (лістинги 7.27–7.30) з метою використання ідіоми вільного від заміщень (стабільного) невіртуального інтерфейсу.

Заключне слово

На завершення курсу дамо кілька рекомендацій стосовних подальших студій. Проектуючи клас як абстрактну модель певної сутності, варто замислюватися над тим, що має становити істотні риси цієї сутності з погляду їхнього відтворення в моделі. Це чи не найскладніший етап розроблення програмних систем, оскільки все, що не потрапить до моделі, опиниться за межами програми. Надто поверхова абстракція може виявитися неадекватною до сутності, надто детальна – надмірною та переобтяженою несуттєвими деталями для задач, розв'язуваних із застосуванням проектованої моделі. Тому спочатку складають словники для позначення класів, їхніх властивостей і елементів поведінки. Далі приймають рішення, яка частина потенційної поведінки несуттєва з погляду клієнта, а яка – взагалі зайва для класу. Не призначені для використання клієнтом методи можуть потрапити до закритої частини класу, якщо вони важливі для його реалізації. Конче потрібно адекватно визначити *бар'єр абстракції*, наділяючи клас важливою змістовною поведінкою та не допускаючи його несподіваної поведінки. Бар'єр абстракції ґрунтується на відомому з ергономіки принципі *мінімізації сюрпризів* (least astonishment). Тому у великих колективах класи та зв'язки між ними проектують найкваліфікованіші розробники, яких називають системними аналітиками.

Принцип мінімізації сюрпризів містить кілька загальних вимог, які застосовують до проектування класів. Передусім потрібно дотримуватися стандартної схеми структури класу, точно регламентуючи призначення стандартних методів: конструкторів, деструкторів, присвоєння. Джерелом сюрпризів може стати, наприклад, застосування стандартних методів, наданих компілятором. Тому, коли потрібно заборонити копіювання об'єктів, відповідний конструктор і присвоєння роблять недоступними, виносячи їх до закритої частини. Програмуючи стандартні операції, варто зберігати властивості їх реалізації в базових типах. Скажімо, додавання має бути комутативним, доступ за індексом і розіменування – враховувати особливості адресної арифметики, нерівність має бути запереченням рівності тощо.

Ніхто не стане заперечувати, що створення якісного програмного забезпечення – найважливіше завдання розробників програмних систем. Якістю програм у софтверних фірмах займаються не менші колективи, ніж персонал власне розробників. За якими критеріями варто оцінювати якість програмного забезпечення? Певна річ, дуже важлива саме довершеність кодів функцій: добротність закладених до них алгоритмів, майстерність кодування розгалужень і циклів, використані структури даних, уживання значущих імен, зрозумілість, очевидність і читабельність коду.

Однак усі ці властивості мають локальний характер, вони стосуються лише окремих частин програмної системи. Тому, вивчивши за цим підручником свого роду міні-програмування, варто перейти до оволодіння глобальними характеристиками якості програмних систем.

Говорячи про якість, часто наводять аналогії з будівництвом. Будівельники можуть використовувати дорогі та якісні оздоблювальні матеріали, кахель, сантехніку, паркет, електричні кабелі, тощо, але краса і якість будівлі залежить не від цього. Тісні коридори, затемнені кімнати, невдало розміщені служби, недоступні для провітрювання сходові клітки, брак комунікацій, службових приміщень і автостоянок зведуть нанівець ефект від використання дорогих і якісних матеріалів та комплектувальних виробів.

Греді Буч [27] висловився з цього питання так: «Ви як майстер своєї справи можете на одному подиху побудувати зручну хатину. Однак самої лише майстерності недостатньо для зведення висотної будівлі. Мене завжди більше цікавить вишуканість архітектури (програми), а не елегантність алгоритму. Архітектура має бути проста й гнучка. Що більше використано стандартних взірців, то краще».

