Ю.О. Міловідов

ОБ’ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ

Навчальний посібник

3-тє видання, доповнене і перероблене

Київ
2025

УДК 004.4(072)

М 60

Рекомендовано до видання рішенням вченої ради Національного університету біоресурсів і природокористування України (Протокол № 4 від 22 жовтня 2025 року)

Рецензенти:

Аль Аморі Алі доктор. техн. наук, професор, завідувач кафедри інформаційно-аналітичної діяльності та інформаційної безпеки Національного транспортного університету.

Іларіонов О.М. канд. техн. наук, доцент, завідуюч кафедри факультету інформаційних технологій Київського Національного університету імени Тараса Шевченка.

Криворучко О.В. доктор. техн. наук, професор, професор кафедри комп'ютерних систем, мереж і кібербезпеки Національного університету біоресурсів і природокористування України.

М 60 Об’єктно-орієнтоване програмування : навчальний посібник / Ю.О. Міловідов. Київ : НУБіП України, 2025. 334 с.

Зміст навчального посібника відповідає освітньо-професійній програмі підготовки бакалаврів за спеціальностями 121 «Інженерія програмного забезпечення», 122 «Комп’ютерні науки» та програмі дисципліни «Об’єктно-орієнтоване програмування».

Розглянуто поняття та принципи об’єктно-орієнтованого програмування.

Проаналізовано проблеми розробки класів об’єктів, побудови їх ієрархій, у сучасних мовах програмування.

У першій частині посібника розглянути базові принципи об’єктно-орієнтованого програмування (ООП) і об’єктно-орієнтована технологія програмування на мові C++, у другій частині розглядаються особливості ООП на мові C#.

Посібник може бути корисним для студентів, викладачів, науковців та аспірантів вищих навчальних закладів, а також для всіх, хто цікавиться вивченням об’єктно-орієнтованого програмування.

УДК 004.4(072

Міловідрв Ю.О. 2025
© НУБіП України

ВІДОМОСТІ ПРО АВТОРІВ

Міловідов Юрій Олегович Старший викладач кафедри комп'ютерних наук Національного університету біоресурсів і природокористування. Викладає дисципліни «Об'єктно-орієнтоване програмування», «Програмна технологія .NET», «Алгоритми і структури даних». Наукові інтереси пов'язані з розробкою програмного забезпечення під платформу .NET. Автор 4-х навчальних посібників, має сертифікат від Microsoft Technology Associate for Software Development Fundamentals.
Електронна адреса: yurii_milovidov@nubip.edu.ua

ЗМІСТ

ВСТУП 7
ЧАСТИНА 1. БАЗОВІ ПОНЯТТЯ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ 8
РОЗДІЛ 1. ПРИНЦИПИ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ 8
1.1 Об'єктно-орієнтоване мислення ... 8
1.2 Об'єктно-орієнтований підхід ... 9
1.3 Загальні ознаки об’єктно-орієнтованого програмування ... 13
РОЗДІЛ 2. КЛАСИ І ОБ’ЄКТИ 18
2.1 Основи поняття класу ... 18
2.2 Доступ до членів класу ... 22
2.3 Конструктори і деструктори ... 23
2.4 Параметризовані конструктори ... 26
2.5 Вбудовані функції ... 29
2.6 Використання вбудованих функцій у визначенні класу ... 30
2.7 Масиви об'єктів ... 31
2.8 Вказівники на об'єкти ... 32
РОЗДІЛ 3. БІЛЬШ ДЕТАЛЬНИЙ РОЗГЛЯД КЛАСІВ 39
3.1 «Дружні» функції ... 39
3.2 Перевантаження конструкторів ... 44
3.3 Присвоювання об'єктів. ... 46
3.4 Передача об’єктів функціям ... 47
3.5 Конструктори, деструктори і передача об'єктів ... 48
3.6 Потенційні проблеми при передачі параметрів ... 49
3.7 Повернення об'єктів функціями ... 51
3.8 Створення і використання конструктора копії ... 53
3.9 Ключове слово this ... 55
РОЗДІЛ 4. ПЕРЕВАНТАЖЕННЯ ОПЕРАТОРІВ 63
4.1 Перевантаження операторів з використанням функцій не членів класу ... 63
4.2 Перевантаження операторів з використанням функцій членів класу ... 65
4.3 Перевантаження операторів відношення і логічних операторів ... 68
4.4 Перевантаження оператора індексації масивів ([]) ... 69
РОЗДІЛ 5. ВІДНОСИНИ МІЖ КЛАСАМИ 76
5.1 Спадкування ... 76
5.2 Управління доступом до членів базового класу ... 79
5.3 Використання захищених членів ... 80
5.4 Множинне спадкування ... 82
5.5 Віртуальні базові класи ... 83

5.6 Залежність ... 88
5.7 Асоціація. Агрегація і композиція ... 89
5.8 Створення діаграм класів засобами Visual Studio ... 93
РОЗДІЛ 6. ВІРТУАЛЬНІ ФУНКЦІЇ І ПОЛІМОРФІЗМ 99
6.1 Вказівники на похідні типи ... 99
6.2 Віртуальні функції ... 100
6.3 Суто віртуальні функції та абстрактні класи ... 106
6.4 Класи-інтерфейси ... 108
6.5 Динамічна ідентифікація типів і оператори приведення типу ... 113
6.6 Обробка виняткових ситуації ... 119
РОЗДІЛ 7. ШАБЛОНИ 132
7.1 Узагальнені функції ... 132
7.2 Узагальнені класи ... 134
7.3 Стандартна бібліотека шаблонів ... 137
ЧАСТИНА 2. ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ НА МОВІ C# 147
РОЗДІЛ 8. ВВЕДЕННЯ В C# 147
8.1 Мова C# і платформа .NET ... 147
8.2 Початок роботи. Visual Studio ... 153
8.3 Базові поняття мови C# ... 157
РОЗДІЛ 9. КЛАСИ І ОБ'ЄКТИ 173
9.1 Визначення класу ... 173
9.2 Конструктори ... 174
9.3 Структури ... 178
9.4 Типи значень і посилальні типи ... 181
9.5 Модифікатори доступу ... 189
9.6 Властивості та інкапсуляція ... 191
9.7 Індексатори ... 195
9.8 Перевантаження операторів ... 198
9.9 Статичні члени і модифікатор static ... 202
РОЗДІЛ 10. СПАДКУВАННЯ І ПОЛІМОРФІЗМ 211
10.1 Спадкування ... 211
10.2 Перетворення типів ... 216
10.3 Віртуальні методи і властивості ... 223
10.4 Абстрактні класи та члени класів ... 232
10.5 Клас object і його методи ... 235
10.6 Узагальнення ... 237
РОЗДІЛ 11. ІНТЕРФЕЙСИ 246

11.1 Введення в інтерфейси 246
11.2 Множинна реалізація інтерфейсів 248
11.3 Інтерфейси в перетвореннях типів 249
11.4 Спадкування інтерфейсів 250
11.5 Явна та неявна реалізація 251
11.6 Динамічна зміна поведінки 253
РОЗДІЛ 12. ДЕЛЕГАТИ, ПОДІЇ, ЛЯМБДА-ВИЗАЗИ 261
12.1 Делегати 261
12.2 Застосування делегатів 268
12.3 Події 274
12.4 Анонімні методи 279
12.5 Лямбда-вирази 283
12.6 Коваріантність і контраваріантність делегатів 288
12.7 Делегати Action, Predicate і Func 290
РОЗДІЛ 13. СТВОРЕННЯ ГРАФІЧНИХ ІНТЕРФЕЙСІВ 297
13.1 Створення графічного застосування 297
13.2 Робота з формами 300
13.3 Основні властивості форм 302
13.4 Події форми 304
13.4 Елементи управління 305
3.7 Динамічне додавання елементів 306
13.8 Діалогові вікна 307
РОЗДІЛ 14. ОБРОБКА ВИНЯТКОВИХ СИТУАЦІЙ 314
14.1 Перехоплення виключень 314
14.2 Блок catch і фільтри виключень 317
14.3 Гарантоване виконання 319
14.4 Створення класів виключень 320
14.5 Пошук блоку catch при обробці виключень 322
СПИСОК ТЕРМІНІВ 331
ЛІТЕРАТУРА 334

ВСТУП

Об’єктно-орієнтоване програмування (ООП) – популярна методологія розробки програмного забезпечення, в якій програма є сукупністю об’єктів, кожний із яких є екземпляром деякого класу.

Об’єктно-орієнтоване програмування – одна з парадигм програмування, яка розглядає програму як множину «об'єктів», що взаємодіють між собою. Основи ООП – інкапсуляція, успадкування та поліморфізм. Сьогодні багато мов програмування (зокрема, Java, C#, C++, VB, Python, PHP, Ruby та Objective-C) підтримують ООП.

Пропонуємо Вашій увазі третє видання посібника «Об’єктно-орієнтоване програмування», який буде корисний не тільки студентам, що спеціалізуються в області комп'ютерних наук і програмної інженерії, а й фахівцям з розробки програм. Необхідність перевидання обґрунтовується тім, що після другого видання посібника у 2022 році до липня 2025 року відбулася низка ключових оновлень у C++ та C#.

У 2023 році офіційно прийнята версія мови C++23. в якій запроваджено багато зручностей й покращень у константних виразах, бібліотеках і синтаксисі.

У 2022 році вийшла версія мови C#11 під оновлену платформу .NET7.0, C# 12 (.NET 8, реліз у серпні–листопаді 2023), C# 13 (.NET 9, реліз у листопаді 2024). C# стає дедалі виразнішою мовою з гнучкими лямбдами, новим синтаксисом колекцій і потужними конструкціями для generics. NET платформа фокусується на продуктивності, швидкому старті, підтримці AOT (Ahead Of Time compilation), просунутих API (Application Programming Interface) і сучасних фреймворках Blazor, MAUI (Multi platform Application User Interface).

У посібнику розглядається кожен аспект об'єктно-орієнтованого програмування та надається детальний опис. Проаналізовано проблеми розробки класів об’єктів, побудови їх ієрархій.

З точки зору практичного застосування, розглянуті та проаналізовані питання реалізації об’єктно-орієнтованих програм різними мовами програмування.

У першій частині посібника розглянути базові принципи ООП і об’єктно-орієнтована технологія програмування на мові C++, у другій частині розглядаються сучасні технології об’єктно-орієнтованого програмування на мові C#.

Крім того, посібник доповнений новими матеріалами – класи-інтерфейси, обробка виняткових ситуації для C++ і C#, сворення графічних застосувань у C#.

Теоретичний матеріал, що подається у посібнику, підкріплюється практичними завданнями до виконання, тестами для закріплення матеріалу і контрольними питаннями.

