Міністерство освіти і науки України
ОДЕСЬКИЙ НАЦІОНАЛЬНИЙ ПОЛІТЕХНІЧНИЙ УНІВЕРСИТЕТ

КОНСПЕКТ ЛЕКЦІЙ
з дисципліни «Програмування»
(частина 2)

Одеса;ОНПУ, -2015

Міністерство освіти і науки України
ОДЕСЬКИЙ НАЦІОНАЛЬНИЙ ПОЛІТЕХНІЧНИЙ УНІВЕРСИТЕТ

КОНСПЕКТ ЛЕКЦІЙ
з дисципліни «Програмування»
(частина 2)

Освітньо-кваліфікаційний рівень – бакалавр
Напрям – 6.170103 – Управління інформаційною безпекою

Затверджено на засіданні
кафедри інформатики та управління
захистом інформаційних систем
Протокол № від 28.08.2015 р.

Одеса;ОНПУ, -2015

Конспект лекцій з дисципліни «Програмування» (частина 2) для студентів **напряму 6.170103 – Управління інформаційною безпекою** /Укл.: К.О. Трифонова - Одеса: ОНПУ, 2018. — 88 с.

Укладач:
К.О. Трифонова ст..викл

ПЕРЕДМОВА

Метою вивчення дисципліни є формування фундаментальних понять і методів інформатики: поняття алгоритму, алгоритмічної конструкції, комп'ютерної програми, мови програмування, методології і технології програмування.

Для досягнення мети вивчення дисципліни потрібно виконати наступні задачі: студенти повинні навчитися вирішувати задачі аналізу, проектування (розробка алгоритму), кодування і компіляції (написання вихідного тексту програми та перетворення його у виконуваний код за допомогою компілятора), тестування та налагодження, супроводження.

Дисципліна «Програмування» базується на знаннях та вміннях шкільного курсу «Інформатика» та вивчається одночасно з дисциплінами «Математичний аналіз», «Алгебра та геометрія», «Основи КТ», а її основні положення використовуються при вивченні дисципліни "Захист інформації", «1С програмування», а також під час дипломного проектування.

Дисципліна «Програмування» є однією з основоположних у системі підготовки бакалаврів за напрямком 6.040302. Вона закладає основу алгоритмічного мислення студентів та реалізація його за допомогою С-подібних мов програмування .

Дисципліна має націлити майбутніх фахівців на творче застосування отриманих знань у їх практичній діяльності.

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

Об'єктно-орієнтоване програмування – одна з парадигм програмування, яка розглядає програму як множину "об'єктів", що взаємодіють між собою. В ній використано декілька технологій від попередніх парадигм, зокрема успадкування, модульність, поліморфізм та інкапсуляцію. Незважаючи на те, що ця парадигма з'явилась ще в 1960-тих роках, вона не мала широкого застосування до 1990-тих. Сьогодні багато мов програмування (зокрема, Java, C#, C++, Python, PHP, Ruby та Objective-C, ActionScript 3) підтримують ООП. Об'єктно-орієнтоване програмування сягає своїм корінням до створення мови програмування Симула в 1960-тих роках, одночасно з посиленням дискусій про кризу програмного забезпечення. Разом із тим, як ускладнювалось апаратне та програмне забезпечення, було дуже важко зберегти якість програм. Об'єктно-орієнтоване програмування частково розв'язало цю проблему шляхом наголошення на модульності програми. На відміну від традиційних поглядів, коли програму розглядали як набір підпрограм, або як перелік насвідно до парадигми об'єктно-орієнтованого програмування, кожний об'єкт здатний отримувати повідомлення, обробляти дані, та надсилати повідомлення іншим об'єктам. Кожен об'єкт – своєрідний незалежний автомат з окремим призначенням та відповідальністю.

У цьому навчальному посібнику огляд особливостей об'єктно-орієнтованого програмування починається з поняття "об'єкт". Загалом об'єкт – це абстрактна суть, наділена характеристиками об'єктів, що оточує нас в реальному світі. Створення об'єктів і маніпулювання ними – це зовсім не привілей мови програмування C++, а швидше результат технології програмування, що утілює в кодових конструкціях опису об'єктів і операції над ними. Кожен об'єкт програми, як і будь-який реальний об'єкт, відрізняється власними атрибутами і характерною поведінкою.

Об'єкти можна класифікувати за різними категоріями: наприклад, цифровий наручний годинник "Orient" належить до класу годинника. Програмна реалізація годинника входить як стандартний додаток до складу операційної системи будь-якого комп'ютера.

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

ЗМІСТ

РОЗДІЛ № 1. ОСНОВНІ ОСОБЛИВОСТІ РОЗРОБЛЕННЯ ОБ'ЄКТНО-ОРІЄНТОВАНИХ ПРОГРАМ МОВОЮ С++

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

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

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

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

1.1 Потреба використання об'єктно-орієнтованого програмування

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

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

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

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

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

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

Проаналізуємо ці недоліки на прикладі розроблення програми ведення складського обліку, наприклад, матеріальних цінностей. У такій програмі глобальними даними є записи в обліковій книзі. Різні функції отримуватимуть доступ до цих даних для виконання різних операцій: створення нового запису, виведення запису на екран, зміни полів наявного запису і т.д. Неконтрольований доступ до даних. У структурній програмі, написаній, наприклад, мовою Pascal, існує два типи даних. Локальні дані знаходяться усередині будь-якої функції та призначені для використання тільки нею.

Наприклад, у програмі ведення складського обліку функція, яка здійснює виведення запису на екран, може використовувати локальні дані для зберігання вартості деяких матеріалів, якщо відома їх закупівельна ціна та наявна кількість. Локальні дані функції недоступні нікому, окрім неї самої, і не можуть бути змінені іншими функціями.

Якщо існує потреба сумісного використання одних і тих самих даних декількома функціями, то ці дані мають бути оголошені як глобальні. Це, як правило, стосується тих даних програми, які є найважливішими. Прикладом тут може слугувати вже згадана вище ціна та кількість матеріалу. Будь-яка функція має доступ до глобальних даних (ми не розглядаємо випадок групування функцій у програмні модулі). Схему, яка ілюструє концепцію області видимості локальних і глобальних даних, наведено на рис. 1.1.

Рис. 1.1. Концепція області видимості глобальних і локальних даних

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

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

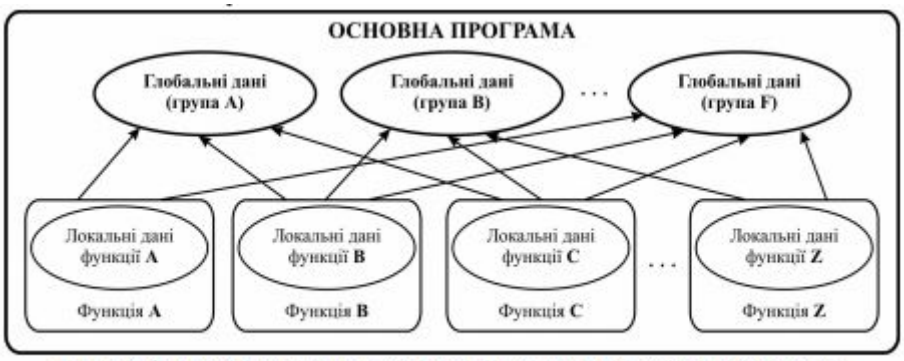

Рис. 1.2. Структурний підхід до встановлення зв'язків між глобальними даними і функціями програми

Коли зміни вносяться в глобальні дані великих програм, то не завжди можна швидко визначити, які функції при цьому необхідно скоректувати. Навіть тоді, коли це вдається зробити оперативно, то згодом через значну кількість зв'язків між функціями і даними виправлені функції починають некоректно працювати з іншими глобальними даними. Таким чином, будь-яка зміна у коді програми призводить до виникнення негативних подальших дій і, як наслідок, появи відповідних проблем.

Відображення картини реального світу. Другий, набагато важливіший недолік структурного програмування полягає в тому, що відокремлення даних від функцій виявляється малопридатним для достовірного "відображення картини реального світу", тобто, адекватного відтворення фізичного змісту технічного завдання. Йдеться про те, що у реальному світі нам доводиться мати справу з фізичними об'єктами, такими, наприклад, як люди або машини.

Ці об'єкти не можна віднести ні до даних, ні до функцій, оскільки реальні речі характеризує сукупність певних властивостей чи їх поведінку.

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

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

Таким чином, ні окремо взяті глобальні дані, ні відокремлені від них функції не здатні адекватно відображати фізичні об'єкти реального світу. 1.2. Поняття про об'єктно-орієнтований підхід до розроблення складних програм

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

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

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

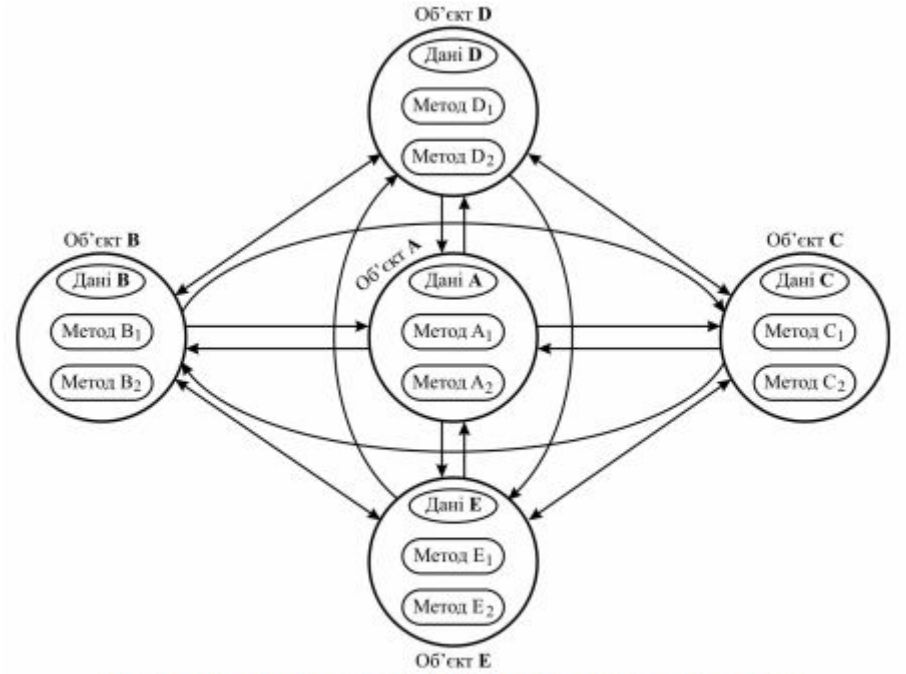

Рис. 1.3. Об'єктно-орієнтований підхід до встановлення зв'язків між даними і методами (функціями)

Якщо необхідно змінити значення даних об'єкта, то, очевидно, ця дія також має бути покладена на відповідну функцію об'єкта. Ніякі інші функції не можуть змінювати значення даних об'єкта, тобто дані класу. Такий підхід полегшує написання, відлагодження та використання програми. Таким чином, типова програма, написана мовою C++, складається з сукупності об'єктів, що взаємодіють між собою за допомогою методів, які викликають один одного.