В списку рекомендованої літератури можна знайти чимало матеріалів, корисних для поглибленого вивчення методів об'єктно-орієнтованого програмування, що є інструментами створення простих і гнучких архітектур. Простота полегшує розуміння будови програмної системи під час її супроводу, а від гнучкості залежить її здатність налагоджуватися на нові застосування (можливо, навіть такі, яких іще не було під час створення системи).

Робота над програмним проектом – зазвичай тривалий процес, упродовж якого доводиться не тільки створювати новий код, але й переглядати наявний для його вдосконалення чи налагодження на нові умови застосування. Такий перегляд називають *рефакторингом*, розуміючи під ним модифікацію коду зі збереженням поведінки програмного проекту в цілому. Перетворений унаслідок рефакторингу програмний код має бути «не гіршим» за початковий. Власне, мета рефакторингу полягає в поліпшенні коду, проте вдосконалення в одному місці може спричинити ускладнення в інших. Тому критерієм вдалого рефакторингу можуть стати перевірки всіх тестових випадків початкового коду. Усі такі перевірки мають бути успішними, а характеристики їх виконання – кращими.

На шляху до справжнього професіоналізму вам доведеться подолати ще чимало перевалів. Щоб досягти вершин майстерності, варто, зокрема, познайомитися з численними інтернет-ресурсами, серед яких особливо виділяється портал Роберта Мартіна **http://www.objectmentor.com/** з його фундаментальним методом створення програмних систем, відомим під назвою SOLID.

Список літератури

1. Н. Вирт, Систематическое программирование. Введение, Москва, «Мир», 1977, 184 с.
2. Б. Керниган, Д. Ричи, Язык программирования С. – Москва: Вильямс, 2006. – 304 с.
3. Б. Страуструп, Программирование. Принципы и практика использования С++, М., «Вильямс», 2011, 1248 с.
4. Б. Страуструп, Дизайн и эволюция С++, М.: ДМК Пресс; Спб.: Питер, 2006. – 448 с.
5. B. Stroustrup, The C++ Programming Language 4th Edition – 2013, 1281 p.
6. У. Дал, Э. Дейкстра, К. Хоор, Структурное программирование, Москва, «Мир», 1975, 246 с.
7. С. Макконнелл, Совершенный код, М., Русская редакция, 2005, 896 с.
8. С. Майерс, Эффективное использование С++, Москва, ДМК Пресс, 2006, 300 с.
9. С. Майерс, Наиболее эффективное использование С++, Москва, ДМК Пресс, 2014, 298 с.
10. Г. Саттер, Решение сложных задач на С++, Москва, Вильямс, 2008, 400 с.
11. Г. Саттер, Новые сложные задачи на С++, Москва, Вильямс, 2005, 272 с.
12. Г. Саттер, А. Александреску, Стандарты программирования на С++, Москва, Вильямс, 2015, 224 с.
13. А. Александреску, Современное проектирование на С++ Москва, Вильямс, 2008. – 336 с.
14. Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. Приемы объектно-ориентированного проектирования. Паттерны проектирования, Санкт-Петербург, Питер, 2007, 366 с.
15. Рекомендации по преподаванию программной инженерии и информатики в університетах, Интернет-Университет Информационных Технологий, Москва, 2007, 472 с.
16. R. Floyd, Paradigms of Programming, Communications of the ACM, August 1979, vol. 22, № 8, pp. 455–460.
17. J. Backus, Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs, Communications of the ACM, August 1978, vol. 21, № 8, pp. 613–641.
18. Н. Вирт, Алгоритмы + структуры данных = программы. Москва, Мир, 1985, 198 с.