ЧАСТИНА 1. БАЗОВІ ПОНЯТТЯ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ

РОЗДІЛ 1. ПРИНЦИПИ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ

1.1 Об'єктно-орієнтоване мислення

Протягом всієї історії програмування ускладнення програм змушувало програмістів шукати шляхи, які б дозволили впоратися зі складністю.

Складну систему можна представити як сукупність взаємодіючих об'єктів. Кожен об'єкт у такій системі моделює поведінку об'єкта реального світу.

Складність великих систем перевершує можливості однієї людини. Ніхто не може тримати в пам'яті все і думати про всю систему не тільки в цілому, але і в деталях. Тому складні системи доводиться розробляти багатьом програмістам, які намагаються впоратися з дедалі більшою складністю такої системи. Як змінити стан справ?

1.1.1 Декомпозиція

Спосіб управління складними системами відомий ще зі стародавніх часів: «Розділяй і володарюй». Поділ програми на дрібні шматочки, кожен з яких може працювати незалежно, – ось перший крок в боротьбі зі складністю.

У структурному програмуванні під декомпозицією розумілося розділення алгоритмів, де кожен алгоритм виконував один з етапів загального процесу. Основою цього принципу є проектування «зверху вниз».

Можна розглядати проблему декомпозиції і з іншого боку: система поділяється за ознакою приналежності її елементів різним абстракціям даної проблемної області. Кожен об'єкт в такій системі моделює поведінку об'єкта реального світу. Така декомпозиція називається об'єктно-орієнтованою декомпозицією.

Об'єктна декомпозиція має багато переваг перед алгоритмічної. Ця перевага особливо явно проявляється під час проектування великої складної системи.

1.1.2 Абстракція

У людини є засіб, що дозволяє боротися зі складністю і він з’явився задовго до появи перших комп'ютерів. Люди абстрагувалися від складності. Не маючи можливості обійняти неосяжне, вони ігнорували дрібні і не важливі деталі, зосередивши свою увагу на основних, ключових моментах. Ми намагаємося створити якусь узагальнену модель дійсності. Об'єкти, отримані в результаті декомпозиції, являють собою абстракції реального світу. Абстракція фокусує увагу на суттєві характеристики об'єкта (рис. 1.1).

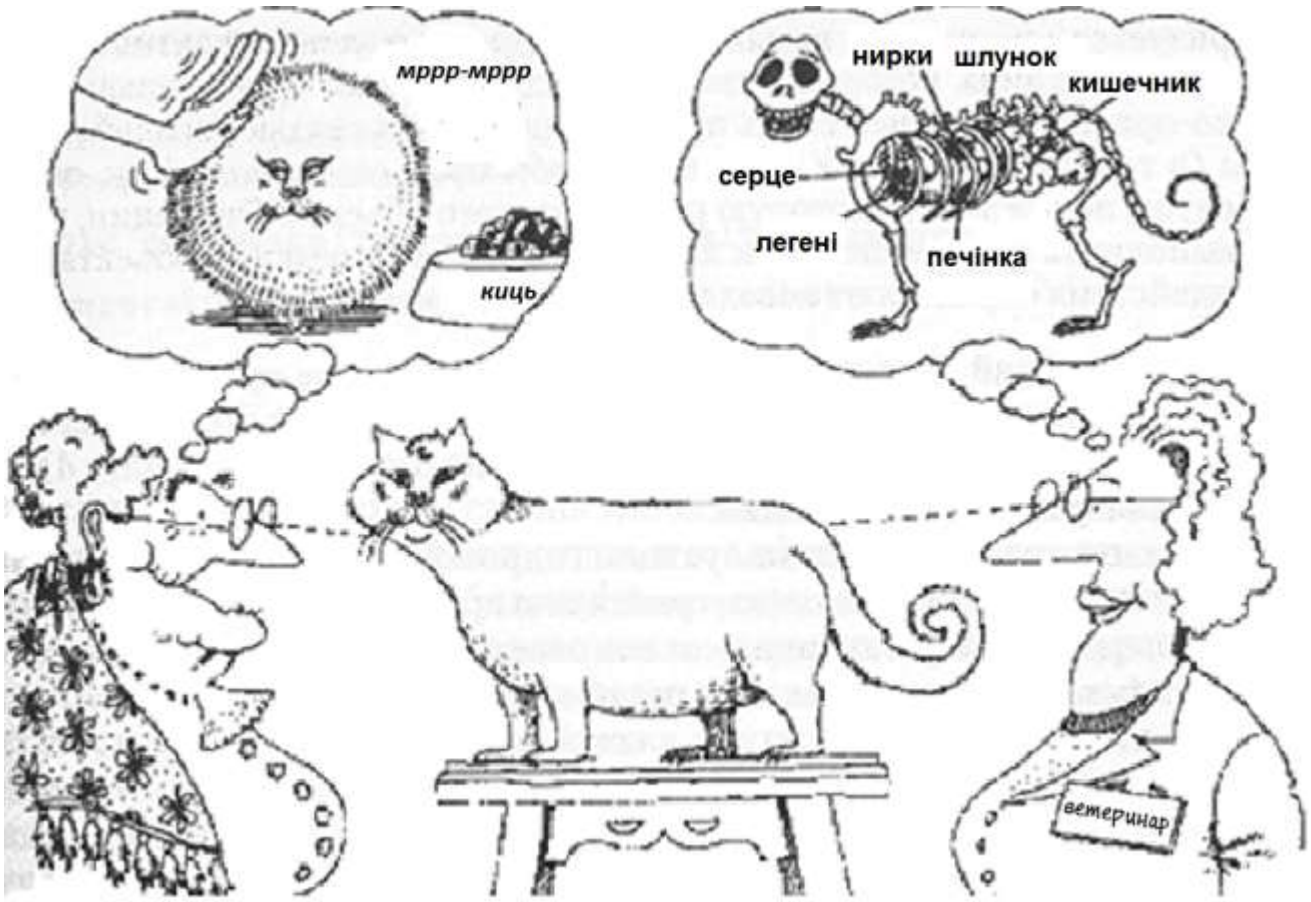


Рис. 1.1. Абстракція фокусує увагу на суттєві характеристики об'єкта з точки зору спостерігача

В об’єктно-орієнтованому програмуванні все побудовано навколо абстракції, але абстракція притаманна для будь-якого програмування, а не тільки для об’єктно-орієнтованого.

1.1.3 Ієрархія

Багато об'єктів мають загальну поведінку або схожу на поведінку інших об'єктів. Розділивши об'єкти на групи, виділивши найбільш загальну модель поведінки і залишаючи спеціалізовану поведінку на деякі виділені об'єкти, ми сформуємо ієрархію об'єктів, що значно спрощує розуміння складної системи.

1.2 Об'єктно-орієнтований підхід

Об'єктно-орієнтована технологія ґрунтується на так званій об'єктній моделі. Основними властивостями цієї моделі є: абстрагування, інкапсуляція, модульність, ієрархічність, типізація, паралелізм і збереженість. Це не нові принципи, але в сукупності вони зустрічаються тільки в об'єктному моделюванні.

Об'єктна модель є основою трьох китів розробки: об'єктно-орієнтованого аналізу, об'єктно-орієнтованого проектування та об'єктно-орієнтованого програмування.

1.2.1 Нова парадигма

Об'єктно-орієнтоване програмування (ООП) стало основною методологією програмування 90-х років. ООП є продуктом 25-річної практики і включає ряд мов: Lisp, Clu, Actor, Eiffel, Objective C, C++, Java, C#, Python та інші. Це техніка програмування, яка дозволяє фіксувати поведінку реального світу таким способом, при якому деталі реалізації приховані.

ООП часто називають новою парадигмою програмування. Інші відомі парадигми програмування: директивна (структурне програмування - Pascal, C), логічна – Prolog, функціональна – Lisp, Effel. Парадигми в програмуванні визначають, як проводити обчислення, як повинна бути структурована і організована робота комп'ютера.

1.2.2 Мова програмування

Мова, якою ми розмовляємо, безпосередньо впливає на спосіб сприйняття світу. Це стосується не тільки природних мов, але і штучних мов – мов програмування.

Студенти задають питання, пов'язані з вибором мови програмування, яка б відповідала сучасним вимогам.

З плином часу одні мови розвивалися, набували нових рис і залишилися затребувані, інші втратили свою актуальність і сьогодні представляють в кращому випадку чисто теоретичний інтерес. На рисунку 1.2 зображено генеалогічне дерево найбільш відомих мов програмування, на якому можна простежити їх еволюцію. Звичайно, ця картинка не є повною, для цього знадобилася б не одна сторінка.

Щоб ефективно використовувати ООП, потрібно дивитися на світ іншим способом: не з точки зору структурних мов.

Мова програмування, яка претендує називатися об'єктно-орієнтованою, повинна надавати зручний механізм підтримки об'єктно-орієнтованого стилю програмування.

Не слід говорити про те, яка мова краща за іншу. Можна лише відзначити, чи є в мові засоби, достатні для підтримки стилю програмування.

Майже всі сучасні мови програмування підтримують об’єктно-орієнтовану технологію.

Мова C++ – це ключ до сучасного об'єктно-орієнтованого програмування. Вона створена для розробки високопродуктивного програмного забезпечення і надзвичайно популярна серед програмістів. Сьогодні бути професійним програмістом високого класу означає бути компетентним в C++.

Ця мова не просто популярна. Вона забезпечує концептуальний фундамент, на який спираються інші мови програмування і багато сучасних засобів обробки даних. Адже не випадково нащадками C++ стали такі шановані мови, як Java і C#.

Основним поняттям C# є об'єктно-орієнтоване програмування. Методика ООП невіддільна від C#, і тому всі програми на C# є об'єктно-орієнтованими хоча б в найменшій мірі. C# є мовою з Сі-подібним синтаксисом.

У зв'язку з цим дуже важливо і корисно засвоїти основні принципи ООП, перш ніж приступати до написання найпростішої програми на C#.

Коли говорять C#, часто мають на увазі технології платформи .NET. І навпаки, коли говорять .NET, нерідко мають на увазі C#. Однак, хоча ці поняття пов'язані, ототожнювати їх не так. Мова C# була створена спеціально для роботи з фреймворком .NET, проте саме поняття .NET дещо ширше.

Мова програмування Java, як і C# вже була народжена як об'єктно-орієнтована, тому всю програму можна уявити як набір взаємодіючих між собою класів і об'єктів. На сьогоднішній день мова Java є однією з найпоширеніших та найпопулярніших мов програмування. Java теж є мовою з Сі-подібним синтаксисом і близька в цьому відношенні до C/C++ та C#.