Об'єктно-орієнтований підхід до встановлення зв'язків між даними і функціями (методами) класу, написаної мовою C++, наведено на рис. 1.3. Виробнича аналогія. Для кращого розуміння призначення об'єктів, уявімо собі деяку промислову фірму, яка складається з бухгалтерії, відділу кадрів, відділу реалізації продукції, відділів головного технолога та головного механіка (рис. 1.4) і т.д. Поділ фірми на відділи є важливою частиною структурної організації її виробничої діяльності. Для більшості фірм (за винятком невеликих) в обов'язки окремого співробітника не входить вирішення одночасно кадрових, виробничих і обліково-бухгалтерських питань. Різні обов'язки чітко розподіляються між підрозділами, тобто у кожного підрозділу є дані, з якими він працює: у бухгалтерії – заробітна плата, у відділі реалізації продукції – інформація, яка стосуються торгівлі, у відділі кадрів – персональна інформація про співробітників, у відділі головного технолога – стан технологічного процесу виготовлення продукції, у відділі головного механіка – технічний стан обладнання та устаткування, транспортних засобів і т.д.

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

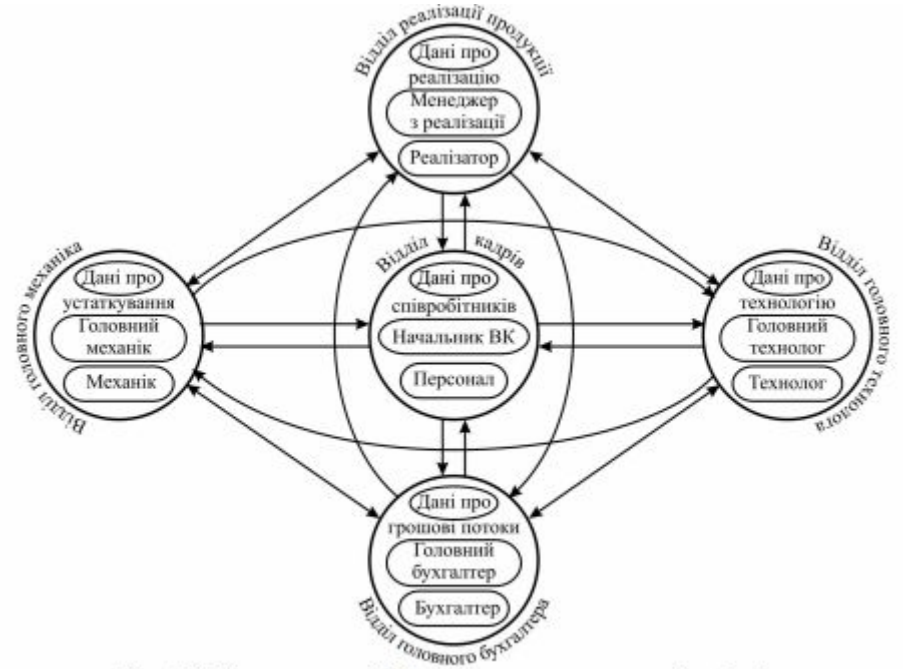

Рис. 1.4. Корпоративний підхід до встановлення зв'язків між відділами і їх виробничими функціями

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

ООП – ефективний спосіб організації програми. ООП ніяк не пов'язане з процесом виконання програми, а є тільки способом її ефективної організації. Велика частина операторів мови C++ є аналогічною операторам процедурних мов програмування, зокрема мови C. Зовні функція класу у мові програмування C++ дуже схожа на звичайну функцію мови C, і тільки за контекстом програми можна визначити, чи є функція частиною структурної C-програми або об'єктно-орієнтованої програми, написаної мовою C++.

1.3 Основні компоненти об'єктно-орієнтованої мови програмування

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

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

Тут ми дамо відповідь тільки на таке запитання: що у коді програми треба представляти у вигляді об'єктів? Остаточну відповідь на це запитання може дати тільки Ваш власний досвід програмування з використанням об'єктно-орієнтованого підходу, а також Ваше досконале

знання фізичної сутності розв'язуваної задачі. Проте нижче наведено декілька прикладів, які можуть виявитися корисними як для початківців-студентів, так і для програмістів-не-фахівців з ООП:

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

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

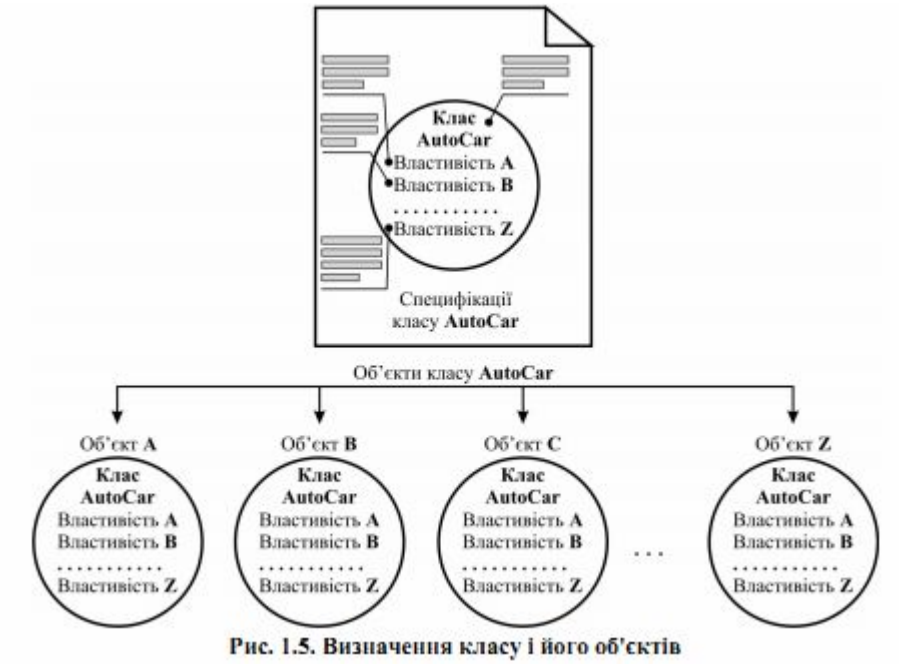

Рис. 1.5. Визначення класу і його об'єктів

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

int day, count, divisor, answer;

За аналогією ми можемо також визначати об'єкти класу, як це показано на рис. 1.5. Тобто, клас – це тип форми, що визначає, які дані та функції будуть включені в об'єкт. Під час

оголошення класу не створюються ніякі об'єкти цього класу, за аналогією з тим, що існування типу int ще не означає наявність змінних цього типу.

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

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

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

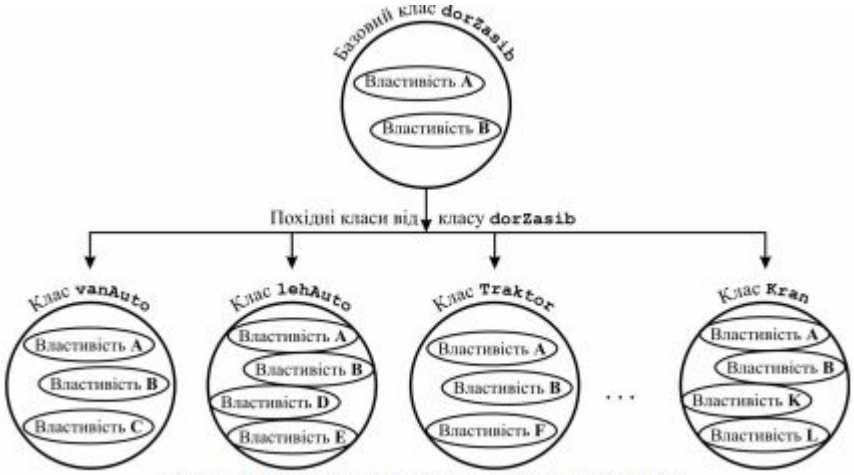

Рис. 1.6. Ілюстрація ідеї успадкування в класах

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

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

Успадкування можна вважати аналогом використання функцій в структурному підході. Якщо ми виявимо декілька функцій, які здійснюють схожі дії, то вилучимо з них однакові частини і винесемо їх у окрему функцію. Тоді початкові функції будуть однаковою мірою викликати свою загальну частину, і водночас у кожній з них міститимуться свої власні настанови. Базовий клас містить елементи, які є загальними для групи похідних класів. Значення успадкування в ООП таке ж саме, як і функцій у структурному програмуванні, – скоротити розмір коду програми і спростити зв'язки між її елементами.

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

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

Припустимо, що Ви розробили клас, який представляє систему меню, аналогічну графічному інтерфейсу Microsoft Windows або іншому графічному інтерфейсу користувача (GUI). Ви не хочете змінювати цей клас, але Вам необхідно мати можливість встановлювати та знімати опції. У цьому випадку

Ви створюєте новий клас, який успадковує всі властивості початкового класу, і додаєте до нього необхідний код програми. Зручність повторного використання кодів уже написаних програм є важливою перевагою технології ООП над іншими технологіями. Багато комп'ютерних компаній стверджують, що можливість вносити в нові версії програмного забезпечення старі коди програм сприяє зростанню їх прибутків, які вони приносять. Детальніше це питання розглядатиметься в інших розділах цього навчального посібника.

Поняття про типи даних користувача. Одним з достоїнств об'єктів є те, що вони дають змогу програмісту створювати свої власні типи даних. Уявіть собі, що Вам необхідно працювати з об'єктами, які мають дві координата, наприклад x і y. Вам хотілося б здійснювати звичайні арифметичні операції над такими об'єктами, наприклад: Ob1 = Ob2 + obj; де змінні Ob1, Ob2 і obj є наборами з двох координат. Описавши клас, який містить пару координат, і оголосивши об'єкти цього класу з іменами Ob1, Ob2 і obj, ми фактично створимо новий тип даних. У мові програмування C++ є засоби, які полегшують створення подібних типів даних користувача.

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

Відповідь на це запитання полягає в тому, що ми самі можемо задати ці дії, зробивши потрібні оператори методами класу Coordinate.

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

1.4 Співвідношення між мовами програмування С і С++

Мова програмування C++ успадкувала можливості мови C. Строго кажучи, мова C++ є розширенням мови C: будь-яка конструкція, написана мовою C, є коректною для мови C++; водночас зворотне твердження – є хибним.

Найбільш значні нововведення, які присутні у мові C++, стосуються класів, об'єктів і ООП (первинна назва мови C++ – "C з класами"). Проте є і інші удосконалення, пов'язані із способами організації введення/виведення і напи-сання коментарів. Ілюстрацію співвідношення між мовами C і C++ наведено на рис. 1.7.

На практиці існує значно більше відмінностей між мови C і C++, ніж може видатися спочатку. Незважаючи на те, що мовою C++ можна написати коди програм такі ж самі як і мовою C, однак навряд чи комусь прийде в голову це так робити. Програмісти, які застосовують мову C++, не тільки користуються перевагами цієї мови перед мовою C, але і по-новому використовують її можливості, частина з яких успадкована від мови C. Якщо Ви

знайомі з мовою С, то це означає, що у Вас вже є деякі знання відносно мови програмування С++, але ймовірніше за все, що значна частина матеріалу виявиться для Вас новою.