19. Н. Вирт, Алгоритмы и структуры данных. Москва, Мир, 1989, 358 с.
20. D. Parnas, On the Criteria To Be Used in Decomposing Systems into Modules, Communications of the ACM, 1972, vol. 15, № 12, 1972, pp.1053–1058.
21. J.V. Guttag, Abstract Data Types and the Development of Data Structures, Communications of the ACM,vol. 20, № 6, 1977, pp. 396–404.
22. Д. Кнут, Искусство программирования, М., «Вильямс», 2007–8.
23. Ф.Л. Бауэр, Р. Гнац, У. Хилл, Информатика: задачи и решения, Москва, «Мир», 1978
24. Д. Грис, Наука программирования, Москва, «Мир», 1984, 416 с.
25. E. Dijkstra, How do we tell truths that might hurt? (EWD-498), in: E. Dijkstra, Selected Writings on Computing: A Personal Perspective, Springer-Verlag, 1982, pp. 129–131.
26. Ф. Брукс, Мифический человеко-месяц или Как создаются программные системы, Символ-Плюс, 2010, 304 с.
27. Г. Буч, Объектно-ориентированный анализ и проектирование с примерами приложений, Вильямс, 2008, 721 с.
28. Інформатика та обчислювальна техніка : Короткий тлумачний словник / В. П. Гондюл, А. Г. Дерев’янко, В. В. Матвєєв, Ю. З. Прохур; за ред. проф. В. П. Гондюла. – К. : Либідь, 2000. – 320 с.
29. Російсько-український словник з інформатики та обчислювальної техніки. /Уклад. В. Я. Карачук, Г. Г. Гульчук, О. О. Карачук, Ю. З. Прохур: За ред.. І. А. Чернетко. – К.: «Рось», 1994. – 362 с.

Глосарій

абстрактний деструктор	abstract destructor	620
абстрактний клас, інтерфейс	abstract class, interface	606
абстрактний тип даних	abstract data type	20
аварійна ситуація bad_alloc	bad_alloc exception	63
агрегація	aggregation	361, 379
агрегація за допомогою відсилок	reference member variables as class members	387
адресна арифметика	address arithmetic	55, 85
Александреску, Андрей	*Andrei Alexandrescu*	11, 542, 630
арифметико-геометричне середнє	arithmetic-geometric mean	75, 204, 447
асоціативний масив	associative array	454
багатосимвольний літерал (рядок символів)	string	44, 89
Бауер, Фрідріх	*Friedrich Bauer*	76, 631
без типу	void	33
безтиповий указник	void pointer	62, 64
Бекус, Джон	*John Backus*	14, 630
бітове поле	bit field	101
блок	block	72
блок захоплення	catch block	79
Брукс, Фредерік	*Frederick Phillips Brooks, Jr,*	545, 631
Буч, Греді	*Grady Booch*	545, 629, 631
визначення функції	function definition	107
виконавча програма	executable program, exe-file	26
випереджальне оголошення	forward declaration	109
відкрита функція	inline function	111, 335, 408
відкритий	public	231
відокремлена компіляція	compilation by separation	502
відсилка (псевдонім)	reference	68
вільна функція, утиліта	non-member function, utility	239, 245
Вірт, Ніклаус	*Niklaus Wirth*	9, 18, 19, 630, 631
віртуальна функція, чисто віртуальна	virtual function, pure virtual	607
віртуальне успадкування	virtual inheritance	603
вісімковий літерал	octal, oct	42
вкладення	layering	361
вміст, одиниця вмісту	content, contained item (containee)	392
вхідний файл	source file	24