Ще одною із популярних мов програмування, що підтримує усі основні концепції ООП, є мова Python. Python є інтерпретуючою мовою, вона працюватиме скрізь, де встановлений інтерпретатор. Фактично вона є універсальною кросплатформеною мовою, тому програми на Python працюватимуть на більшості відомих платформ, таких як Windows, Linux, MacOS. Python - дуже проста мову програмування, вона має короткий і в той же час досить простий і зрозумілий синтаксис. Це одна з причин, через яку вона є однією з найпопулярніших мов програмування саме для навчання. Однак, як і для більшості інтерпретованих мов, додатки, створені на Python, працюють повільніше, ніж, наприклад, ті, що створені на C++.

У цьому посібнику ми зосередимося на вивченні об'єктно-орієнтованої технології програмування на базі двох мов: C++ і C#. У першій частині посібника розглянути базові принципи ООП і об’єктно-орієнтована технологія програмування на мові C++, у другій частині – на мові C#.

Кращий спосіб вивчення будь-якої мови програмування − це програмувати на ній. Тому, прочитавши черговий розділ, закріпіть матеріал на практиці. Перш ніж переходити до наступного розділу, переконайтеся в тому, що ви розумієте, чому приклади програм роблять те, що вони роблять. Корисно також експериментувати з програмами, змінюючи один чи два параметра і аналізуючи вплив цих змін на результати. Чим більше ви будете програмувати, тим вище буде ваш рівень як програміста.

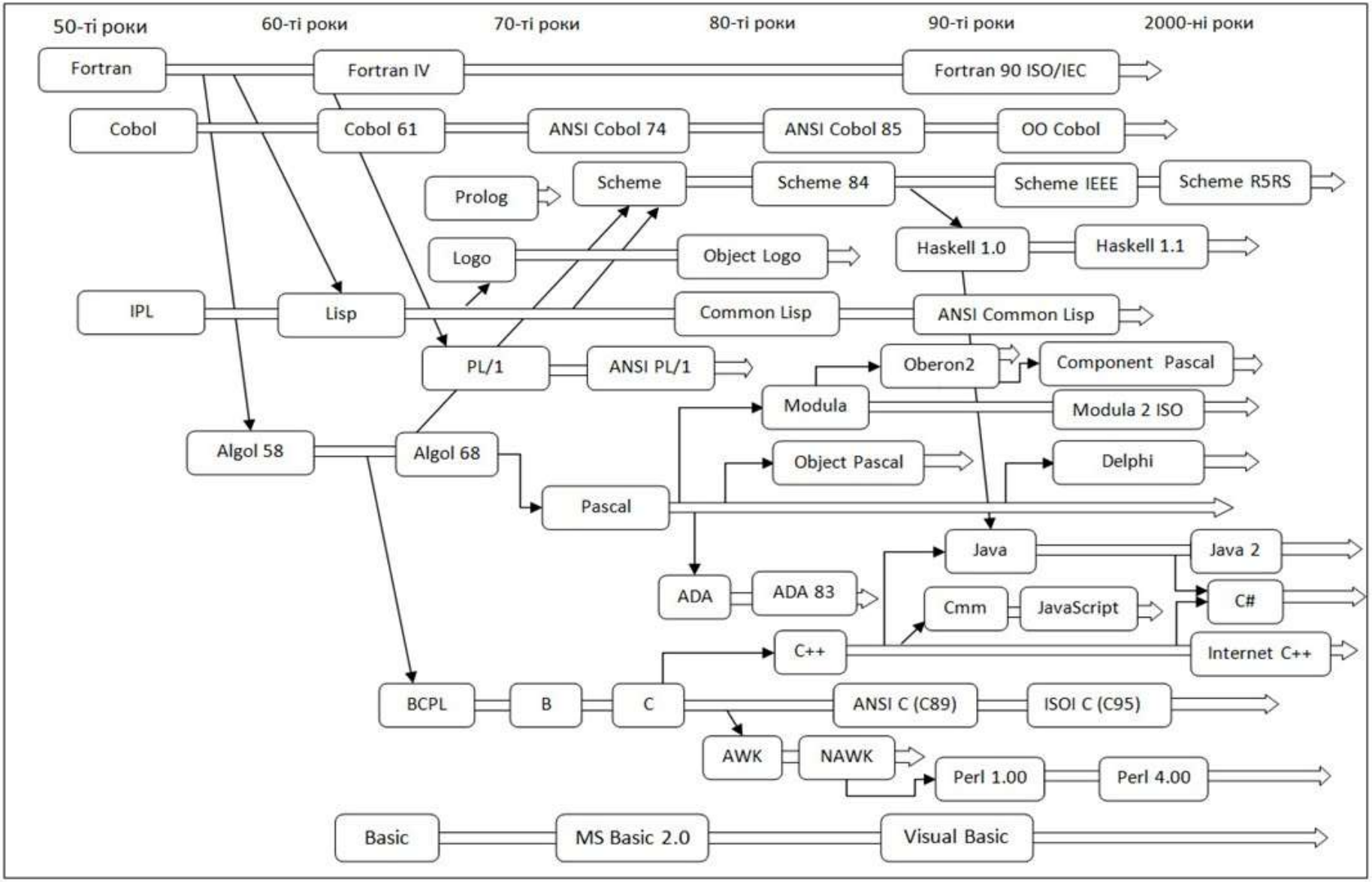


Рис. 1.2. Генеалогічне дерево мов програмування.

1.3 Загальні ознаки об’єктно-орієнтованого програмування

Об'єктно-орієнтоване програмування об'єднало кращі ідеї структурованого з рядом потужних концепцій, які сприяють більш ефективної організації програм. Об'єктно-орієнтований підхід до програмування дозволяє розкласти задачу на складові частини таким чином, що кожна складова частина буде являти собою самостійний об'єкт, який містить власні інструкції і дані. При такому підході істотно знижується загальний рівень складності програм, що дозволяє програмісту впоратися з більш складними програмами, ніж раніше (тобто написаними при використанні структурованого програмування).

Всі мови об'єктно-орієнтованого програмування характеризуються трьома загальними ознаками: інкапсуляцією, спадкуванням і поліморфізмом. Розглянемо коротко кожен з них (детально вони будуть описані нижче в цьому посібнику).

1.3.1 Інкапсуляція

Всі програми, як правило, складаються з двох основних елементів: інструкцій (коду) і даних. Код – це частина програми, яка виконує дії, а дані представляють собою інформацію, на яку спрямовані ці дії. Інкапсуляція – це такий механізм програмування, який пов'язує воєдино код і дані, які він обробляє, щоб убезпечити їх як від зовнішнього втручання, так і від неправильного використання.

В об'єктно-орієнтованої мови код і дані можуть бути пов'язані способом, при якому створюється самостійний чорний ящик. У цьому «ящику» містяться всі необхідні (для забезпечення самостійності) дані і код. При такому зв'язуванні коду і даних створюється об'єкт, тобто об'єкт – це конструкція, яка підтримує інкапсуляцію.

Усередині об'єкту, код, дані або обидві ці складові можуть бути закритими в «рамках» цього об'єкта або відкритими. Закритий код (або дані) відомий і доступний тільки іншим частинам того ж об'єкта. Іншими словами, до закритого коду або даних не може отримати доступ та частина програми, яка існує поза цим об'єктом. Відкритий код (або дані) доступний будь-яким іншим частинам програми, навіть якщо вони визначені в інших об'єктах. Зазвичай відкриті частини об'єкта використовуються для надання керованого інтерфейсу з закритими елементами об'єкта (рис. 1.3).

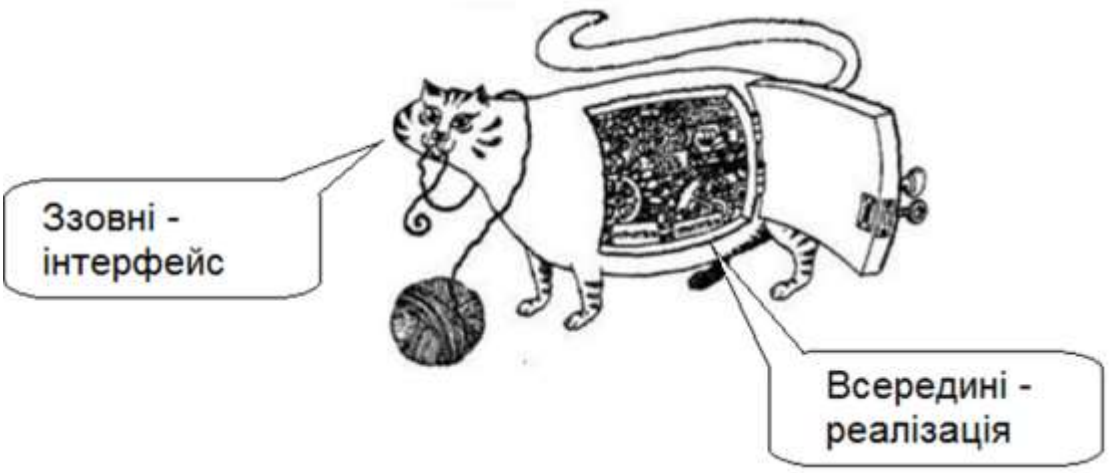


Рис. 1.3. Приховуються деталі реалізації об’єкта, залишаючи назовні лише взаємодію через інтерфейс.

1.3.2 Спадкування

Спадкування – це процес, завдяки якому один об'єкт може набувати властивості іншого. Наприклад, кіт Томас є частиною сімейства котячих, яке в свою чергу є частиною класу ссавці, а той – частиною ще більшого класу тварини. Таким чином, клас тварини має певні якості, які можна застосувати й до підкласу ссавці. Крім цих якостей, клас ссавці має специфічні характеристики, які відрізняють їх від інших тварин. У класі сімейство котячих визначаються якості, специфічні для всіх котячих. Клас кіт успадковує якості всіх попередніх класів і при цьому визначає якості, які є унікальними для цього представника тваринного світу.

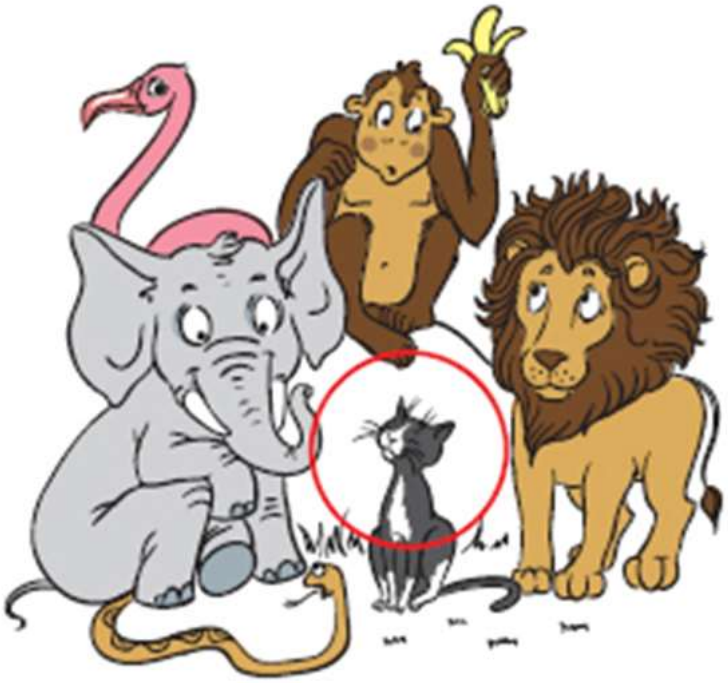


Рис. 1.4. Один об'єкт представляє конкретний екземпляр більш загального класу.