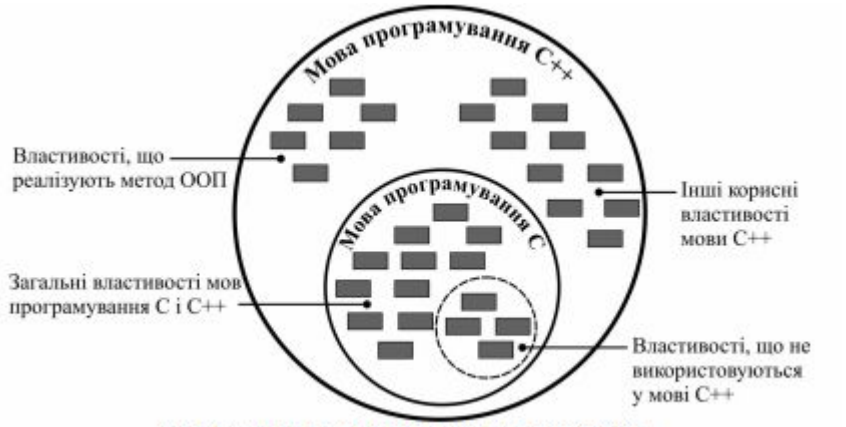

Рис. 1.7. Співвідношення між мовами С і С++

Завдання цього навчального посібника полягає в тому, щоб якнайшвидше навчити Вас створювати об'єктно-орієнтовані програми мовою програмування С++. Оскільки, як було вже сказано раніше, значна частина можливос-тей мови С++ успадкована від її попередниці – мови С, то навіть при об'єктно-орієнтованій структурі програми її основу становлять "старомодні" процедурні засоби.

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

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

1.5 Поняття про універсальну мову моделювання

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

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

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

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

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

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

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

Контрольні запитання

1. Назвіть основну необхідність використання об'єктно-орієнтованого програмування.
2. Які поняття відносяться до об'єктно-орієнтованого підходу до розроблення складних програм?
3. Назвіть основні компоненти об'єктно-орієнтованої мови програмування.
4. Які існують співвідношення між мовами програмування C і C++?
5. Основне призначення універсальної мови моделювання?

РОЗДІЛ № 2. ОСНОВА ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ

1. Базові поняття класу
2. Поняття про конструктори і деструктори
3. Особливості реалізації механізму доступу до членів класу
4. Класи і структури – споріднені типи
5. Об'єднання та класи – споріднені типи
6. Поняття про вбудовані функції
7. Особливості організації масивів об'єктів
8. Особливості використання покажчиків на об'єкти

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

2.1 Базові поняття класу

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

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

даних класу використовують термін змінна класу, а замість терміна функція-член класу використовують термін метод класу.

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

Варіант 1

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

Варіант 2

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

У цих оголошеннях елемент ім'я_класу означає ім'я "класового" типу. Воно стає іменем нового типу, яке можна використовувати для побудови об'єктів цього класу. Об'єкти класу інколи створюють шляхом вказання їх імен безпосередньо за закритою фігурною дужкою оголошеного класу як елемент перелік_об'єктів_класу (варіанти 1). Проте найчастіше об'єкти створюють в міру потреби після оголошення класу (варіант 2).

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

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

Розглянемо детально механізм визначення цього класу. Усі члени класу myClass оголошені в тілі настанови class. Членом даних класу myClass є змінна a. Окрім цього, тут визначено три функції-члени класу: Init() – ініціалізація об'єта, Get() – введення в об'єкт значення і Put() – виведення з об'єкта значення. Будь-який клас може містити як закриті, так і відкриті члени. За замовчуванням усі члени, визначені в класі, є закритими (private-членами). Наприклад, змінна a є закритим членом даних класу. Це означає, що до неї можуть отримати доступ тільки функції-члени класу myClass; ніякі інші частини програми цього зробити не можуть. У цьому полягає один з проявів інкапсуляції: програміст повною мірою може керувати доступом до певних елементів даних. Закритими можна оголосити і функції (у наведеному вище прикладі таких немає), внаслідок чого їх зможуть викликати тільки інші члени цього класу.

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

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

ObjB типу myClass:

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

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

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

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

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

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

За своєю сутністю такий запис повідомляє компілятор про те, що дана версія функції Get() належить класу myClass. Іншими словами, оператор дозволу "::" заявляє компілятору, що функція Get() знаходиться у області видимості класу myClass.

Різні класи можуть використовувати однакові імена функцій. Однак, компілятор самостійно визначить, до якого класу належить та чи інша функція за допомогою імені класу та оператора дозволу області видимості.

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

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

Під час виклику функції ObjA.Init() дії, визначені у функції Init(), будуть спрямовані на копії даних, які належать тільки об'єкту ObjA. Необхідно мати на увазі, що ObjA і ObjB – це два окремі об'єкти. Це означає, що ініціалізація змінних об'єкта ObjA зовсім не приводить до ініціалізації змінних об'єкта ObjB. Єдине, що зв'язує об'єкти ObjA і ObjB, це те, що вони мають один і той самий класовий тип і функції-члени класу, які обробляють його дані.

Якщо одна функція-член класу викликає іншу функцію-члена того ж самого класу, то не потрібно вказувати ім'я об'єкта і використовувати оператор "крапка".

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

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

Код програми 2.1. Демонстрація механізму оголошення класу та його застосування

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

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

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

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

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

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

}

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

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

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

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

getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:
Вміст об'єкта ObjA: 4.06663
Вміст об'єкта ObjB: 20.0294
Закриті члени класу доступні тільки функціям, які є членами цього класу.

Наприклад, таку настанову

ObjA.a = 0;

не можна помістити у функцію main() нашої програми.

2.2 Поняття про конструктори і деструктори

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

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

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

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

Тепер наведемо код функції myClass():

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

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

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

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

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

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

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

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

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

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

Код програми 2.2. Демонстрація механізму використання конструктора і деструктора

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

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

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

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

// Введення в об'єкт значення.
```

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

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

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

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

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

getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

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

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

```
class myClass { // Оголошення класового типу
double a;
int nom;  // Містить ідентифікаційний номер об'єкта
public:
myClass(int n); // Оголошення параметризованого конструктора


~myClass(); // Оголошення деструктора


void Get(double, double); // Введення в об'єкт значення
double Put(); // Виведення з об'єкта значення
};
```

Член-даних nom використовують для зберігання ідентифікаційного номера об'єкта, який створюється основною функцією. Його реальне значення визначається значенням, яке передається конструктору як формальний параметр n при створенні змінної типу myClass. Параметризований конструктор myClass() буде мати такий вигляд:

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

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

Варіант 1 Варіант 2 (альтернативний)

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

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

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

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

тип_класу ім'я_об'єкта(перелік_аргументів);

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

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

Код програми 2.3. Демонстрація механізму використання параметризованого конструктора

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

class myClass { // Оголошення класового типу
double a;
int nom;  // Містить ідентифікаційний номер об'єкта
public:
myClass(int n); // Оголошення параметризованого конструктора


~myClass(); // Оголошення деструктора


void Get(double, double); // Введення в об'єкт значення
double Put(); // Виведення з об'єкта значення
};

// Визначення параметризованого конструктора.
myClass::myClass(int n)
```

```
{
a = 0; nom = n;
cout << "Об'єкт " << nom << " ініціалізовано" << endl;
}

// Визначення деструктора.


myClass::~myClass()


{
cout << "Об'єкт " << nom << " зруйновано" << endl;
}

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

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

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

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

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

getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

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

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

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

Код програми 2.4. Демонстрація механізму передачі конструктору двох параметрів

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

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

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

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

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

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

getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

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

2.3 Особливості реалізації механізму доступу до членів класу

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

Код програми 2.5. Демонстрація механізму доступу до членів класу

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

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

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

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

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

void myClass::Fun()  // Надання значень членам-даних
{
Set(0);  // Прямий виклик функції Set() створеним об'єктом
}

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

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

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

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

getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:
Об'єкт Obj після виклику функції Set(5): 5 25
Об'єкт Obj після встановлення члена Obj.b=20: 5 20
Об'єкт Obj після виклику функції Obj.Fun(): 0 0

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

a = c; // Пряме звернення до змінної a

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

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

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

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

Оскільки ця настанова не належить тілу класу myClass, то до змінної b можливий доступ тільки з використанням імені конкретного об'єкта (у цьому випадку об'єкта Obj) і оператора "крапка". Тепер зверніть увагу на те, як викликається функція-член класу Fun() з основної функції main():

Obj.Fun();

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

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

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

2.4 Класи і структури – споріднені типи

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

Насправді відповідно до формального синтаксису мови програмування C++ оголошення структури створює тип класу. Розглянемо приклад використання структури з властивостями, подібними до властивостей класу.

Код програми 2.6. Демонстрація механізму використання структури для створення класу

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

struct myStruct { // Оголошення структурного типу
int Put();  // Ці члени відкриті (public)
void Get(int d); // за замовчуванням.
```

```
private:
int c;
};

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

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

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

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

getch(); return 0;
}
```

У цьому коді програми визначається тип структури з іменем myStruct, у якій функції-члени Put() і Get() є відкритими (public), а член даних c – закритим (private). Зверніть увагу на те, що у структурах для оголошення закритих членів використовують ключове слово private. У наведеному нижче прикладі показано еквівалентну програму, яка використовує замість типу struct тип class.

Код програми 2.7. Демонстрація механізму використання класу замість структури

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

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

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

int main()
{
```

```
myClass Obj; // Створення об'єкта класу

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

getch(); return 0;
}
```

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

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

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

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

2.5 Об'єднання та класи – споріднені типи

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

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

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

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

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

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

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

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

getch(); return 0;
}
```

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

2.6 Поняття про вбудовані функції

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

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

Модифікатор inline повинен передувати усім решта настанов оголошення самої функції. Причиною існування вбудованих функцій є ефективність їх використання. Адже під час кожного виклику звичайної функції виконається деяка послідовність настанов, пов'язаних з обробленням самого виклику, що містить занесення її аргументів у стек, чи поверненням їх з

функції. В деяких випадках значна кількість циклів центрального процесора використовується саме для виконання цих дій.

Але, якщо функція вбудовується в рядок програми, то таких системних витрат просто немає, і загальна швидкість виконання програми зростає. Якщо ж вбудована функція виявляється великою, то загальний розмір програми може істотно збільшитися. Тому краще як вбудовані використовувати тільки маленькі функції, а ті, що є більшими, оформляти у вигляді звичайних функцій. Продемонструємо механізм використання вбудованої функції на прикладі такої програми.

Код програми 2.9. Демонстрація механізму використання вбудованої функції

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

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

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

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

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

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

getch(); return 0;
}
```

У цьому коді програми замість виклику функцій Put() і Get() підставляють їх код. Так, у функції main() рядок Obj.Get(10); функціонально еквівалентний такій настанові присвоєння: Obj.c = 10;

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

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

деякі компілятори не генерують вбудованого коду, якщо відповідна функція містить цикл, конструкцію switch або настанову goto;

найчастіше вбудованими не можуть бути рекурсивні функції;

як правило, механізм вбудовування "не проходить" для функцій, які містять статичні (static) змінні.

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

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

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

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

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

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

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

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

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

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

```
};
```

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

2.7 Особливості організації масивів об'єктів

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

Код програми 2.11. Демонстрація механізму організації масиву об'єктів

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

enum resolution {low, medium, high};

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

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

int main()

{
displayClass Monitor[3];

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

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

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

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

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

```
cout << w << " x " << h << endl;
}

getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

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

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

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

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

Код програми 2.12. Демонстрація механізму ініціалізації масиву об'єктів

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

class myClass { // Оголошення класового типу
int a;
public:
myClass(int b)  { a = b; }
double Put()  { return a; }
};

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

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

Результати виконання цієї програми

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

підтверджують, що конструктору myClass дійсно були передані значення від -1 до -4. Насправді синтаксис ініціалізації масиву, виражений рядком myClass array[4] = { -1, -2, -3, -4 };, є скороченим варіантом такого (довшого) формату: myClass array[4] = { myClass(-1), myClass(-2), myClass(-3), myClass(-4)}; Формат ініціалізації, представлений у наведеній вище програмі, використовується програмістами частіше, ніж його довший еквівалент, проте

необхідно пам'ятати, що він працює для масивів таких об'єктів, конструктор яких приймає тільки один аргумент. При ініціалізації масиву об'єктів, конструктор яких приймає декілька аргументів, необхідно використовувати довший формат ініціалізації. Розглянемо такий приклад.