Глушков В.М.	*V.M.Glushkov*	74
Гріс, Девід	*David Gries*	77
Гріс, Девід	*David Gries*	77, 631
Гуттаг, Джон	*John Vogel Guttag*	20, 631
Дейкстра, Едсгер	*Edsger Wybe Dijkstra*	10, 545, 630, 631
декларативна парадигма програмування	declarative programming	13
деструктор	destructor	219
деструктор за умовчанням	default destructor	223
динамічне зв'язування	dynamic binding	607
динамічний масив	dynamic array	87
дійсний	float, double	33
дійсні типи з рухомою крапкою	floating point	34
довизначення (поширення) функцій	function overloading	175
дописування програм	programming by difference	22
доповнювальний код	complement-on-two	42
дружній клас, функція	friend class. friend function	332
Єршов А.П.	*A.P. Ershov*	9
завислий указник	dangling pointer	52
заголовок функції	function declaration	107
закрита функція, виклик	function call	116
закрите успадкування	private inheritance	591
закритий	private	231
заміщення методів	overriding	607
захищене успадкування	protected inheritance	588
захищений	protected	577
зміна парадигми	paradigm shift	16
змінюваний	mutable	348
знак зворотної скісної риски	backslash	43
знову вживаний	reusability	22, 549, 621
ідентичність об'єкту	object identity	213
ідіома вільного від заміщень (стабільного) інтерфейсу	stable interface	616, 621
ідіома закритих віртуальних налаштувань	private virtuality	619
ідіома невіртуальногог інтерфейсу	NVI (Non Virtual Interface)	616
іменована константа	const variable	45
імперативна парадигма програмування	imperative programming	13

інваріант	invariant	77
інкапсуляція (вбудовування)	incapsulation	19, 211
інструкція випробувань	try statement	79
інструкція спричинення	throw statement	79
інтегральний тип	integral type	34
ітератор	iterator	87, 454
клас об'єктів	class, object type	213, 238
команда виходу	return statement	110
компілятор	compiler	24
компіляція приєднанням	compilation by including	502
композиція, композит	composition, composite object	361
компонент	component	361
компонувальник	linker	24, 26, 110
конвертор типу	type converter	298
конкретизація шаблону	template specialization	500
конструктор	constructor	219
конструктор за умовчанням	default constructor	223
конструктор копіювання за умовчанням	default copy constructor	254
конструктор копіювання, глибинне копіювання	copy constructor	249, 255
конструктор мультикопіювання	multicopy constructor	258
контейнер	container	392
кратне розгалуження	switch statement	73
кратне успадкування	multiple inheritance	598
лівобічне (іменувальне) значення	left value (lvalue, location value)	47
літерал (буквальний)	literal	41
логічний тип	bool	33
Макконнелл, Стів	*Steven C. McConnell*	11, 17, 630
макрос суджень	assertion statement	77
маніпулятор, монолітний	handler	419, 420
Мартін, Роберт	*Robert Cecil Martin*	629
масив	array	81
Мейерс, Скотт	*Scott Meyers*	11, 630
модифікатор	modifier	239
мультипарадигменність	multi-paradigmality	14
налагоджувальний режим	debug mode	78
налагоджувач	debugger	24
невизначений указник	null pointer	52
необов'язковий атрибут	optional attribute	380
непряме адресування	indirect addressing	51
неявне перетворення типів	implicit type conversion	50

об'єднання	union	102
об'єкт, екземпляр класу	object, class instance	213, 239
об'єктне (модульне) програмування	object programming	15, 19–20
об'єктний код	object code	26
об'єктний файл	object file	25
об'єктно-орієнтоване (ієрархічне) програмування	object-oriented programming	15, 21–24
область видимості	scope	212
одиниця трансляції	translation unit	25
оператор sizeof	sizeof	35
оператор виділення пам'яті	operator new	56
оператор виклику функції	function call operator	446
оператор доступу до члена структури (класу) за указником	arrow operator	96, 425
оператор доступу до члена структури (крапка-оператор)	dot operator	96
оператор звільнення пам'яті	delete statement	62
оператор індексування	subscript operator	281–291, 392, 398, 404, 407, 412, 442
оператор копіювального присвоєння	copy assignment operator	260
оператор присвоєння	assignment operator	46
операторний вираз	expression statement	71
операції над указниками	pointer operator	67
операція адресування	address operator	53
операції логічні	logical operator	38
операції арифметичні	arithmetic operator	37
операція імплікації	ternary conditional operator	39
операція коми	comma operator	37
операції порівняння	comparison operator	38
побітові операції	bitwise operator	39–40
операція розіменування, непрямого доступу	dereferencing operator, indirection operator	55, 424
параметр за замовчуванням	default parameter	176
параметризований клас	parametric class	482
Парнас, Девід	*David Lorge Parnas*	19, 631
передавання параметрів значенням	call-by-value	121
перелік	enumeration	36
переносна програма	portable software	40
перетворення типів	type cast	51
перехід до підпрограми	go to subroutine	110