Якщо не використовувати ієрархічне представлення ознак, для кожного об'єкта довелося б в явній формі визначити всі притаманні йому характеристики. Але завдяки наслідуванню об'єкту потрібно визначити тільки ті якості, які роблять його унікальним всередині його класу, оскільки він (об'єкт) успадковує загальні атрибути свого батька. Отже, саме механізм спадкування дозволяє одному об'єкту представляти конкретний екземпляр більш загального класу (рис. 1.4).

1.3.3 Поліморфізм

Поліморфізм (від грецького слова πολυμορφισμός, що означає «багато форм») – це властивість, що дозволяє використовувати один інтерфейс для цілого класу дій. Конкретна дія визначається характерними ознаками ситуації. Як простий приклад поліморфізму можна привести кермо автомобіля, катера і машини на повітряній подушці. Для керма (тобто інтерфейсу) байдуже, який тип рульового механізму використовується в транспортному засобі. Іншим словами, кермо працює однаково. Якщо ви знаєте, як поводитися з кермом, ви зможете вести транспортний засіб будь-якого типу (рис. 1.5).

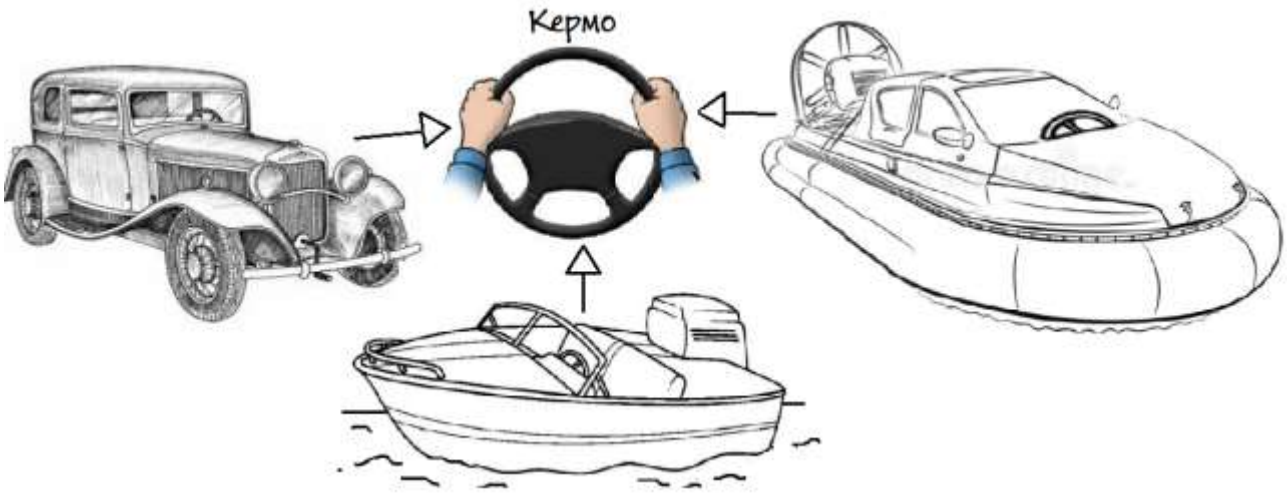


Рис. 1.5. Для різних методів можна використовувати один узагальнений інтерфейс (кермо).

Ще один приклад використання узагальненого інтерфейсу: користувач комп'ютера майже автоматично використовує комбінацію клавіш Ctrl+C і Ctrl+V для копіювання і вставки. Можна казати, що це для нього звичний інтерфейс. Він використовує цю комбінацію клавіш як для копіювання і вставки тексту при роботі з текстовим редактором так і для копіювання і вставки файлів. І хоча в цьому випадку використовується один і той самий прийом (інтерфейс), його реалізація для першого і другого випадку різна.

Той же принцип можна застосувати до програмування. Розглянемо, наприклад, стек, або список, додавання і видалення елементів до якого здійснюється за принципом «останнім прибув – першим обслужений». У вас може бути програма, в якій використовуються три різних типи стека. Один стек призначений для цілочисельних значень, другий – для значень з плаваючою

точкою і третій – для символів. Алгоритм реалізації всіх стеків – один і той же, незважаючи на те, що в них зберігаються дані різних типів. Таким чином, знаючи, як використовувати один стек, ви можете використовувати всі інші.

У більш загальному вигляді концепція поліморфізму виражається фразою «один інтерфейс – багато методів». Це означає, що для групи пов'язаних дій можна використовувати один узагальнений інтерфейс. Поліморфізм дозволяє знизити рівень складності за рахунок можливості застосування одного і того ж інтерфейсу для завдання цілого класу дій.

Резюме

1. У структурному програмуванні під декомпозицією розумілося розділення алгоритмів, де кожен алгоритм виконував один з етапів загального процесу.
2. Об'єктно-орієнтований підхід до програмування дозволяє розкласти завдання на складові частини таким чином, що кожна складова частина буде являти собою самостійний об'єкт, який містить власні інструкції і дані.
3. Об'єктно-орієнтована декомпозиція передбачає, що кожний об’єкт має власну поведінку, та кожний об’єкт моделює певний об’єкт реального світу.
4. Абстракція фокусує увагу на суттєві характеристики об'єкта.
5. Формування ієрархії об'єктів значно спрощує розуміння складної системи.
6. Всі мови об'єктно-орієнтованого програмування характеризуються трьома загальними ознаками: інкапсуляцією, спадкуванням і поліморфізмом.

Завдання до виконання

1. В об'єктно-орієнтованої ієрархії успадкування кожен наступний рівень є більш спеціалізованою формою попереднього. Наведіть приклад ієрархії з повсякденного життя з цією властивістю. Деякі ієрархії не є ієрархіями успадкування. Вкажіть приклад ієрархії без успадкування з реального життя.
2. Є кілька різних сутностей: студент, людина, викладач, староста, доцент. Побудуйте ієрархію цих сутностей так, щоб на чолі ієрархії була найбільш загальна сутність, а в низу – найбільш часткова.
3. Є різні побутові прилади, які оточують вас у повсякденному житті: телевізор, комп'ютер, мобільний телефон, кухонний комбайн. У кожному з них розробники використовували принцип інкапсуляції. Поясніть, у чому він проявляється для всіх перелічених приладів.
4. Наведіть приклади поліморфізму з повсякденного життя.
5. Сформулюйте принципи об'єктно-орієнтованого програмування.
6. Визначте основні відмінності структурного і об'єктно-орієнтованого підходу в програмуванні.

Тести для закріплення матеріалу

1. Що означає принцип інкапсуляції у об'єктно-орієнтованому програмуванні?

а) розміщення файлів класу і головної функції в одному проекті;

б) можливість спадкування елементів базового класу;

в) об'єднання даних з функціями їх обробки разом з приховуванням інформації, яка не потрібна для використання цих даних;

2. Механізм спадкування об’єкта дозволяє: (вкажіть всі правильні відповіді)

а) визначити всі притаманні йому характеристики;

б) визначити тільки ті якості, які роблять його унікальним всередині його класу;

в) використовувати один інтерфейс для багатьох методів;

г) дозволяє одному об'єкту представляти конкретний екземпляр більш загального класу.

3. Що з перерахованого є прикладом поліморфізму?

а) наявність в класі декількох деструкторів;

б) використання одного інтерфейсу для багатьох методів;

в) створення декількох об'єктів класу;

г) всі відповіді невірні.

4. Які ознаки об’єктно-орієнтованої декомпозиції?

а) кожний об’єкт має власну поведінку, та кожний об’єкт моделює певний об’єкт реального світу;

б) розділення алгоритмів, де кожен алгоритм виконував один з етапів загального процесу;

в) приховування деталей реалізації методів класу, залишаючи назовні лише взаємодію через інтерфейс.

РОЗДІЛ 2. КЛАСИ І ОБ’ЄКТИ

У цьому розділі ми познайомимося з класом. Клас – це фундамент, на якому побудована підтримка об'єктно-орієнтованого програмування, а також ядро багатьох більш складних програмних засобів. Клас – це базова одиниця інкапсуляції, яка забезпечує механізм створення об'єктів.

2.1 Основи поняття класу

Об'єктно-орієнтоване програмування побудовано на понятті класу.

Почнемо з визначення термінів класу і об'єкту. Клас визначає новий тип даних, який задає формат об'єкта. Клас включає як дані, так і код, призначений для виконання дій над цими даними. Отже, клас пов'язує дані з кодом. У C++ специфікація класу використовується для побудови об'єктів. Об'єкти – це екземпляри класу. По суті, клас є набором планів, які визначають, як будувати об'єкт. Важливо розуміти, що клас – це логічна абстракція, яка реально не існує до тих пір, поки не буде створено об'єкт цього класу, тобто то, що стане фізичним представленням цього класу в пам'яті комп'ютера.

Визначаючи клас, ви оголошуєте дані, які він містить, і код, який виконується над цими даними. Хоча дуже прості класи можуть містити тільки код або тільки дані, більшість реальних класів містять обидва компоненти. У класі дані оголошуються у вигляді змінних, а код оформляється у вигляді функцій. Функції та змінні, складові класу, називаються його членами. Таким чином, змінна, яка оголошена в класі, називається членом даних, а функція, яка оголошена в класі, називається функцією-членом. Іноді замість терміна член даних використовується термін змінна екземпляра (або змінна реалізації).

Оголошення класу починається з ключового слова class.

Оголошення класу синтаксично подібно оголошенню структури. Розглянемо приклад.

Наступний клас визначає тип Cat, який описує представника сімейства котячих. Кожен кіт має ім’я та хвіст певної довжини.

```
// Створення класу Cat
class Cat
{
 string name; // ім'я кота
 int tail; // довжина хвоста кота
public:
 void init();
 void SetName(string name);
 string GetName();
 void SetTail(int tail);
 int GetTail();
};
```

Розглянемо детально оголошення цього класу.

Всі члени класу Cat оголошені в тілі інструкції class. Членами даних класу Cat є змінні name і tail. Крім того, тут визначено кілька функцій-членів: init(), SetName(string name), GetName(), SetTail(int tail) і GetTail().