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

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

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

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

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

Внаслідок виконання ця програма відображає на екрані такі результати:

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

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

2.8 Особливості використання покажчиків на об'єкти

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

getch(); return 0;
}
```

Ось як виглядають результати виконання цієї програми:

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

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

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

Контрольні запитання

1. Назвіть базові поняття класу.
2. Назвіть призначення конструкторів і деструкторів.
3. В чому особливості реалізації механізму доступу до членів класу?
4. В чому основна відмінність між класами і структурами?
5. В чому основна відмінність між об'єднанням та класами?
6. Основне призначення вбудованих функцій?
7. Як організовані масиви об'єктів?
8. Які існують особливості використання покажчиків на об'єкти?

РОЗДІЛ № 3. ОРГАНІЗАЦІЯ КЛАСІВ І ОСОБЛИВОСТЕЙ РОБОТИ З ОБ’ЄКТАМИ

1. Поняття про функції-"друзі" класу
2. Особливості перевизначення конструкторів
3. Особливості механізму динамічної ініціалізації конструктора
4. Особливості механізму присвоєння об'єктів

5. Особливості механізму передачі об'єктів функціям
6. Конструктори, деструктори і передача об'єктів
7. Потенційні проблеми, які виникають при передачі об'єктів
8. Особливості механізму повернення об'єктів функціями
9. Механізми створення та використання конструктора копії
10. Використання конструктора копії для ініціалізації одного об'єкта іншим
11. Механізм використання конструктора копії для передачі об'єкта функції
12. Механізм використання конструктора копії при поверненні функцією об'єкта
13. Конструктори копії та їх альтернативи
14. Поняття про ключове слово this

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

3.1 Поняття про функції-"друзі" класу

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

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

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

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

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

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

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

```
return obj.a + obj.b;
}

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

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

getch(); return 0;
}
```

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

Незважаючи на те, що у наведеному вище прикладі не отримуємо ніякої користі з оголошення "другом" функції Sum(), а не членом класу myClass, існують певні обставини, при яких статус "дружньої" функції класу має велике значення. По-перше, функції-"друзі" є корисними для перевизначення операторів певних типів. По-друге, функції-"друзі" спрощують створення деяких функцій введення-виве-дення. Усі ці питання розглядатимемо згодом у цьому навчальному посібнику. Третя причина частого використання функцій-"друзів" полягає у тому, що в деяких випадках два (або більше) класи можуть містити члени, які перебувають у взаємному зв'язку з іншими частинами програми. Наприклад, у нас є два різні класи, які під час виникнення певних подій відображають на екрані "спливаючі" повідомлення. Інші частини програми, які призначені для виведення даних на екран, повинні знати, чи є "спливаюче" повідомлення активним, щоб випадково не перезаписати його. Для уникнення цього у кожному класі можна створити функцію-члена, що повертає значення, за якою робляться висновки про те, є повідомлення активним чи ні. Однак перевірка цієї умови вимагатиме додаткових витрат (тобто двох викликів функцій замість одного). Якщо статус "спливаючого" повідомлення необхідно перевіряти часто, то ці додаткові витрати можуть виявитися відчутними. Проте за допомогою функції, "дружньої" для обох класів, можна безпосередньо перевіряти статус кожного об'єкта, викликаючи для цього тільки одну і ту саму функцію, яка матиме доступ до обох класів. У таких ситуаціях "дружня" функція класу дає змогу написати більш ефективний програмний код. Реалізацію зазначеної ідеї продемонструємо на прикладі такої програми.

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

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

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

class bClass;  // Випереджувальне оголошення класу

class aClass {  // Оголошення класового типу
int status;  // IDLE, якщо повідомлення неактивне
```

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

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

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

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

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

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

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

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

getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

Екран вільний.

Відображається повідомлення.

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

Зверніть увагу на те, що у цьому коді програми використовується випереджувальне оголошення (часто його також називають випереджувальним посиланням) для класу bClass. Потреба у ньому пов'язана з тим, що оголошення функції

Put() у класі aClass використовує посилання на клас bClass ще до його оголошення. Щоб створити випереджувальне оголошення для будь-якого класу, достатньо зробити це так, як

показано у наведеному вище коді програми. "Дружня" функція одного класу може бути членом іншого класу. Переробимо наведену вище програму так, щоби функція Put() стала членом класу aClass. Зверніть увагу на використання оператора дозволу області видимості (або оператора дозволу контексту) під час оголошення функції Put() як "друга" класу bClass. Код програми 3.3. Демонстрація механізму використання функції – члена одного класу і одночасно "дружньої функції" – для іншого класу

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

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

class bClass;  // Випереджувальне оголошення класу

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

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

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

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

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

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

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

getch(); return 0;
}
```

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

3.2 Особливості перевизначення конструкторів

Незважаючи на те, що конструктори призначені для виконання унікальних дій, проте вони не надто відрізняються від функцій-членів класу інших типів і також можуть піддаватися перевизначенню. Щоб перевизначати конструктор класу, достатньо оголосити його в усіх потрібних форматах і визначити кожну дію, пов'язану з кожним форматом. Наприклад, у наведеному нижче коді програми оголошено клас timerClass, який діє як таймер зворотного відліку. При створенні об'єкта типу timerClass таймеру присвоюється деяке початкове значення часу. При виклику функції Run() таймер здійснює відлік часу у зворотному порядку до нуля, а потім подає звуковий сигнал. У наведеному нижче коді програми конструктор перевизначається тричі, надаючи тим самим можливість задавати час як у секундах (причому або числом, або рядком), так і у хвилинах і секундах (за допомогою двох цілочисельних значень). У цьому коді програми використовується стандартна бібліотечна функція clock(), яка повертає кількість сигналів, прийнятих від системного годинника з моменту початку виконання програми. Прототип цієї функції має такий вигляд:

```
clock_t clock();
```

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

Код програми 3.4. Демонстрація механізму використання перевизначених конструкторів

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

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

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

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

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

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

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

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

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

getch(); return 0;
}
```

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

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

3.3 Особливості механізму динамічної ініціалізації конструктора

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

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

Подібно до простих змінних, об'єкти можна ініціалізувати динамічно під час їх створення. Цей механізм дає змогу створювати об'єкт потрібного типу з використанням інформації, яка стає відомою тільки у процесі виконання програми.

Щоб показати, як працює механізм динамічної ініціалізації конструктора, спробуємо модифікувати код програми реалізації таймера, який було наведено в попередньому розділі. Згадаймо, що в першому прикладі коду програми таймера ми не отримали великої переваги від перевизначення конструктора timerClass(), оскільки всі об'єкти цього типу

ініціалізувалися за допомогою констант, відомих при компілюванні програми. Але у випадках, коли об'єкт необхідно ініціалізувати у процесі виконання програми, можна отримати істотний виграш від наявності множини різних форматів ініціалізації. Це дає змогу програмісту вибрати з наявних конструкторів той, який найточніше відповідає поточному формату даних. Наприклад, у версії коду програми таймера для створення двох об'єктів ObjB і ObjC, яку наведено нижче, використовується динамічна ініціалізація конструктора.

Код програми 3.5. Демонстрація механізму динамічної ініціалізації конструктора

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

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

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

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

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

void Run(); // Таймер відліку часу
};

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

int main()
{
timerClass ObjA(10); // Створення об'єкта класу
ObjA.Run();
char str[80];

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

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

```
ObjC.Run();

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

getch(); return 0;
}
```

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

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

Захаращення коду програми конструкторами для оброблення ситуацій, які рідко виникають, як правило, дестабілізаційно впливає на читабельність коду програми.

3.4 Особливості механізму присвоєння об'єктів

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

Код програми 3.6. Демонстрація механізму присвоєння об'єктів

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

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

int main()
```

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

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

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

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

getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Об'єкт ObjA до присвоєння:
a = 10; b = 20
Об'єкт ObjB до присвоєння:
a = 0; b = 0

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

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

3.5 Особливості механізму передачі об'єктів функціям

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

Код програми 3.7. Демонстрація механізму передачі об'єктів функціям

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

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

```
};

void Fun(myClass obj) // Визначення функції не члена класу
{
obj.Show("t2= "); // Виведення числа 10.
obj.Set(100); // Встановлює тільки локальну копію.
obj.Show("t3= "); // Виведення числа 100.
}
int main()
{
myClass Obj;  // Створення об'єкта класу

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

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

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

getch(); return 0;
}
```

Ось як виглядають результати виконання цієї програми.

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

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

3.6 Конструктори, деструктори і передача об'єктів

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

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

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

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

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

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

Get(ObjA);

getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

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

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

При передачі об'єкта функції створюється його копія (і ця копія стає параметром у функції). Створення копії означає появу нового об'єкта. Коли виконання функції завершується, копія аргумента (тобто параметр) руйнується. Тут виникає відразу два запитання. По-перше, чи викликається конструктор об'єкта при створенні копії? По-друге, чи викликається деструктор об'єкта під час руйнування копії? Відповіді на ці запитання можуть здивувати Вас. Оскільки під час виклику функції створюється копія аргумента, то звичайний конструктор об'єкта не викликається. Натомість викликається конструктор копії об'єкта, який визначає, як має створюватися копія об'єкта. Але, якщо в класі безпосередньо не визначено конструктор копії, то мова програмування C++ надає його за замовчуванням. Конструктор копії за замовчуванням створює побітову (тобто однакову) копію об'єкта. Оскільки звичайний конструктор використовують для ініціалізації тільки деяких даних об'єкта, то він не повинен викликатися для створення копії вже наявного об'єкта. Такий виклик змінив би його вміст. При передачі об'єкта функції потрібно використовувати поточний стан об'єкта, а не його початковий стан.

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

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

3.7 Потенційні проблеми, які виникають при передачі об'єктів

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

Описана ситуація робить оригінальний об'єкт "збитковим" і, по суті, непридатним для використання. Для розуміння сказаного вище розглянемо таку навчальну програму.

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

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

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

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

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

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

Ось як виглядають результати виконання цієї програми.

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

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

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

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

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

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

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

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

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

Get(ObjA);

getch(); return 0;
}
```

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

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

Як бачимо, тут деструктор викликається тільки один раз, оскільки при передачі за посиланням аргумента функції Get() побітова копія об'єкта не створюється. Передача об'єкта за посиланням – типове вирішення описаної вище проблеми, але тільки у випадках, коли утворена ситуація дає змогу прийняти таке рішення, що буває далеко не завжди. На щастя, є більш загальне рішення: можна створити власну версію конструктора копії об'єкта. Це дасть змогу точно визначити, як саме потрібно створювати побітову копію об'єкта і тим самим уникнути описаних вище проблем. Але перед тим, як займемося конструктором копії, є сенс

розглянути ще одну ситуацію, у вирішенні якої ми також можемо отримати певний виграш завдяки створенню конструктора копії.