перехід за міткою	go to	17
підрахунок відсилок	reference counting	460
поведінка об’єкту	object behavior	213
поверхневе присвоєння	simple assignment operator	96
подвійний указник	pointer to pointer	150
поліморфізм	polymorphism	173, 607
порожня інструкція	null statement	71
послідовність	sequence	395
правобічне (іменоване) значення	right value (rvalue)	47
препроцесор	preprocessor	24, 25
принцип звуження методів	single responsibility principle	413
прихована реалізації	pointer to implementation	458
програмний проект	software project	24, 26
проксі	proxy	468
просте присвоєння	simple assignment	46
процедурне програмування	procedural programming	15, 17–19
пряме адресування	direct addressing	51
режим компіляції робочої версії	release mode	78
Річі, Денніс	*Dennis MacAlistair Ritchie*	19, 630
розбухання коду	code bloating	495
розгалуження	if statement	72
роздільна компіляція	separate compilation	25
розширення типу	type promotion	50
Саттер, Герб	*Herb Sutter*	11, 630
селектор	selector	232
сигнатура функції	function signature	26, 110
символьний	char, wchar_t	33
символьний літерал	character literal	43
Сімпсона метод	Simpson's rule	201–204, 445
складене присвоєння	compound assignment	46
специфікатор сталої	qualifier const	45
стала відсилка	const reference	70
стала функція класу	constant member function	270
сталий указник	constant pointer	62
стан об’єкту	object state	213
статичний поліморфізм	static polymorphism	173
статичний член класу	static class member	319
сторонній ефект	side effect	38
Страуструп, Бьярне	*Bjarne Stroustrup*	11, 16, 21, 630
структура	structure	92
структурне програмування	structured programming	18

супровід	maintenance	549
термінальний клас	terminal class	620
тип даних	data type	239
ти́повий параметр	type parameter	482
тіло функції	function body	107
узагальнене програмування	generic programming	15, 186, 495
указник	pointer	51, 82, 95
указник константи	pointer to const	59
указник на поточний об’єкт	this pointer	228, 246, 262, 319
указник на функцію	pointer to function	198
указник спільного доступу	shareable pointer	460
унарна операція інкременту/декременту	increment/decrement operator	48
успадкування відкрите	public inheritance	552, 555
успадкування інтерфейсу	interface inharitance	594, 619
успадкування реалізації	inheritance of realization	594
утрата пам’яті	memory leak	98
файл заголовків	header file	25
файл реалізації	implementation file	25
Флойд, Роберт	*Robert W Floyd*	13, 630
фундаментальний тип	fundamental type	33
функтор	functor	446
функціональний об’єкт	functional object	445
функціональний специфікатор явного виклику	explicit specifier	298, 309
функція клонування, псевдоконструктор	clone, pseudo constructor	620
функція точності виведення	precision()	43
функція-член класу, метод	member function, method	239, 245, 267
цикл з лічильником	for statement	74
цикл із передумовою	while statement	75
цикл із постумовою	do-while statement	75
цілий	int, (un)signed int	33
часткова спеціалізація шаблону	partial template specialization	538
число Фібоначчі	Fibonacci number	74, 75, 104, 193–197
член класу	class member	210, 239
шаблон класу	class template	482, 497
шаблон функції	function template	186
шістнадцятковий літерал	hexadecimal, hex	42