Будь-який клас може містити як закриті, так і відкриті члени. За замовчуванням всі елементи, які визначені в класі, є закритими (private-члени). Наприклад, змінні name і tail є закритими. Це означає, що до них можуть отримати доступ тільки інші члени класу Cat, ніякі інші частини програми цього зробити не можуть. У цьому полягає один із проявів інкапсуляції: програміст в повній мірі може керувати доступом до певних елементів даних. Закритими можна оголосити і функції (в цьому прикладі таких немає), і тоді їх зможуть викликати тільки інші члени цього класу.

Ключове слово **public** використовується для оголошення відкритих членів класу.

Щоб зробити частини класу відкритими (тобто доступними для інших частин програми), необхідно оголосити їх після ключового слова public. Всі змінні або функції, визначені після модифікатора public, доступні для всіх інших функцій програми. Отже, в класі Cat функції init(), SetName(string name), GetName(), SetTail(int tail) і GetTail() є відкритими. Зазвичай в програмі організується доступ до закритих членів класу через його відкриті функції. Зверніть увагу на те, що після ключового слова public слід двокрапка.

Слід мати на увазі, що об'єкт утворює свого роду зв'язок між кодом і даними. Так, будь-яка функція-член має доступ до закритих елементів класу. Це означає, що функції init(), SetName(string name), GetName(), SetTail(int tail) і GetTail() мають доступ до змінних name і tail. Щоб додати функцію-член в клас, треба визначити її прототип в оголошенні цього класу.

Визначивши клас, можна створити об'єкт цього «класового» типу, використовуючи ім'я класу. Таким чином, ім'я класу стає модифікатором нового типу. Наприклад, при виконанні наступної інструкції створюється два об'єкти Tomas і Barsik типу Cat:

```
Cat Tomas, Barsik;
```

Після створення об'єкт класу матиме власну копію членів даних, які складають клас. Це означає, що кожен з об'єктів Tomas і Barsik буде мати власні окремі копії змінних name і tail. Отже, дані, пов'язані з об'єктом Tomas, відокремлені (ізольовані) від даних, пов'язаних з об'єктом Barsik.

Щоб отримати доступ до відкритого члену класу через об'єкт цього класу, використовується оператор «крапка» (саме так це робиться і при роботі зі структурами). Наприклад, щоб вивести на екран значення name, що належить об'єкту `Tomas`, використовується наступна інструкція:

```
cout << Tomas.GetName();
```

В C++ клас створює новий тип даних, який можна використовувати для створення об'єктів. Зокрема, клас створює логічну конструкцію, яка визначає

відносини між її членами. Оголошуючи змінну класу, ми створюємо об'єкт. Об'єкт характеризується фізичним існуванням і є конкретним екземпляром класу. Кожен об'єкт займає певну область пам'яті. Кожен об'єкт класу має власну копію даних, визначених у цьому класі.

В оголошенні класу Cat містяться прототипи функцій-членів. Оскільки функції-члени забезпечені своїми прототипами у визначенні класу, їх не потрібно поміщати більше ні в яке інше місце програми.

Щоб реалізувати функцію, яка є членом класу, необхідно повідомити компілятору, до якого класу вона належить, назвавши ім'я цієї функції разом з ім'ям класу. Наприклад, ось як можна записати код функції SetName(string name):

```
void Cat::SetName(string name)
{
 this->name = name;
}
```

Оператор дозволу області видимості кваліфікує ім'я члена разом з ім'ям його класу.

Оператор "::" називається оператором дозволу області видимості. По суті, він повідомляє компілятору, що дана версія функції SetName(string name) належить класу Cat. Іншими словами, оператор "::" заявляє про те, що функція SetName (string name) знаходиться в області видимості класу Cat.

Різні класи можуть використовувати однакові імена функцій. Компілятор ж визначить, до якого класу належить функція, за допомогою оператора дозволу області видимості і імені класу.

Функції-члени можна викликати тільки із заданого об'єкта. Щоб викликати функцію-член з частини програми, яка знаходиться поза класом, необхідно використовувати ім'я об'єкта і оператор "крапка". Наприклад, при виконанні цього коду буде викликана функція init () для об'єкта obj.

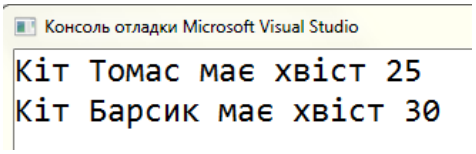
```
Cat obj;
obj.init();
```

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

У наведеній нижче програмі клас Cat ілюструється повністю (для цього об'єднані всі вже знайомі вам частини коду і додані відсутні деталі).

```
    // Створення класу Cat
    class Cat
    {
     string name; // ім'я кота
     int tail; // довжина хвоста кота
    public:
     void init();
     void SetName(string name);
     string GetName();
     void SetTail(int tail);
     int GetTail();
    };

    void Cat::init()
    {
     name = "Томас";
     tail = 25;
    }
    void Cat::SetName(string name)
    {
     this->name = name;
    }
    string Cat::GetName()
    {
     return name;
    }
    void Cat::SetTail(int tail)
    {
     this->tail = tail;
    }
    int Cat::GetTail()
    {
     return tail;
    }

    int main()
    {
     setlocale(0, "ukr");
     Cat Tomas, Barsik;
     Tomas.init();
     cout << "Кіт " << Tomas.GetName() << " має хвіст " <<
Tomas.GetTail() << endl;
     Barsik.SetName("Барсик");
     Barsik.SetTail(30);
     cout << "Кіт " << Barsik.GetName() << " має хвіст " <<
Barsik.GetTail() << endl;
    }
```

При виконанні ця програма генерує такі результати:



2.2 Доступ до членів класу

Тепер розглянемо, як здійснюється доступ до членів класу Cat. Члени класу змінні name і tail оголошені без ключового слова **private**, а це означає, що ці змінні є прихованими, бо доступ **private** визначається за замовчуванням. Це і є реалізація принципу інкапсуляції. Закриті члени класу name і tail доступні тільки іншим членам того ж класу. До них не може отримати доступ та частина програми, яка існує поза цим класом. Яким чином можна отримати доступ до закритих членів? Принцип інкапсуляції передбачає для цього використання відкритого інтерфейсу, тобто відкритих методів (функцій), які реалізують свою власну внутрішню логіку взаємодії із закритими членами.

Для доступу до змінних name і tail створені функції, що дозволяють встановлювати і отримувати значення змінних name: SetName, GetName, SetTail(int tail) і GetTail(). Прийнято називати їх, використовуючи імена, які починаються відповідно на «Set» і «Get». На програмістському жаргоні їх ще називають сеттери і ґеттери (setters і getters). Як видно з приведеного коду, ці функції досить прості, але зверніть увагу на те, як описується функція SetName:

```
void Student::SetName(string name)
{
 this->name = name;
}
```

Тут в якості параметра використовується змінна на ім’я name, але таке ж саме ім'я має і відповідний член даних. Цю невизначеність, коли і ліворуч і праворуч від оператора присвоювання стоять різні змінні з однаковими іменами (ліворуч – назва члена даних, а праворуч – назва параметра) розв’язують за допомогою ключового слова this, це неявно визначений вказівник на поточний об'єкт класу. Зверніть увагу, що для доступу до членів класу через вказівник використовується оператор "->".

Коли доступ до деякого члену класу відбувається ззовні цього класу, його необхідно кваліфікувати (уточнити) за допомогою імені конкретного об'єкта.

Створюються об’єкти (екземпляри класу Cat) на ім’я Tomas і Barsik:

```
 Cat Tomas, Barsik;
```

Виклик відкритої функції SetName ззовні класу Cat здійснюється за допомогою імені об'єкта Barsik:

```
Barsik.SetName("Барсик");
```

2.3 Конструктори і деструктори

Конструктор – це функція, яка викликається при створенні об'єкта.

Як правило, деяку частину об'єкта, перш ніж його можна буде використовувати, необхідно ініціалізувати. Наприклад, розглянемо клас Cat (він представлений вище в цьому розділі). Перш ніж клас Cat можна буде застосовувати, змінним на ім’я name і tail потрібно присвоїти початкові значення. В даному конкретному випадку це виконується за допомогою функції init(). Крім того, для визначення змінних name і tail використовуються функції SetName і SetTail. Але, оскільки вимога ініціалізації членів класу вельми поширена, в C++ передбачена реалізація цієї можливості при створенні об'єктів класу. Така автоматична ініціалізація виконується завдяки використанню конструктора.

Конструктор – це спеціальна функція, яка є членом класу і ім'я якої збігається з ім'ям класу. Ось, наприклад, як став виглядати клас Cat після переробки, пов'язаної із застосуванням конструктора для його ініціалізації:

```
// Створення класу Cat
class Cat
{
 string name; // ім'я кота
 int tail; // довжина хвоста кота
public:
 Cat();  // конструктор
 void SetName(string name);
 string GetName();
 void SetTail(int tail);
 int GetTail();
};
```

Зверніть увагу на те, що в оголошенні конструктора Cat() відсутній тип значення, що повертається. Конструктори ніколи не повертають значень і, отже, немає сенсу у вказуванні їх типу. (При цьому не можна вказувати навіть тип void.)

Тепер наведемо код функції Cat ().

```
// Визначення конструктора
Cat::Cat()
{
 name = "Томас";
 tail = 25;
 cout << "З'явився об'єкт класу Cat на ім'я " << name << endl;
}
```

Конструктор об'єкту викликається при створенні об'єкта. Це означає, що він викликається при виконанні інструкції оголошення об'єкта. Конструктори

глобальних об'єктів викликаються на самому початку виконання програми, ще до звернення до функції main(). Що стосується локальних об'єктів, то їх конструктори викликаються кожен раз, коли зустрічається оголошення такого об'єкта.

Якщо клас немає конструкторів, то компілятор C++ автоматично згенерує для цього класу відкритий конструктор за замовчуванням (default constructor). Його іноді називають неявним конструктором (або "неявно згенерованим конструктором"). Так, у прикладі, де клас Cat не має конструктора, компілятор згенерує наступний конструктор:

```
public:
Cat()
{
}
```

Цей конструктор дозволяє створювати об'єкти класу, але не виконує їх ініціалізацію та не надає значення членам класу. Якщо клас має інші конструктори, то конструктор за замовчуванням створюватися не буде.

Деструктор – це функція, яка викликається при руйнуванні об'єкта.