3.8 Особливості механізму повернення об'єктів функціями

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

Код програми 3.11. Демонстрація механізму повернення об'єкта функцією

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

class strClass {  // Оголошення класового типу
char str[80];
public:
void Set(char *s) { strcpy(str, s); }
void Show()  { cout << "Рядок: " << str << endl; }
};

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

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

return obj;
}

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

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

getch(); return 0;
}
```

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

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

об'єкт, який отримує повернуте функцією значення, все ще цю пам'ять використовує. Розглянемо таку некоректну версію попереднього коду програми.

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

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

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

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

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

return obj;
}

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

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

getch(); return 0;
}
```

Результати виконання цієї програми мають такий вигляд:

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

Зверніть увагу на те, що виклик деструктора ~strClass() відбувається тричі! Вперше він викликається при виході локального об'єкта obj з області видимості у момент його повернення з функції Init(). Другий виклик деструктора ~strClass() відбувається тоді, коли руйнується тимчасовий об'єкт, який повертається функцією Init(). Коли функція повертає об'єкт, то автоматично створюється невидимий (для Вас) тимчасовий об'єкт, який зберігає повернуте значення. У нашому випадку цей об'єкт просто є побітовою копією об'єкта obj. Отже, після повернення з функції використовується деструктор тимчасового об'єкта. Оскільки область

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

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

3.9 Механізми створення та використання конструктора копії

Одним з важливих форматів застосування перевизначеного конструктора є конструктор копії об'єкта. Як було показано в попередніх прикладах, при передачі об'єкта функції або при поверненні його з неї можуть виникати певні проблеми. Один із способів їх уникнення полягає у використанні конструктора копії, який є спеціальним типом перевизначеного конструктора. Основна причина використання конструктора копії полягає у тому, що при передачі об'єкта функції створюється побітова (тобто точна) його копія, яка передається параметру цієї функції. Проте трапляються ситуації, коли така однакова копія об'єкта небажана. Наприклад, якщо оригінальний об'єкт містить покажчик на динамічно виділену область пам'яті, то і покажчик, який належить побітовій копії об'єкта, також посилатиметься на ту ж саму область пам'яті. Отже, якщо у копію об'єкта будуть внесені зміни у вміст цієї області пам'яті, то ці зміни стосуватимуться також оригінального об'єкта! Понад це, внаслідок завершення роботи функції, побітова копія об'єкта буде зруйнована за викликом деструктора, що негативно відобразиться на початковому об'єкті.

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

Основна причина виникнення цієї проблеми полягає у створенні побітової копії об'єкта. Щоб запобігти їй, необхідно точно визначити, що повинно відбуватися, коли створюється така копія об'єкта, і, тим самим, уникнути небажаних побічних ефектів. Цього можна домогтися шляхом створення конструктора копії об'єкта.

Найпоширеніший формат конструктора копії об'єкта має такий вигляд:

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

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

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

один об'єкт безпосередньо ініціалізує інший об'єкт, як, наприклад, в оголошенні;

копія об'єкта передається як аргумент параметру функції;

генерується тимчасовий об'єкт (найчастіше як значення, що повертається функцією).

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

myClass:

myClass ObjX = ObjY; // Об'єкт ObjY безпосередньо ініціалізує об'єкт ObjX.

Fun1(ObjY); // Об'єкт ObjY передається як аргумент

ObjY = Fun2(); // Об'єкт ObjY приймає об'єкт, що повертається функцією.

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

3.10 Використання конструктора копії для ініціалізації одного об'єкта іншим

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

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

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

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

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

myClass ObjB = ObjA; // Викликається конструктор копії.

getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

Виділення р-пам'яті звичайним конструктором

Виділення р-пам'яті конструктором копії

Звільнення р-пам'яті

Звільнення р-пам'яті

Звільнення р-пам'яті

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

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

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

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

3.11 Механізм використання конструктора копії для передачі об'єкта функції

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

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

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

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

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

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

Get(ObjA);

getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Виділення р-пам'яті звичайним конструктором
Виділення р-пам'яті конструктором копії
```

*p= 10
Звільнення p-пам'яті
Звільнення p-пам'яті

У процесі виконання цієї програми тут відбувається таке: коли у функції main() створюється об'єкт ObjA, "стараннями" звичайного конструктора виділяється область пам'яті, адреса якої присвоюється покажчику ObjA.p. Потім об'єкт ObjA передається функції Get(), а саме – її параметру obj. У цьому випадку викликається конструктор копії, який для об'єкта ObjA створює побітову його копію з іменем obj.

Конструктор копії виділяє пам'ять для цієї копії, а значення покажчика на виділену область пам'яті присвоює члену p об'єкта-копії. Потім значення, яке адресується покажчиком p початкового об'єкта, записується в область пам'яті, адреса якої зберігається в покажчику p об'єкта-копії. Таким чином, області пам'яті, що адресуються покажчиками ObjA.p і obj.p, є різними та незалежними одна від одної, але значення (на які вказують ObjA.p і obj.p), що зберігаються в них, однакові. Якби конструктор копії у класі myClass не був завчасно визначений, то внаслідок створення за замовчуванням побітової копії члени ObjA.p і obj.p указували б на одну і ту саму область пам'яті.

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

3.12 Механізм використання конструктора копії при поверненні функцією об'єкта

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

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

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

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

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

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

Внаслідок виконання ця програма відображає на екрані такі результати:

Звичайний конструктор

Звичайний конструктор

Конструктор копії

Тут звичайний конструктор викликається двічі: перший раз при створенні об'єкта ObjA у функції main(), другий – при створенні об'єкта obj у функції Fun(). Конструктор копії викликається в ту мить, коли генерується тимчасовий об'єкт як значення, яке повертається з функції Fun(). Для багатьох програмістів-початківців "приховані" виклики конструкторів копії зазвичай є не зрозумілими. Проте практично кожен клас у професійно написаних програмах містить певний конструктор копії, без якого не можливо уникнути побічних ефектів, які виникають внаслідок створення за замовчуванням побітових копій об'єкта.

3.13 Конструктори копії та їх альтернативи

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

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

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

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

3.14 Поняття про ключове слово this

Під час кожного виклику функції-члена класу їй автоматично передається покажчик на об'єкт, який іменується ключовим словом this, для якого викликається ця функція. Покажчик this – це неявний параметр, який приймається всіма функціями-членами класу. Отже, в будь-якій функції-члені класу покажчик this використовується для посилання на викликуваний об'єкт.

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

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

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

```
c = 10;
```

Насправді попередня настанова є скороченою формою такої:
this->c = 10;
Щоби зрозуміти, як працює покажчик this, розглянемо таку навчальну програму.
Код програми 3.16. Демонстрація механізму застосування ключового слова this

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

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

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

Внаслідок виконання ця програма відображає на екрані число 100. Безумовно, цей приклад тривіальний, але у ньому показано, як можна використовувати покажчик this. З матеріалу наступних розділів Ви зрозумієте, чому покажчик this є таким важливим для написання програм мовою С++.

Контрольні запитання

1. Наведіть основне призначення функції-"друзі" класу.
2. Поясніть особливості перевизначення конструкторів.
3. Пояснчть особливості механізму динамічної ініціалізації конструктора.
4. Поясніть особливості механізму присвоєння об'єктів.
5. Поясніть особливості механізму передачі об'єктів функціям.
6. Наведіть загальну форму конструкторів, деструкторів.
7. Які виникають потенційні проблеми, при передачі об'єктів?
8. В чому особливості механізму повернення об'єктів функціями?
9. Поясніть механізми створення та використання конструктора копії.
10. Наведіть приклаж використання конструктора копії для ініціалізації одного об'єкта іншим.
11. Поясніть механізм використання конструктора копії для передачі об'єкта функції.
12. Поясніть механізм використання конструктора копії при поверненні функцією об'єкта.
13. Наведіть приклад застосування конструкторів копії та їх альтернативи.
14. Основне призначення ключового слова this?

РОЗДІЛ № 4. ОСОБЛИВОСТІ МЕХАНІЗМУ ПЕРЕ ВИЗНАЧЕННЯ ОПЕРАТОРІВ

1. Механізми перевизначення операторів з використанням функцій-членів класу
2. Перевизначення бінарних операторів додавання "+" і присвоєння "="
3. Перевизначення унарних операторів інкремента "++" та декремента "--"
4. Особливості реалізації механізму перевизначення операторів
5. Механізми перевизначення операторів з використанням функцій-не членів класу
6. Використання функцій-"друзів" класу для перевизначення бінарних операторів
7. Використання функцій-"друзів" класу для перевизначення унарних операторів
8. Перевизначення операторів відношення та логічних операторів

9. Особливості реалізації оператора присвоєння
10. Механізми перевизначення оператора індексації елементів масиву "[]"
11. Механізми перевизначення оператора виклику функцій "()"
12. Механізми перевизначення рядкових операторів
13. Конкатенація та присвоєння класу рядків з рядками класу
14. Конкатенація та присвоєння класу рядків з рядками, що закінчуються нульовим символом

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

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

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

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

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

4.1 Механізми перевизначення операторів з використанням функцій-членів класу

Операторна функція може бути членом класу або не бути ним. Операторні функції, які не є членами класу, визначаються як його "друзі". Операторні функції-члени і не члени класу відрізняються між собою механізмом перевизначення. Кожний з механізмів перевизначення операторів спробуємо розглянути більш детально на конкретних прикладах.

4.2 Перевизначення бінарних операторів додавання "+" і присвоєння "="

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

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

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

class kooClass {  // Оголошення класового типу
int x, y, z;  // Тривимірні координати
```

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

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

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

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

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

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

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

Внаслідок виконання ця програма відображає на екрані такі результати:

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

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

tmp.x = x + obj.x;

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

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

Зверніть увагу на те, що операторна функція operator+() повертає об'єкт типу kooClass, хоча вона могла б повертати значення будь-якого іншого допустимого типу, що визначається мовою програмування C++. Однак, той факт, що вона повертає об'єкт типу kooClass, дає змогу використовувати оператор додавання "+" у таких складних виразах, як ObjA + ObjB + ObjC – множинне додавання. Частина цього виразу (ObjA + ObjB) отримує результат типу kooClass, який потім додається до об'єкта ObjC. І якби ця частина виразу генерувала значення іншого типу (а не типу kooClass), то таке множинне додавання просто не відбулося б. На відміну від оператора додавання "+", оператор присвоєння "=" модифікує один зі своїх аргументів. Оскільки операторна функція operator=() викликається об'єктом, який розташований зліва від символу присвоєння "=", то саме цей об'єкт і модифікується внаслідок виконання операції присвоєння. Після виконання цієї операції значення, яке повертається перевизначеним оператором, містить об'єкт, який було вказано зліва від символу присвоєння. Наприклад, щоб можна виконувати настанови, подібні до такої (множинне присвоєння)

ObjA = ObjB = ObjC = ObjD;,

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

Операція присвоєння – це одне з найважливіших застосувань покажчика this.