Для замовлення книг звертайтесь за
тел. 050-648-05-00
ел. пошта **ihorza@gmail.com**
сайт **http://itknyga.com.ua**

Навчальне видання

Володимир Васильович Бублик
ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ

Підручник

Науковий редактор О.М. Левченко
Комп'ютерна верстка О.М. Левченко
Обкладинка І.О. Завадський

ФОП І.О. Завадський
Свідоцтво про внесення суб'єкта видавничої справи
до Державного реєстру видавців, виготівників
і розповсюджувачів видавничої продукції
серія КІ № 151 від 07.07.2011

Підписано до друку 12.01.15. Формат 60×84/16
Ум. друк. аркушів 40
Наклад 400 прим. Замовлення №

Віддруковано на ДП
«Вінницька картографічна фабрика»
21100, м. Вінниця, вул. 600-річчя, 19

Свідоцтво про внесення суб'єкта видавничої справи
до Державного реєстру видавців, виготівників
і розповсюджувачів видавничої продукції
ДК № 869 від 26.03.2002 р.

«Інфопульс Україна»

«Інфопульс Україна» входить до складу найбільшої норвезької ІТ-групи EVRY та надає сервіси для всього життєвого циклу ІТ-систем і підтримки телеком-інфраструктури для провідних організацій світу. Заснований більш ніж 20 років тому, зараз «Інфопульс» є надійним партнером для більш ніж 1000 спеціалістів, які працюють у різних регіонах України, а також в європейських офісах компанії.

Основні напрями діяльності:

- проектування, розробка та тестування програмного забезпечення;
- управління ІТ-інфраструктурою;
- автоматизація розгортання програмного забезпечення та віртуалізація;
- аутсорсинг бізнес-процесів.

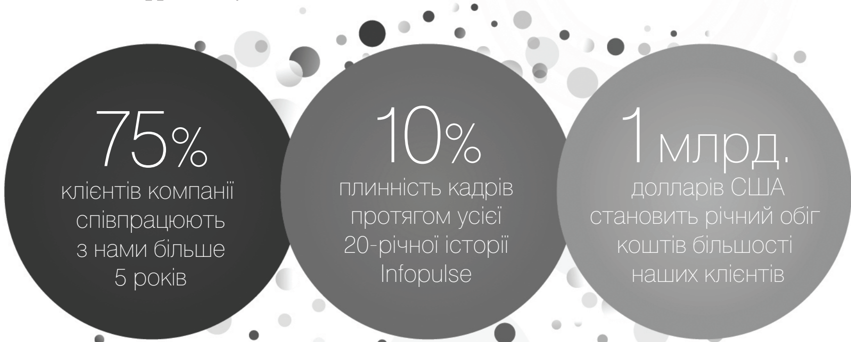
Робота в «Інфопульсі» – це:

Розвиток

Різнопланове сервіс-портфоліо та, відповідно, розмаїття проектів дозволяють нам не тільки стабільно зростати, але й надавати широкі можливості для розвитку наших спеціалістів.

Стабільність

Наш фокус – довгострокова співпраця з нашими замовниками та спеціалістами. Цифри кажуть самі за себе:



Приємна атмосфера

Ми віримо, що успіх усієї компанії напряму залежить від кожного з її спеціалістів, розуміємо важливість балансу між роботою та особистим життям, підтримуємо ініціативи, інвестуємо у розвиток талантів та даємо можливість обирати нові шляхи професійного розвитку в компанії.

Грамотний проектний менеджмент і доброзичливі відносини в колективі — це те, чому ми завжди приділяли і будемо приділяти особливу увагу.

Ми пишаємося нашою командою професіоналів та запрошуємо вас стати її частиною!

Надсилайте ваші резюме на адресу:
apply@infopulse.com.ua
Дізнайтеся більше на www.infopulse.com

EVRY
WE ARE PART OF EVRY GROUP