У багатьох випадках при руйнуванні об'єкта необхідно виконати деяку дію або навіть деяку послідовність дій. Локальні об'єкти створюються при вході в блок, в якому вони визначені, і руйнуються при виході з нього. Глобальні об'єкти руйнуються при завершенні програми. Існує безліч факторів, що обумовлюють необхідність знищення об’єкта. Наприклад, об'єкт повинен звільнити раніше виділену для нього пам'ять. У C++ саме деструкторам доручається обробка процесу дезактівізация об'єкта. Ім'я деструктора збігається з ім'ям конструктора, але передує символом "~" Подібно конструкторам деструктори не повертають значень, отже, в їх оголошеннях відсутня тип значення. Розглянемо вже знайомий нам клас Cat, але тепер він містить конструктор і деструктор.

```
// Створення класу Cat
class Cat
{
 string name; // ім'я кота
 int tail; // довжина хвоста кота
public:
 Cat();  // конструктор
 ~Cat(); // деструктор
 void SetName(string name);
 string GetName();
 void SetTail(int tail);
 int GetTail();
};
```

Ось як виглядає нова версія програми, в якій демонструється використання конструктора і деструктора.

```
// Визначення конструктора
Cat::Cat()
{
 name = "Томас";
 tail = 25;
 cout << "З'явився об'єкт класу Cat на ім'я " << name << endl;
}
// Визначення деструктора
Cat::~Cat()
{
 cout << "Знищений об'єкт класу Cat на ім'я " << name << endl;
}

void Cat::SetName(string name)
{
 this->name = name;
}
string Cat::GetName()
{
 return name;
}
void Cat::SetTail(int tail)
{
 this->tail = tail;
}
int Cat::GetTail()
{
 return tail;
}

int main()
{
 setlocale(0, "ukr");
 Cat Tomas, Barsik;
 Tomas.init();
 cout << "Кіт " << Tomas.GetName() << " має хвіст ";
 cout << Tomas.GetTail() << endl;
 Barsik.SetName("Барсик");
 Barsik.SetTail(30);
 cout << "Кіт " << Barsik.GetName() << " має хвіст ";
 cout << Barsik.GetTail() << endl;
}
```

Вивід повідомлення конструктором

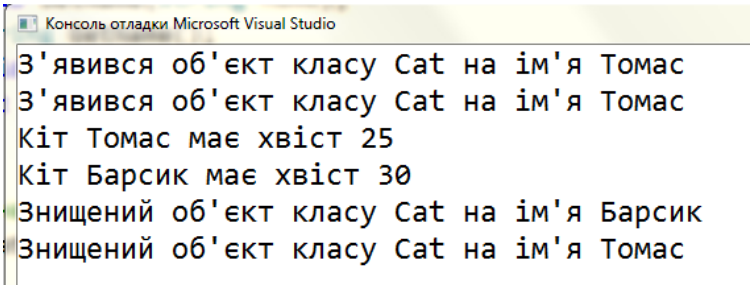
```
cout << "З'явився об'єкт класу Cat на ім'я " << name << endl;
```

і вивід повідомлення деструктором

```
cout << "Знищений об'єкт класу Cat на ім'я " << name << endl;
```

виконується суто у навчальних цілях.

При виконанні цієї програми виходять такі результати:



Рекомендується завжди створювати принаймні один конструктор у класі. Це дозволить вам контролювати процес створення об'єктів вашого класу, і запобігатиме виникненню потенційних проблем після додавання інших конструкторів.

2.4 Параметризовані конструктори

Конструктор може мати параметри. З їх допомогою при створенні об'єкта членам даних (змінним класу) можна привласнити деякі початкові значення, які визначаються в програмі. Це реалізується шляхом передачі аргументів конструктору об'єкта. У наступному прикладі ми вдосконалюємо клас Cat так, щоб він приймав аргументи, які будуть визначати значення змінних name і tail. Перш за все необхідно внести зміни в визначення класу Cat. Тепер воно виглядає так:

```
// Створення класу Cat
class Cat
{
 string name; // ім'я кота
 int tail; // довжина хвоста кота
public:
 Cat(string name, int tail); // параметризований конструктор
 ~Cat(); // деструктор (завжди без параметрів)
 void SetName(string name);
 string GetName();
 void SetTail(int tail);
 int GetTail();
};
```

Запам’ятайте, що деструктор завжди немає жодних параметрів.
Конструктор класу Cat виглядає тепер таким чином:

```
// Визначення конструктора
Cat::Cat(string name, int tail)
{
 this->name = name;
 this->tail = tail;
 cout << "З'явився об'єкт класу Cat на ім'я " << name << endl;
}
```

Щоб передати аргумент конструктору, необхідно зв’язати цей аргумент з об'єктом при оголошенні об'єкту. C++ підтримує два способи реалізації такого зв'язування. Ось як виглядає перший спосіб:

```
Cat Tomas = Cat("Томас", 25);
```

Але ця форма (в такому контексті) використовується рідко, оскільки другий спосіб має більш короткий запис і зручніше для використання.

У другому способі аргумент повинен слідувати за ім'ям об'єкта і укладатися у круглі дужки. Наприклад, така інструкція еквівалентна попередньому оголошенню:

```
Cat Tomas("Томас", 25);
```

Це найпоширеніший спосіб оголошення параметризованих об'єктів. Спираючись на цей метод, наведемо загальний формат передачі аргументів конструкторам:

тіп_класу ім'я_змінної (список_аргументів);

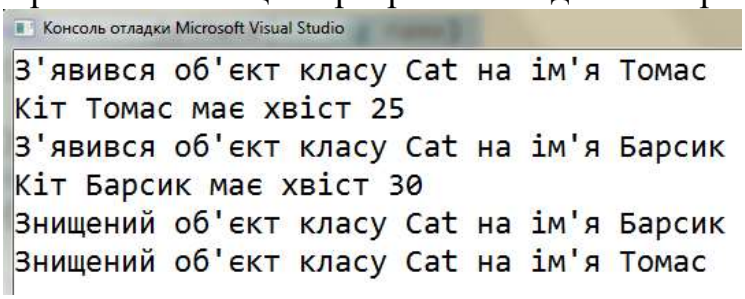
Тут список_аргументів – це список розділених комами параметрів, переданих конструктору.

У наступній версії програми демонструється використання параметризованого конструктора.

```
// Визначення конструктора
Cat::Cat(string name, int tail) // параметризований конструктор
{
 this->name = name;
 this->tail = tail;
 cout << "З'явився об'єкт класу Cat на ім'я " << name << endl;
}

// Визначення деструктора (завжди без параметрів)
Cat::~Cat()
{
 cout << "Знищений об'єкт класу Cat на ім'я " << name << endl;
}
void Cat::SetName(string name)
{
 this->name = name;
}
string Cat::GetName()
{
 return name;
}
void Cat::SetTail(int tail)
{
 this->tail = tail;
}
int Cat::GetTail()
```

```
{
 return tail;
}

int main()
{
 setlocale(0, "ukr");
 Cat Tomas("Томас", 25);
 cout << "Кіт " << Tomas.GetName() << " має хвіст ";
 cout << Tomas.GetTail() << endl;
 Cat Barsik("Барсик", 30);
 cout << "Кіт " << Barsik.GetName() << " має хвіст ";
 cout << Barsik.GetTail() << endl;
}
```

При виконанні цієї програми виходять такі результати:



Альтернативний варіант ініціалізації об'єкта.

Якщо конструктор приймає тільки один параметр, можна використовувати альтернативний спосіб ініціалізації членів об'єкта. Розглянемо наступну програму.

```
class Pryklad {
 int data;
public:
 Pryklad(int param);
 int GetData();
};
Pryklad::Pryklad(int x)
{
 data = x;
}
int Pryklad::GetData()
{
 return data;
}
int main()
{
 Pryklad ob = 4; //виклик функції Pryklad(4)
 cout << ob.GetData();
}
```

Тут конструктор для об'єктів класу Pryklad приймає тільки один параметр. Зверніть увагу на те, як у функції main() оголошується об'єкт ob. Для цього використовується такий формат оголошення:

```
Pryklad ob = 4;
```

У цій формі ініціалізації об'єкта число 4 автоматично передається параметру `data` при виклику конструктора Pryklad(4). Іншими словами, ця інструкція оголошення обробляється компілятором так, якщо б вона була записана в такий спосіб.

```
Pryklad ob = Pryklad(4);
```

2.5 Вбудовані функції

Вбудованою називається функція, 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, щоб створити вбудовану функцію f(), яка повертає int-значення і не приймає жодного параметра, досить оголосити її таким чином.

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

Якщо ж вбудована функція виявляється не такою вже маленькою, загальний розмір програми може істотно збільшитися. Тому найкраще в якості вбудованих використовувати тільки дуже маленькі функції, а ті, що побільше, - оформляти у вигляді звичайних.

Продемонструємо використання вбудованої функції на прикладі наступної програми.

```
class cl {
int i;
 public:
 int get_i();
 void put_i(int j);
 };

 inline int cl::get_i()
 {
 return i;
 }

 inline void cl::put_i(int j)
 {
 i = j;
```

```
}

int main()
{
cl s;
s.put_i(10);
cout << s.get_i();
}
```

Тут замість виклику функцій get_i () і put_i () підставляється їх код.

Важливо розуміти, що в дійсності використання модифікатора inline є запитом, а не командою, по якій компілятор згенерує вбудований (inline) код. Існують різні ситуації, які можуть не дозволити компілятору задовольнити наш запит. Ось кілька прикладів.

– Деякі компілятори не генерують вбудований код, якщо відповідна функція містить цикл, конструкцію switch або інструкцію goto.

– Найчастіше вбудованими не можуть бути рекурсивні функції.

– Як правило, вбудовування «не проходить» для функцій, які містять статичні (static) змінні.

2.6 Використання вбудованих функцій у визначенні класу

Існує ще один спосіб створення вбудованої функції. Він полягає у визначенні коду для функції-члена класу в самому оголошенні класу. Будь-яка функція, яка визначається в оголошенні класу, автоматично стає вбудованою. В цьому випадку необов'язково випереджати її оголошення ключовим словом inline. Наприклад, попередню програму можна переписати в такому вигляді.

```
class cl {
int i;
public:
// автоматично вбудовані функції
int get_i() { return i; }
void put_i(int j) { i = j; }
};

int main()
{
cl s;
s.put_i(10);
cout << s.get_i();
return 0;
}
```

Визначення невеликих функцій-членів в оголошенні класу – звичайна практика в C++ програмуванні. І справа навіть не в засобі автоматичного вбудовування, а просто в зручності. Навряд чи ви зустрінете в професійних програмах, щоб короткі функції-члени визначалися поза класом.

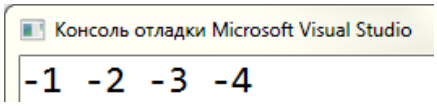
2.7 Масиви об'єктів

Масиви об'єктів можна створювати точно так же, як створюються масиви значень інших типів.