4.3 Перевизначення унарних операторів інкремента "++" та декремента "--"

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

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

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

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

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

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

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

```
z++;

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

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

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

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

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

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

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

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

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

getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Координати об'єкта <A>:  x= 1, y= 2, z= 3
Координати об'єкта <B>:  x= 10, y= 10, z= 10
Координати об'єкта <C=A-B>:  x= -9, y= -8, z= -7
Координати об'єкта <C=A-B-C>: x= 0, y= 0, z= 0
Координати об'єкта <B=A>:  x= 1, y= 2, z= 3
Координати об'єкта <C=B>:  x= 1, y= 2, z= 3
Координати об'єкта <++C>:  x= 2, y= 3, z= 4
Координати об'єкта <C>:  x= 3, y= 4, z= 5
Координати об'єкта <A=++C>  x= 3, y= 4, z= 5
```

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

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

++ObjC;

і у постфіксній формі

ObjC++;.

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

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

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

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

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

kooClass tmp = *this;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

// Перевизначення префіксної форми унарного оператора зміни знаку "-".
```

```
kooClass kooClass::operator-()
{
x=-x; // Зміна знаку координат x, y і z
y=-y;
z=-z;
// Повернення модифікованого об'єкта операнда, адресованого покажчиком
return *this;
}

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

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

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

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

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

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

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

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

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

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

getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

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

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

4.4 Особливості реалізації механізму перевизначення операторів

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

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

:: .* ?

Оператор ".*" – це оператор спеціального призначення, який буде розглядатися нижче у цьому навчальному посібнику. Значення порядку слідування операндів. Перевизначаючи бінарні оператори, потрібно пам'ятати, що у багатьох випадках порядок слідування операндів має значення. Наприклад, вираз A + B комутативний, а вираз A – B – ні. Отже, реалізуючи перевизначені версії не комутативних операторів, потрібно пам'ятати, який операнд знаходиться зліва від символу операції, а який – праворуч від нього. Наприклад, у наведеному нижче коді програми продемонстровано механізм перевизначення оператора ділення для класу kooClass:

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

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

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

4.5 Механізми перевизначення операторів з використанням функцій-не членів класу

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

4.6 Використання функцій-"друзів" класу для перевизначення бінарних операторів

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

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

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

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

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

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

```
}

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

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

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

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

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

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

getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

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

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

```
tmp.obj + 10; // працюватиме
```

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

10 + tmp.obj; // не працюватиме

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

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

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

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

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

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

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

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

tmp.x = c * obi.x;
tmp.y = c * obi.y;
```

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

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

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

return tmp; // Повертає модифікований тимчасовий об'єкт
}
// Перевизначення бінарного оператора ділення "/".
kooClass operator/(int c, kooClass obi)
{
kooClass tmp; // Створення тимчасового об'єкта

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

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

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

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

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

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

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

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

ObjC = a / ObjB; // Ділення числа a на об'єкт ObjB
```

```
ObjC.Show("C=a/B");

getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

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

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

4.7 Використання функцій-"друзів" класу для перевизначення унарних операторів

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

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

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

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

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

Ця операторна функція не працездатна, оскільки тільки копія об'єкта, яка активізує виклик функції operator++(), передається функції через параметр obi. Таким чином, зміни в тілі функції operator++() не вплинуть на викликуваний об'єкт, позаяк вони змінюють тільки локальний параметр.

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

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

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

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

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

class kooClass {  // Оголошення класового типу
int x, y, z; // Тривимірні координати
public:
kooClass() {x = y = z = 0; }
kooClass(int c, int d, int f) {x = c; y = d; z = f; }
friend kooClass operator*(kooClass obi, kooClass obj);
kooClass operator=(kooClass obj);
// Ці функції для перевизначення оператора інкремента "++"
// використовують посилальні параметри.
friend kooClass operator++(kooClass &obi);
friend kooClass operator++(kooClass &obi, int notused);
void Show(char *s);
};

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

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

/* Перевизначення префіксної форми унарного оператора інкремента "++"  з використанням
"дружньої" функції класу. Для цього необхідне використання посилального параметра. */
kooClass operator++(kooClass &obi)
{
obi.x++;
obi.y++;
obi.z++;
return obi;
}

/* Перевизначення постфіксної форми унарного оператора інкремента "++" з використанням
"дружньої" функції класу. Для цього необхідне використання посилального параметра. */
kooClass operator++(kooClass &obi, int notused)
{
kooClass tmp = obi;
obi.x++;
obi.y++;
obi.z++;
return tmp; // Повертає модифікований тимчасовий об'єкт
}

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

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

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

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

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

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

ObjC++;    // Постфіксна версія інкремента
```

```
ObjC.Show("C++");

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

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

getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

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

4.8 Перевизначення операторів відношення та логічних операторів

Оператори відношення (наприклад, "==", "<", ">", "<=", ">=", "!=") і логічні оператори (наприклад, "&&" або "□□") також можна перевизначати, причому механізм їх реалізації не представляє жодних труднощів. Як правило, перевизначена операторна функція відношення повертає об'єкт того класу, для якого вона перевизначається. А будь-який перевизначений оператор відношення або логічний оператор повертає одне з двох можливих значень: true або false. Це відповідає звичайному застосуванню цих операторів і дає змогу використовувати їх в умовних виразах.

Розглянемо приклад перевизначення операторної функції дорівнює "==" для вже розглянутого вище класу kooClass:

```
// Перевизначення операторної функції дорівнює "=="
bool kooClass::operator==(kooClass obj)
{
if((x == obj.x) && (y == obj.y) && (z == obj.z))
return true;
else
return false;
}
```

Якщо вважати, що операторна функція operator==() вже реалізована, то такий код програми є абсолютно коректним:

```
kooClass ObjA, ObjB;
//...
if(ObjA == ObjB) cout << "ObjA = ObjB" << endl;
else cout << "ObjA не дорівнює ObjB" << endl;
```

Оскільки операторна функція operator==() повертає результат типу bool, то її можна використовувати для керування настановою if. Як вправу рекомендуємо самостійно реалізувати й інші оператори відношення та логічні оператори для класу kooClass.

4.9 Особливості реалізації оператора присвоєння

У попередньому розділі ми розглянули потенційну проблему, пов'язану з передачею об'єктів функціям і поверненням їх з них. У обох випадках проблема виникла під час використання конструктора за замовчуванням, який створює побітову копію об'єкта. Згадаймо (див. роз. 3.7), раніше це питання ми вирішували шляхом створення власного конструктора копії, який точно визначає, як повинна бути створена копія об'єкта.

Подібна проблема може виникати і під час присвоєння одного об'єкта іншому. За замовчуванням об'єкт, який знаходиться з лівого боку від операції присвоєння "=", отримує побітову копію об'єкта, який знаходиться справа. До негативних наслідків це може призвести у випадках, коли при створенні об'єкт виділяє певний ресурс (наприклад, пам'ять), а потім змінює його або звільняє. Якщо після виконання операції присвоєння об'єкт змінює або звільняє цей ресурс, то другий об'єкт також змінюється, оскільки він все ще використовує його ресурс. Вирішувати це питання можна також шляхом перевизначення оператора присвоєння.

Щоб до кінця зрозуміти суть описаної вище проблеми, розглянемо таку (некоректну) програму.

Код програми 4.7. Демонстрація механізму появи помилки, яка виникає при поверненні об'єкта з функції

```
#include <vcl>
#include <iostream>  // Для потокового введення-виведення
#include <conio>  // Для консольного режиму роботи
using namespace std;   // Використання стандартного простору імен
class strClass {  // Оголошення класового типу
char *s;
public:
strClass() { s = 0; }
strClass(const strClass &obj); // Оголошення конструктора копії
~strClass() {if(s) delete[]s; cout << "Звільнення s-пам'яті" << endl; }
void Show(char *c) { cout << c << s << endl; }
void Set(char *str);
};

// Визначення конструктора копії.
strClass::strClass(const strClass &obj)
{
s = new char[strlen(obj.s)+1];
strcpy(s, obj.s);
}

// Завантаження рядка.
void strClass::Set(char *str)
{
s = new char[strlen(str)+1];
strcpy(s, str);
}

// Ця функція повертає об'єкт типу strClass.
strClass Init()
{
```

```
char str[80];
strClass obj;

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

int main()
{
strClass Obj; // Створення об'єкта класу
// Присвоюємо об'єкт, повернутий функцією Init(), об'єкту Obj.
Obj = Init(); // Ця настанова генерує помилку!!!!
Obj.Show("s= ");
getch(); return 0;
}
```

Можливі результати виконання цієї програми мають такий вигляд:

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

Залежно від використовуваного компілятора, на екрані монітора Ви можете побачити переважно "сміття" або й ні. Програма може також згенерувати помилку тривалості її виконання. У будь-якому випадку помилки не минути. І ось чому. У цьому коді програми конструктор копії коректно обробляє повернення об'єкта функцією Init(). Згадаймо, у разі, коли функція повертає об'єкт, то для зберігання повернутого нею значення створюється тимчасовий об'єкт. Оскільки при створенні об'єкта-копії конструктор копії виділяє нову область пам'яті, то член-даних s початкового об'єкта і член-даних s об'єкта-копії вказуватимуть на різні області пам'яті, які, як наслідок, не стануть псувати один одного. Проте помилки не минути, якщо повернутий функцією об'єкт присвоюється об'єкту Obj, оскільки у процесі виконання операції присвоєння за замовчуванням створюється побітова його копія. У цьому випадку тимчасовий об'єкт, який повертається функцією Init(), копіюється в об'єкт Obj. Як наслідок, член obj.s вказує на ту ж саму область пам'яті, що і член s тимчасового об'єкта. Але після виконання операції присвоєння в процесі руйнування тимчасового об'єкта ця пам'ять звільняється. Отже, член obj.s тепер вказуватиме на вже звільнену пам'ять! Окрім цього, пам'ять, яка адресується членом obj.s, повинна бути звільнена і після завершення роботи коду програми, тобто удруге. Щоб запобігти цьому, необхідно перевизначити оператор присвоєння так, щоб об'єкт, який розташовується зліва від оператора присвоєння, виділяв власну область пам'яті. Реалізацію цього рішення покажемо у такій відкоректованій програмі.

Код програми 4.8. Демонстрація механізму появи помилки, яка може виникнути при поверненні об'єкта з функції

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

class strClass {  // Оголошення класового типу
char *s;
public:
strClass();  // Оголошення звичайного конструктора
strClass(const strClass &obj); // Оголошення конструктора копії
```

```
~strClass() { if(s) delete[]s; cout << "Звільнення s-пам'яті" << endl; }
void Show(char *c) { cout << c << s << endl; }
void Set(char *str);
// Перевизначений оператор присвоєння
strClass operator=(const strClass &obj);
};

// Визначення звичайного конструктора.
strClass::strClass()
{
s = new char ('\0'); // Член s вказує на NULL-рядок.
}
// Визначення конструктора копії.
strClass::strClass(const strClass &obj)
{
s = new char[strlen(obj.s)+1];
strcpy(s, obj.s);
}

// Завантаження рядка.
void strClass::Set(char *str)
{
s = new char[strlen(str)+1];
strcpy(s, str);
}

// Перевизначення оператора присвоєння "=".
strClass strClass::operator=(const strClass &obj)
{
/* Якщо виділена область пам'яті має недостатній
розмір, виділяється нова область пам'яті. */
if(strlen(obj.s) > strlen(s)) {
delete[]s;
s = new char[strlen(obj.s)+1];
}
strcpy(s, obj.s);

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

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

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

return obj;
}
```

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

// Присвоюємо об'єкт, повернутий функцією Init(), об'єкту Obj
Obj = Init(); // Тепер тут все гаразд!
Obj.Show("s= ");

getch(); return 0;
}
```

Ця програма тепер відображає такі результати (у припущенні, що на пропозицію "Введіть рядок: " Ви введете "Привіт").

```
Введіть рядок: Привіт
Звільнення s-пам'яті.
Звільнення s-пам'яті.
Звільнення s-пам'яті.
Привіт
Звільнення s-пам'яті.
```

Як бачимо, ця програма тепер працює коректно. Спробуйте детально проаналізувати програму і зрозуміти, чому виводиться кожне з повідомлень "Звільнення s-пам'яті.".

4.10 Механізми перевизначення оператора індексації елементів масиву "[]"

На додаток до традиційних перевизначених операторів мова програмування C++ дає змогу перевизначати і більш "екзотичні", наприклад, оператор індексації елементів масиву "[]". У мові програмування C++ (з погляду механізму перевизначення) оператор "[]" вважається бінарним. Його можна перевизначати тільки для класу і тільки з використанням функції-члена класу. Ось як виглядає загальний формат операторної функції-члена класу operator[]().

```
тип ім'я_класу::operator[](int індекс)
{
//...
}
```

Формально параметр індекс необов'язково повинен мати тип int, але оператор на функція operator[]() зазвичай використовують для забезпечення індексації елементів масивів, тому в загальному випадку як аргумент цієї функції передається цілочисельне значення. Припустимо, нехай створено об'єкт ObjA, тоді вираз ObjA[3] перетвориться в такий виклик операторної функції operator[]():

```
ObjA.operator[](3);
```

Іншими словами, значення виразу, що задається в операторі індексації елементів масиву "[]", передається операторній функції operator[]() як безпосередньо заданий аргумент. При цьому покажчик this вказуватиме на об'єкт ObjA, тобто об'єкт, який здійснює виклик цієї функції.

У наведеному нижче коді програми в класі aClass оголошується масив для зберігання трьох int-значень. Його конструктор ініціалізує кожного члена цього масиву. Перевизначена операторна функція operator[]() повертає значення елемента, що задається його параметром.

Код програми 4.9. Демонстрація механізму перевизначення оператора індексації елементів масиву "[]"

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

const int size = 3;
```

```
class aClass {  // Оголошення класового типу
int aMas[size];
public:
aClass() { for(int i=0; i<size; i++) aMas[i] = i*i; }
int operator[](int i) {return aMas[i]; }
};

int main()
{
aClass ObjA;

cout << "aMas[2]= " << ObjA[2] << endl; // Відображає число 4

cout << "Значення елементів масиву <A>:" << endl;
for(int i=0; i<3; i++)
cout << "aMas[" << i << "]= " << ObjA[i] << endl;

getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
a[2]= 4
Значення елементів масиву <A>:
a[0]= 0
a[1]= 1
a[2]= 4
```

Ініціалізація масиву aMas за допомогою конструктора (у цій і наступній програмах) здійснюється тільки з ілюстративною метою. У цьому коді програми функція operator[]() спочатку повертає значення 3-го елемента масиву aMas. Таким чином, вираз ObjA[2] повертає число 4, яке відображається настановою cout. Потім у циклі виводяться усі елементи масиву.

Можна розробити операторну функцію operator[]() так, щоб оператор індексації елементів масиву "[]" можна було використовувати як зліва, так і праворуч від оператора присвоєння. Для цього достатньо вказати, що значення, що повертається операторною функцією operator[](), є посиланням. Цю можливість продемонстровано у наведеному нижче коді програми.

Код програми 4.10. Демонстрація механізму перевизначення оператора індексації елементів масиву "[]" як зліва, так і праворуч від оператора присвоєння

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

const int size = 3;

class aClass {  // Оголошення класового типу
int aMas[size];
public:
aClass() { for(int i=0; i<size; i++) aMas[i] = i*i; }
int &operator[](int i) {return aMas[i]; }
};

int main()
{
```

```
aClass ObjA;

cout << "Значення елементів масиву <A>:" << endl;
for(int i=0; i<3; i++)
cout << "aMas[" << i << "]= " << ObjA[i] << endl;

// Оператор "[]" знаходиться зліва від оператора присвоєння "=".
ObjA[2] = 25;
cout << endl << "aMas[2]= " << ObjA[2]; // Тепер відображається число 25.

getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Значення елементів масиву <A>:
a[0]= 0
a[1]= 1
a[2]= 4

a[2]= 25
```

Оскільки операторна функція operator[]() тепер повертає посилання на елемент масиву, що індексується параметром i, то оператор індексації елементів масиву "[]" можна використовувати зліва від оператора присвоєння, що дасть змогу моди-фікувати будь-який елемент масиву. Одна з наявних переваг перевизначення оператора індексації елементів масиву "[]" полягає у тому, що за допомогою нього ми можемо забезпечити реалізацію безпечної індексації елементів масиву. Як уже зазначалося вище, у мові програмування C++ можливий вихід за межі масиву у процесі виконання програми без відповідного повідомлення (тобто без генерування повідомлення про динамічну помилку). Але, якщо створити клас, який містить масив, і надати доступ до цього масиву тільки через перевизначений оператор індексації елементів масиву "[]", то в процесі виконання програми можливе перехоплення індексу, значення якого вийшло за дозволені межі. Наприклад, наведений нижче код програми (в основу якої покладений програмний код попередньої) оснащена засобом контролю потрапляння індексу масиву в допустимий інтервал його перебування.

Код програми 4.11. Демонстрація прикладу організації безпечного масиву

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

const int size = 3;

class aClass {  // Оголошення класового типу
int aMas[size];
public:
aClass() { for(int i=0; i<size; i++) aMas[i] = i*i; }
int &operator[](int i);
};

// Забезпечення контролю потрапляння індексу масиву
// в допустимий інтервал його перебування.
int &aClass::operator[](int i)
{
if(i<0 || i> size-1) {
```

```
cout << endl << "Значення індексу " << i <<
" виходить за межі допустимого інтервалу" << endl;
getch(); exit(1);
}
return aMas[i];
}

int main()
{
aClass ObjA;
cout << "Значення елементів масиву <A>:" << endl;
for(int i=0; i<3; i++)
cout << "aMas[" << i << "]= " << ObjA[i] << endl;

ObjA[2] = 25;  // Оператор "[]" знаходиться в лівій частині.
cout << endl << "aMas[2]= " << ObjA[2];  // Відображається число 25.

ObjA[3] = 44;  // Виникає помилка тривалості виконання, оскільки
   // значення індексу 3 виходить за межі допустимого інтервалу.
getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Значення елементів масиву <A>:
a[0]= 0
a[1]= 1
a[2]= 4

a[2]= 25
```

Значення індексу 3 виходить за межі масиву.

У процесі виконання настанови

```
ObjA[3] = 44;
```

операторною функцією operator[]() перехоплюється помилка порушення меж допустимого інтервалу перебування індексу масиву, після чого програма відразу завершується, щоб не допустити потім ніяких потенційно можливих руйнувань.

4.11 Механізми перевизначення оператора виклику функцій "()"

Можливо, найбільш інтригуючим оператором, якого можна перевизначати, є оператор виклику функції "()". Під час його перевизначення створюється не новий спосіб виклику функцій, а операторна функція, якій можна передати довільну кількість параметрів. Почнемо з такого прикладу. Припустимо, що певний клас містить наведене нижче оголошення перевизначеної операторної функції:

```
int operator()(float f, char *p);
```

І якщо у програмі створюється об'єкт obj цього класу, то настанова

```
obj(99.57, "перевизначення");
```

перетвориться в такий виклик операторної функції operator():

```
operator()(99.57, "перевизначення");
```

У загальному випадку при перевизначенні оператора виклику функцій "()" визначаються параметри, які необхідно передати функції operator(). Під час використання оператора "()" у програмі задані аргументи копіюються в ці параметри. Як завжди, об'єкт, який здійснює виклик операторної функції (obj у наведеному прикладі), адресується покажчиком this. Розглянемо приклад перевизначення оператора виклику функцій "()" для класу kooClass. Тут створюється новий об'єкт класу kooClass, координати якого є

результатом підсумовування відповідних значень координат об'єкта і значень, що передаються як аргументи.

Код програми 4.12. Демонстрація механізму перевизначення оператора виклику функцій "()"

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

class kooClass {  // Оголошення класового типу
int x, y, z;  // Тривимірні координати
public:

kooClass() { x = y = z = 0; }
kooClass(int c, int d, int f) {x = c; y = d; z = f; }
kooClass operator()(int a, int b, int c);
void Show(char *s);
};

// Перевизначення оператора виклику функцій "()".
kooClass kooClass::operator()(int a, int b, int c)
{
kooClass tmp; // Створення тимчасового об'єкта
tmp.x = x + a;
tmp.y = y + b;
tmp.z = z + c;
return tmp;  // Повертає модифікований тимчасовий об'єкт
}

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

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

ObjB = ObjA(10, 11, 12); // Виклик функції operator()

ObjA.Show("A");
ObjB.Show("B");
getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

Координати об'єкта <A>: x= 1, y= 2, z= 3

Координати об'єкта <B>: x= 11, y= 13, z= 15

Не забувайте, що при перевизначенні оператора виклику функцій "()" можна використовувати параметри будь-якого типу, та і сама операторна функція operator() може повертати значення будь-якого типу. Вибір типу повинен диктуватися потребами конкретних програм.

4.12 Механізми перевизначення рядкових операторів

За винятком таких операторів, як new, delete, ->, ->* і "кома", решту C++-оператори можна перевизначати таким самим способом, як це було показано в попередніх прикладах. Перевизначення операторів new і delete вимагає застосування спеціальних методів, повний опис яких наведено в розд. 8 (він присвячений обробленню виняткових ситуацій). Оператори ->, ->* і "кома" – це спеціальні оператори, детальний перегляд яких виходить за рамки цього навчального посібника. Читачі, яких цікавлять інші приклади перевизначення операторів, можуть звернутися до такої книги [27]. У цьому ж підрозділі розглядатимемо механізм перевизначення тільки рядкових операторів.

4.13 Конкатенація та присвоєння класу рядків з рядками класу

Завершуючи тему перевизначення операторів, розглянемо приклад, який часто називають квінтесенцією прикладів, присвячених вивченню механізму пере-визначення операторів класу рядків. Незважаючи на те, що C++-підхід до рядків (які реалізуються у вигляді символьних масивів, що завершуються нулем, а не як окремий тип) є дуже ефективним і гнучким, проте початківці C++-програмування часто стикаються з недоліком у понятійній ясності реалізації рядків, яка наявна в таких мовах, як BASIC. Звичайно ж, цю ситуацію неважко змінити, оскільки у мові програмування C++ існує можливість визначити клас рядків, який забезпечуватиме їх реалізацію подібно до того, як це зроблено в інших мовах програмування. Правду кажучи, на початкових етапах розвитку мови програмування C++ реалізація класу рядків була забавою для програмістів. І хоча стандарт мови програмування C++ тепер визначає рядковий клас, який описано далі у цьому навчальному посібнику, проте спробуйте самостійно реалізувати простий варіант такого класу. Цей приклад наочно ілюструє потужність механізму перевизначення операторів класу рядків.

Код програми 4.13. Демонстрація механізму конкатенації та присвоєння класу рядків

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

class strClass { // Оголошення класового типу
char string[80];
public:
strClass(char *str = "") { strcpy(string, str); }
strClass operator+(strClass obj); // Конкатенація рядків
strClass operator=(strClass obj); // Присвоєння рядків
// Виведення рядка
void Show(char *s) { cout << s << string << endl; }
};
```

Як бачимо, в класі strClass оголошується закритий символьний масив string, призначений для зберігання рядка. У наведеному прикладі домовимося, що розмір рядків не перевищуватиме 79 байтів. У реальному ж класі рядків пам'ять для їх зберігання повинна виділятися динамічно, однак це обмеження зараз діяти не буде. Окрім цього, щоби не захаращувати логіку цього прикладу, ми вирішили звільнити цей клас (і його функції-члени) від контролю виходу за межі масиву. Безумовно, в будь-якій справжній реалізації подібного класу повинен бути забезпечений повний контроль за помилками.