Якщо клас включає параметризований конструктор, то масив об'єктів такого класу можна ініціалізувати. Наприклад, в наступній програмі використовується параметризований клас samp і ініціалізуємий масив sampArray об'єктів цього класу.

```
// Ініціалізація масиву об'єктів.
class Pryklad {
 int data;
public:
 Pryklad(int x)
 {
       data = x;
 }
 int GetData()
 {
       return data;
 }
};

int main()
{
 Pryklad PArray[4] = { -1, -2, -3, -4 };
 for(int i=0; i<4; i++)
    cout << PArray[i].GetData()<<" ";
}
```

Результат виконання цієї програми підтверджує, що конструктору Pryklad дійсно були передані значення від -1 до -4:



Насправді синтаксис ініціалізації масиву, виражений рядком

```
Pryklad PArray[4] = { -1, -2, -3, -4 };
```

являє собою скорочений варіант наступного (більш довгого) формату:

```
Pryklad PArray[4] = { Pryklad(-1), Pryklad(-2), Pryklad(-3),
Pryklad(-4) };
```

Формат ініціалізації, представлений в програмі, використовується програмістами частіше, ніж його довший еквівалент, проте слід пам'ятати, що він працює для масивів таких об'єктів, конструктори яких мають тільки один

аргумент. При ініціалізації масиву об'єктів, конструктори яких мають кілька аргументів, необхідно використовувати більш довгий формат ініціалізації.

Покажемо створення масиву відомих нам об’єктів класу Cat, для чого використовуємо трохи перероблений клас із вбудованими функціями:

```
// Створення класу Cat
class Cat
{
 string name; // ім'я кота
 int tail; // довжина хвоста кота
public:
 Cat(string name, int tail)// параметризований конструктор
 {
      this->name = name;
      this->tail = tail;
 }
 ~Cat() {} // деструктор (завжди без параметрів)
 void  PrintInfo()
 {
 cout << "Кіт "<<name << " має хвіст " << tail << " см." << endl;
 }
};
int main()
{
 Cat C[3] = { Cat("Томас", 25), Cat("Барсик", 30), Cat("Лелик", 35)}
     for(int i=0; i<3; i++)
       C[i].PrintInfo();
}
```

Результат виконання цієї програми:

```
Консоль отладки Microsoft Visual Studio
Кіт Томас має хвіст 25 см.
Кіт Барсик має хвіст 30 см.
Кіт Лелик має хвіст 35 см.
```

2.8 Вказівники на об'єкти

Щоб отримати доступ до окремого члена об'єкта виключно «силами» самого об'єкта, використовується оператор «крапка». А якщо для цього служить вказівник на цей об'єкт, необхідно використовувати оператор «стрілка» (->). (Застосування операторів «крапка» і «стрілка» для об'єктів відповідає їх застосуванню для структур та об'єднань.)

Щоб оголосити вказівник на об'єкт, використовується той же синтаксис, як і в разі оголошення вказівників на значення інших типів. У наступній програмі створюється простий клас P_example, визначається об'єкт цього класу ob і оголошується вказівник на об'єкт типу P_example з ім'ям p. Цей приклад демонструє, як можна безпосередньо отримати доступ до об'єкта ob і як

використовувати для цього вказівник (в цьому випадку ми маємо справу з непрямим доступом).

```
// Простий приклад використання вказівника на об'єкт.
class P_example {
int num;
public:
void set_num(int val) {num = val;}
void show_num();
};
void P_example::show_num()
{
cout << num << " ";
}

int main()
{
P_example ob, *p; // Оголошуємо об'єкт і вказівник на нього.
ob.set_num(1); // Отримуємо прямий доступ до об'єкта ob.
ob.show_num();
p = &ob; // Надаємо вказівнику p адресу об'єкта ob.
p->show_num(); // Отримуємо доступ до об'єкта ob за
               // допомогою вказівника.
}
```

Посилання на об'єкти.

На об'єкти можна посилатися таким же чином, як і на значення будь-якого іншого типу. Для цього не існує ніяких спеціальних інструкцій або обмежень. Але, як буде показано в наступному розділі, використання посилань на об'єкти дозволяє справлятися з деякими специфічними проблемами, які можуть зустрітися при використанні класів.

Резюме

1. Клас – це базова одиниця інкапсуляції, яка забезпечує механізм створення об'єктів. Клас визначає новий тип даних, який задає формат об'єкта.
2. Клас включає як дані, так і код, призначений для виконання над цими даними. Отже, клас пов'язує дані з кодом.
3. Клас – це логічна абстракція, яка реально не існує до тих пір, поки не буде створено об'єкт цього класу, тобто то, що стане фізичним представленням цього класу в пам'яті комп'ютера.
4. Будь-який клас може містити як закриті (private), так і відкриті (public) члени. За замовчуванням всі елементи, які визначені, в класі, є закритими.
5. Оператор "::" називається оператором дозволу області видимості.
6. Конструктор – це спеціальна функція, яка є членом класу і ім'я якої збігається з ім'ям класу. Конструктор викликається при створенні об'єкта.
7. Конструктори не повертають значень і, отже, немає сенсу у вказівці їх типу. (При цьому не можна вказувати навіть тип void.)
8. Якщо клас немає конструкторів, то компілятор C++ автоматично згенерує для цього класу відкритий конструктор за замовчуванням.
9. Конструктор може мати параметри. З їх допомогою при створенні об'єкта членам даних (змінних класу) можна привласнити деякі початкові значення, які визначаються в програмі.
10. Деструктор – це функція, яка викликається при руйнуванні об'єкта. Деструктори ніколи не мають параметрів.

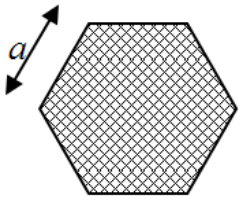
Завдання виконання до лабораторної роботи

1. Визначити користувальницький клас у відповідності з варіантом завдання.
2. Визначити в класі конструктор без параметрів і конструктор з параметрами.
3. Визначити в класі деструктор.
4. Визначити в класі компоненти-функції для перегляду та ініціалізації полів даних.
5. Визначити покажчик на компоненту-функцію.
6. Визначити покажчик на екземпляр класу.
7. Написати демонстраційну програму, в якій створюються і руйнуються об'єкти певного класу і кожен виклик конструктора і деструктора супроводжується видачею відповідного повідомлення. (який об'єкт який конструктор або деструктор викликав).
8. Показати в програмі використання покажчика на об'єкт і покажчика на компоненту функцію.

Варіант 1. Створити клас «людина», такий, що:

1) він містить рядкові змінні «ім'я», «прізвище» «телефон». Змінні цілого типу «ідентифікаційний код», «вік»;
2) його конструктор без параметрів створює екземпляр з тими значеннями, які ви вважаєте за доцільне присвоїти за замовчуванням, а конструктор з параметрами створює екземпляр з відповідними значеннями параметрів.
3) його функції дозволяють отримувати і присвоювати значення відповідних змінних (сеттери і ґеттери);
4) функція print () виводить на екран всю інформацію про людину.

Варіант 2. Створити клас «бджолиний стільник» у вигляді правильного шестикутника:



$$S = \frac{3\sqrt{3}}{2}a^2$$

1) він містить дійсні змінні «сторона» і «площа» (a і s);
2) його конструктор без параметрів створює екземпляр з нульовим значенням сторони a, а конструктор з параметрами створює екземпляр з відповідним значенням параметра a. При створенні об'єкта має бути визначена його площа (змінна s), яка обчислюється за формулою, вказаною вище;
3) його функції дозволяють отримувати і присвоювати значення змінної a (сеттери і ґеттери), обчислювати і отримувати значення площі (ґеттер);
4) функція print () виводить на екран всю інформацію про фігуру «бджолиний стільник».

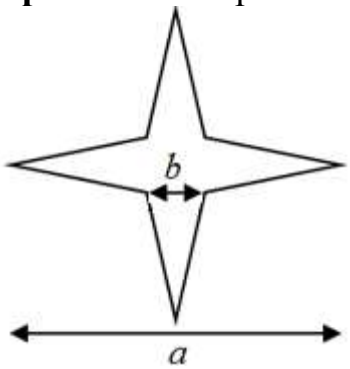
Варіант 3. Створити клас Triangle (трикутник»), що визначається за значеннями трьох його сторін:

1) його екземпляр містить дійсні змінні *a, b, c* зі значеннями трьох сторін трикутника і змінну s – площу;
2) його конструктор без параметрів створює екземпляр з нульовим значенням сторін, а параметризований конструктор створює екземпляр з відповідним значеннями параметрів *a, b, c*. При створенні об'єкта має бути визначена його площа (змінна s), яка обчислюється за формулою Герона:
$s = \sqrt{p(p-a)(p-b)(p-c)}, p = (a+b+c)/2$;
3) його функції дозволяють отримувати і присвоювати значення сторін (сеттери і ґеттери), обчислювати отримувати значення площі (ґеттер);
4) функція print() виводить на екран значення його сторін і площі трикутника.

Варіант 4. Створити клас «смайлик», такий, що:

1) його екземпляр містить дійсні змінні для розміру радіуса обличчя R, радіуса r очей і змінну s – площу обличчя, на яку наноситься крем, кількість якого залежить від площі обличчя (зрозуміло, на очі крем не наноситься, тому площа обличчя не містить площі очей);
2) його конструктор без параметрів створює екземпляр зі значенням 0, а конструктор з параметрами створює екземпляр з відповідним значенням R і r;
3) його функції дозволяють отримувати і присвоювати значення R і r і обчислювати значення площі, при цьому необхідно передбачити ситуацію, коли очі поміщаються на обличчя і створити свою власну обробку такої ситуації;
4) функція print() виводить на екран значення її R, r площі (без площі очей).

Варіант 5. Створити клас «зірка»:



1) його екземпляр містить розміри a і b, що показані на рисунку вище і змінну s – площу «зірки»;
2) його конструктор без параметра створює екземпляр зі значеннями a=0 і b=0, а конструктор з параметрами створює екземпляр з відповідним значенням a і b і обчислює площу «зірки» за формулою, яку ви маєте визначити самі, доклавши для цього трохи кмітливості;
3) його функції дозволяють отримувати і присвоювати значення a, b і обчислювати значення площі;
4) функція print() виводить на екран значення a, b і площі цієї фігури.

Тести для закріплення матеріалу

1. При створенні об'єкта класу
   а) створюються нові копії членів-даних і членів-функцій класу;
   б) автоматично викликається конструктор;
   в) підключається бібліотека з файлами класу;
   г) інше.
2. При зверненні до члена класу через ім'я об'єкта використовується операція
   а):
   б) ::
   в) ->
   г).
3. При зверненні до члена класу через вказівник на об'єкт використовується операція
   а):
   б) ::
   в) ->
   г).
4. Що означають елементи опису членів класу private, protected і public?
   а) це директиви елементів класу;
   б) це модифікатори доступу
   в) це базові методи;
   г) інше.
5. За що відповідають модифікатори доступу private і public?
   а) у private оголошуються тільки змінні;
   б) public відповідає тільки за область видимості методів;
   в) private і public відповідають за область видимості зазначених у них елементів класу;
   г) інше.
6. Коли викликається конструктор ?
   а) при створенні об'єкта;
   б) викликається програмістом;
   в) при запуску програми;
   г) інше.