Клас strClass має один конструктор, який можна використовувати для ініціалізації масиву string з використанням заданого значення або для присвоєння йому порожнього рядка у разі відсутності ініціалізації. У цьому класі також оголошуються два перевизначені оператори, які виконують операції конкатенації та присвоєння. Нарешті, клас strClass містить функцію Show(), яка виводить рядок на екран.

Ось як виглядають коди операторних функцій operator+() і operator=():

```
// Конкатенація двох рядків
```

```
strClass strClass::operator+(strClass obj)
{
strClass tmp; // Створення тимчасового об'єкта
strcpy(tmp.string, string);
strcat(tmp.string, obj.string);
return tmp; // Повертає модифікований тимчасовий об'єкт
}

// Присвоєння одного рядка іншому
strClass strClass::operator=(strClass obj)
{
strcpy(string, obj.string);
// Повернення модифікованого об'єкта операнда, адресованого покажчиком
return *this;
}
```

Маючи визначення цих операторних функцій, продемонструємо, як їх можна використовувати на прикладі наведеної нижче основної функції main():

```
int main()
{
strClass ObjA("Всім "), ObjB("привіт"), ObjC;
ObjA.Show("A: ");
ObjB.Show("B: ");

ObjC = ObjA + ObjB;
ObjC.Show("C=A+B: ");
getch(); return 0;
}
```

Спочатку вона конкатенує рядки (об'єкти класу strClass) ObjA і ObjB, а потім присвоює результат конкатенації рядку ObjC.

Внаслідок виконання ця програма відображає на екрані такі результати:

A: Привіт
B: усім
C=A+B: Привіт усім

4.14 Конкатенація та присвоєння класу рядків з рядками, з нульовим символом

Потрібно мати на увазі, що оператори присвоєння "=" і конкатенації "+" визначено тільки для об'єктів типу strClass. Наприклад, наведена нижче настанова не працездатна, оскільки вона є спробою присвоїти об'єкту ObjA рядок, який завершується нульовим символом:

```
ObjA = "Цього поки що робити не можна.";
```

Але клас strClass, як буде показано далі, можна удосконалити і дати йому змогу виконувати такі настанови. Для розширення переліку операцій, підтримуваних класом strClass(наприклад, щоб можна було об'єктам типу strClass присвоювати рядки з завершальним нуль-символом, або конкатенувати рядок, який завершується нульовим символом, з об'єктом типу strClass), необхідно перевизначити оператори "=" і "+" ще раз. Спочатку змінимо оголошення класу:

```
// Перевизначення рядкового класу: остаточний варіант
class strClass { // Оголошення класового типу
char string[80];
public:
class strClass { // Оголошення класового типу
char string[80];
public:
strClass(char *str = "") { strcpy(string, str); }
// Конкатенація об'єктів типу strClass
```

```
strClass operator+(strClass obj);
// Конкатенація об'єкта з рядком, що завершується нулем
strClass operator+(char *str);
// Присвоєння одного об'єкта типу strClass іншому
strClass operator=(strClass obj);
// Присвоєння рядка, що завершується нулем, об'єкту типу strClass
strClass operator=(char *str);
void Show(char *s) { cout << s << string << endl; }
};
```

Потім реалізуємо перевизначення операторних функцій operator+() і operator=():

```
// Присвоєння рядка об'єкту типу strClass, що завершується нулем
strClass strClass::operator=(char *str)
{
strClass tmp; // Створення тимчасового об'єкта

strcpy(string, str);
strcpy(tmp.string, string);

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

// Конкатенація рядка з об'єктом типу strClass, що завершується нулем
strClass strClass::operator+(char *str)
{
strClass tmp; // Створення тимчасового об'єкта

strcpy(tmp.string, string);
strcat(tmp.string, str);

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

Уважно проаналізуйте коди цих функцій. Зверніть увагу на те, що правий аргумент є не об'єктом типу strClass, а покажчиком на символьний масив, який завершується нулем, тобто звичайним C++-рядком. Але обидві ці функції повертають об'єкт типу strClass. І хоча теоретично вони могли б повертати об'єкт будь-якого іншого типу, весь сенс їх існування і полягає у тому, щоб повертати об'єкт типу strClass, оскільки результати цих операцій приймаються також об'єктами типу strClass. Перевага визначення рядкової операції, у якій як правий операнд бере участь рядок, який завершується нульовим символом, полягає у тому, що воно дає змогу писати деякі настанови в природній формі. Наприклад, наведені нижче настанови є цілком законними:

```
strClass a, b, c;
a = "Привіт усім"; // Присвоєння рядка, який завершує нулем, об'єкту
c = a + "Георгій";  // Конкатенація об'єкта з рядком, що завершується нулем
```

Наведений нижче код програми містить додаткові визначення операторів присвоєння "=" і конкатенації "+".

Код програми 4.14. Демонстрація механізму конкатенації та присвоєння класу рядків з рядками, що закінчуються нульовим символом

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

```
class strClass { // Оголошення класового типу
char string[80];
public:
strClass(char *str = "") { strcpy(string, str); }
// Конкатенація об'єктів типу strClass
strClass operator+(strClass obj);
// Конкатенація об'єкта з рядком, що завершується нулем
strClass operator+(char *str);
// Присвоєння одного об'єкта типу strClass іншому
strClass operator=(strClass obj);
// Присвоєння рядка об'єкту типу strClass, що завершується нулем
strClass operator=(char *str);
void Show(char *s) { cout << s << string << endl; }
};

strClass strClass::operator+(strClass obj)
{
strClass tmp; // Створення тимчасового об'єкта

strcpy(tmp.string, string);
strcat(tmp.string, obj.string);

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

strClass strClass::operator=(strClass obj)
{
strcpy(string, obj.string);
// Повернення модифікованого об'єкта операнда, адресованого покажчиком
return *this;
}

strClass strClass::operator=(char *str)
{
strClass tmp; // Створення тимчасового об'єкта

strcpy(string, str);
strcpy(tmp.string, string);

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

strClass strClass::operator+(char *str)
{
strClass tmp; // Створення тимчасового об'єкта

strcpy(tmp.string, string);
strcat(tmp.string, str);
return tmp; // Повертає модифікований тимчасовий об'єкт
}

int main()
```

```
{
strClass ObjA("Привіт "), ObjB("всім"), ObjC;

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

ObjC = ObjA + ObjB;
ObjC.Show("C=A+B: ");

ObjA = "для програмування, тому що";
ObjA.Show("A: ");

ObjB = ObjC = "C++ це супер";
ObjC = ObjC + " " + ObjA + " " + ObjB;
ObjC.Show("C: ");

getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

A: Привіт
B: усім
C=A+B: Привіт усім
A: для програмування, тому що
C: C++ це супер для програмування, тому що C++ це супер

Перш ніж переходити до наступного розділу, спробуйте переконатися у тому, що до кінця розумієте, як отримано ці результати. Тепер постарайтеся також самостійно визначати ще деякі інші операції над рядками. Наприклад, спробуйте визначити операцію видалення підрядка на основі оператора вилучення "-". Зокрема, якщо рядок об'єкта A містить фразу "Це важкий-важкий тест", а рядок об'єкта B –фразу "важкий", то обчислення виразу A-B дасть у підсумку "Це – тест". У цьому випадку з початкового рядка були видалені всі входження підрядка "важкий". Визначте також "дружню" функцію, яка б давала змогу рядку, що завершується нулем, знаходитися зліва від оператора конкатенації "+". Нарешті, додайте у програму код, який забезпечує контроль за помилками – виходу індексу за межі масиву.

Контрольні запитання

1. Поясніть механізми перевизначення операторів з використанням функцій-членів класу
2. Наведіть приклад перевизначення бінарних операторів додавання "+" і присвоєння "=".
3. Наведіть приклад перевизначення унарних операторів інкремента "++" та декремента "--".
4. Наведіть особливості реалізації механізму перевизначення операторів.
5. Поясніть механізми перевизначення операторів з використанням функцій-не членів класу.
6. Наведіть приклад використання функцій-"друзів" класу для перевизначення бінарних операторів.
7. Наведіть приклад використання функцій-"друзів" класу для перевизначення унарних операторів
8. Наведіть приклад перевизначення операторів відношення та логічних операторів.
9. Наведіть особливості реалізації оператора присвоєння.
10. Поясніть механізми перевизначення оператора індексації елементів масиву "[]".
11. Поясніть механізми перевизначення оператора виклику функцій "()".
12. Поясніть механізми перевизначення рядкових операторів.
13. Наведіть приклад застосування конкатенації та присвоєння класу рядків з рядками класу.
14. Наведіть приклад застосування конкатенації та присвоєння класу рядків з рядками, що закінчуються нульовим символом.

ЛІТЕРАТУРА

Основна література

1. Грицюк, Ю. Обєктно-оріентовне програмування момвою С++, Ю. Грицюк, Т. Рак. – Львів, Вид-во ЛДУ БЖД, 2011. – 202 с.
2. Лафоре Р. Объектно-ориентированное программирование в С++, Лафоре Р. – М.: Питер, 2004. – 902 с.
3. Голицына О.Л. Основы алгоритмизации и программирования: учеб. Пособие. - М.: ФОРУМ, 2008. – 432 с.

Додаткова література

1. Иванова Г.С. Основы программирования: Учебник для вузов. – М: Изд-во МГТУ им. Н.Э. Баумана, 2002. – 416 с.
2. Иванова Г.С. Объектно-ориентированное программирование: Учебник для вузов. – М: Изд-во МГТУ им. Н.Э. Баумана, 2001. – 320 с

Навчальне видання

Конспект лекцій з дисципліни «Програмування» (частина 2) для студентів **напряму 6.170103 – Управління інформаційною безпекою**

Укладач:
Катерина Олексіївна Трифонова

Підписано до друку _____ Формат 60х84/16. Папір газетний. Друк офсетний. 0,87 ум. друк. арк. 0,94 обл. - вид. арк.
Тираж 100 пр. Зам. №

Одеський державний політехнічний університет
65044, Одеса, пр. Шевченка, 1