7. Імені деструктора безпосередньо передує символ
    а) ::
    б) ~
    в) &
    г) #
8. Якщо програміст не вказав конструктор, компілятор
    а) видасть помилку;
    б) створить абстрактний клас;
    в) створить клас, але видасть помилку при зверненні до об'єкта;
    г) створить автоматично конструктор за замовчуванням.
9. Скільки може бути в класі конструкторів і деструкторів?
    а) один конструктор і один деструктор;
    б) безліч конструкторів і один деструктор;
    в) безліч конструкторів і безліч деструкторів;
    г) один конструктор і безліч деструкторів.
10. Чи обов’язково в описі класу повинні міститись конструктор і деструктор?
    а) обов’язково один конструктор і один деструктор;
    б) обов’язково безліч конструкторів і один деструктор;
    в) обов’язково безліч конструкторів і безліч деструкторів;
    г) якщо клас не міститись конструктор і деструктор, компілятор створить їх автоматично замовчуванням.

РОЗДІЛ 3. БІЛЬШ ДЕТАЛЬНИЙ РОЗГЛЯД КЛАСІВ

У цьому розділі ми продовжимо розглядати класи. Тут ви познайомитеся з «дружніми» функціями, перевантаженням конструкторів, а також з можливістю передачі і повернення об'єктів функціями. Крім того, ви дізнаєтеся про існування спеціального типу конструктора, іменованого конструктором копії, який використовується в разі, коли виникає необхідність у створенні копії об'єкту. І «дружні» функції і конструктор копії широко застосовуються саме в мові C++. В інших об’єктно-орієнтованих мовах, зокрема у родинних мовах C# і Java, застосовуються інші підходи та технології.

3.1 «Дружні» функції

У C++ існує можливість дозволити доступ до закритих членів класу функціям, які не є членами цього класу. Для цього достатньо оголосити ці функції «дружніми» (або «друзями») стосовно розглянутого класу. Щоб зробити функцію «другом» класу, треба включити її прототип в public-розділ оголошення класу і передувати його ключовим словом **friend**. Наприклад, в цьому фрагменті коду функція frnd() оголошується «другом» класу Cat.

```
class Cat {
// . . .
public:
friend void frnd(Cat obj);
// . . .
};
```

Ключове слово friend надає функції, яка не є членом класу, доступ до його закритих членів.

Розглянемо короткий приклад, в якому функція-«друг» використовується для доступу до закритих членів класу MyClass.

```
// Демонстрація використання функції-"друга".
class MyClass {
int a, b;
public:
MyClass (int i, int j) { a=i; b=j; }
friend int sum(MyClass x); // Функція sum() - "друг" класу MyClass.
};

// Зверніть увагу на те, що функція sum() не є членом жодного класу
int sum(MyClass x)
{
/* Оскільки функція sum() - "друг" класу MyClass,
вона має право на прямий доступ до його членів-даних a й b. */
return x.a + x.b;
}
```

```
int main ()
{
MyClass n(3, 4);
cout << sum(n);  // виводить число 7
}
```

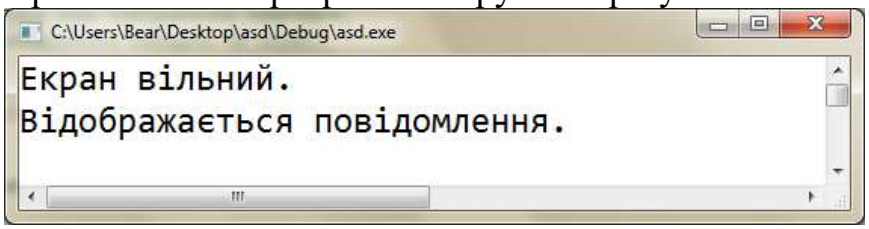
Незважаючи на те що в даному прикладі ми не отримаємо ніякої користі з оголошення функції sum() «другом», а не членом класу MyClass, існують певні обставини, при яких статус «функції-друга» має велике значення. По-перше, «функції-друзі» можуть бути корисні для перевантаження операторів певних типів. По-друге, «функції-друзі» спрощують створення деяких функцій вводу-виводу. Про це мова попереду.

Третя причина використання «функцій-друзів» полягає в тому, що в деяких випадках два (або більше) класи можуть містити члени, які знаходяться у взаємному зв'язку з іншими частинами програми. Наприклад, у нас є два різних класи, які при виникненні 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. У кожному класі можна створити функцію-член, що повертає значення, за яким можна судити про те, чи активно повідомлення чи ні; однак перевірка цієї умови потребує додаткових витрат (тобто двох викликів функцій замість одного). Якщо статус «спливаючого» повідомлення необхідно перевіряти часто, ці додаткові витрати можуть виявитися просто неприйнятними. Однак за допомогою функції, «дружній» для обох класів, можна безпосередньо перевіряти статус кожного об'єкта, викликаючи тільки одну функцію, яка буде мати доступ до обох класів. У подібних ситуаціях функція-«друг» дозволяє написати більш ефективний код. Ця ідея ілюструється на прикладі наступної програми.

```
// Використання функції-"друга".
const int IDLE=0;
const int INUSE=1;

class C2; // випереджальне оголошення

class C1 {
int status; // IDLE якщо повідомлення неактивно,
// INUSE якщо повідомлення виведено на екран.
public:
void set_status(int state);
friend int idle(C1 a, C2 b);
};

class C2 {
int status; // IDLE якщо повідомлення неактивно,
// INUSE якщо повідомлення виведено на екран.
public:
void set_status(int state);
```

```
friend int idle(C1 a, C2 b); // idle – бездіяльність (англ.)
};

void C1::set_status(int state)
{
status = state;
}
// клас C2 визначається після оголошення дружньої функції у класі C1
// тому необхідно попередньо оголосити клас C2
 void C2::set_status(int state)
{
status = state;
}
// Функція idle () - "друг" для класів C1 і C2.
int idle(C1 a, C2 b) // idle – бездіяльність (англ.)
{
if(a.status || b.status) return 0;
else return 1;
}

int main()
{
setlocale(0,"Ukr");
C1 x;
C2 y;
x.set_status(IDLE);
y.set_status(IDLE);
if(idle(x, y)) cout << "Екран вільний."<<endl;
else cout << "Відображається повідомлення."<<endl;
x.set_status(INUSE);
if(idle(x, y)) cout << "Екран вільний."<<endl;
else cout << "Відображається повідомлення."<<endl
}
```

При виконанні програма генерує такі результати:



У випадку, коли функція оголошується «другом» для кількох класів. можна зіткнутися з однією проблемою. Наприклад, «дружня» функція оголошується як у класі A так й у класі B і має в якості параметрів об’єкти класів A і B, а сам клас B визначається після оголошення «дружньої» функції у класі A. При компіляції з'являється повідомлення про помилку в рядку, де ця «дружня» функція оголошується у класі A. Для того, щоб уникнути такої помилки, необхідно попередньо оголосити клас B.

Оскільки функція idle () є «другом» як для класу C1, так і для класу C2, вона має доступ до закритого члену status, визначеному в обох класах. Таким чином, стан об'єкта кожного класу одночасно можна перевірити одним зверненням до функції idle().

Зверніть увагу на те, що в цій програмі використовується випереджальне оголошення для класу C2. Щоб створити випереджальне оголошення для класу, досить використовувати формат, представлений в цій програмі:

```
class C2; // випереджальне оголошення
```

«Друг» одного класу може бути членом іншого класу. Перепишемо попередню програму так, щоб функція idle() стала членом класу C1. Зверніть увагу на використання оператора дозволу області видимості (або оператора дозволу контексту) при оголошенні функції idle() в якості «друга» класу C2.

```
//Функція може бути членом одного класу і одночасно "другом" іншого
const int IDLE=0;
const int INUSE=1;
class C2; // випереджальне оголошення
class C1 {
int status; // IDLE якщо повідомлення неактивно,
// INUSE якщо повідомлення виведено на екран.
public:
void set_status(int state);
int idle(C2 b); //тепер це член класу C1
};

class C2 {
int status; // IDLE якщо повідомлення неактивно,
// INUSE якщо повідомлення виведено на екран.
public:
void set_status(int state);
friend int C1::idle(C2 b); // функція-"друг"
};

void C1::set_status(int state)
{
status = state;
}
 void C2::set_status(int state)
{
status = state;
}
// Функція idle () - член класу C1 і "друг" класу C2.
int C1::idle(C2 b)
{
if(status || b.status) return 0;
else return 1;
}
```

```
int main()
{
setlocale(0,"Ukr");
C1 x;
C2 y;
x.set_status(IDLE);
y.set_status(IDLE);
if(x.idle(y)) cout << "Екран вільний."<<endl;
else cout << "Відображається повідомлення."<<endl;
x.set_status(INUSE);
if(x.idle(y)) cout << "Екран вільний."<<endl;
else cout << "Відображається повідомлення."<<endl;
}
```

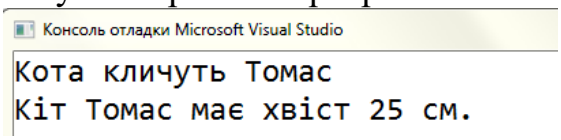
Оскільки функція idle() є членом класу C1, вона має прямий доступ до змінної status об'єктів типу C1. Отже, в якості параметра необхідно передавати функції idle () тільки об'єкти типу C2.

Взагалі, цілий клас A може бути «другом» класа B , тоді всі функії-члени класу A є «друзями» класа B. Розглянемо приклад, в якому один клас є «дружнім» до іншого.

```
class MyFunc; // зазделегідь оголошуємо клас, якиий стане дружнім
class Cat
{
 string name;
 int tail;
public:

 Cat(string Name, int Tail)
 {
      name = Name;
      tail = Tail;
 }
 friend MyFunc;  // дружній клас
};

//Всі функції класу MyFunc мають доступ до всіх членів класу Cat
class MyFunc
{
public:
 void AboutName(Cat cat)
 {
      cout << "Кота кличуть " << cat.name << endl;
 }
 void AboutTail(Cat cat)
 {
      cout << "Кіт " << cat.name << " має хвіст ";
      cout << cat.tail << " см." << endl;
 }
};
```

```
int main()
{
 setlocale(0, "ukr");
 Cat c("Томас", 25);
 MyFunc mf;
 mf.AboutName(c); // виклик функції AboutName
 mf.AboutTail(c); // виклик функції AboutTail
}
```

Результат роботи програми:



3.2 Перевантаження конструкторів

Незважаючи на виконання конструкторами унікальних дій, вони по суті всі є функціями і, як всі функції, можуть перевантажуватися. Щоб перевантажити конструктор класу, досить оголосити його в усіх потрібних форматах і визначити кожну дію, пов'язану з відповідним форматом.

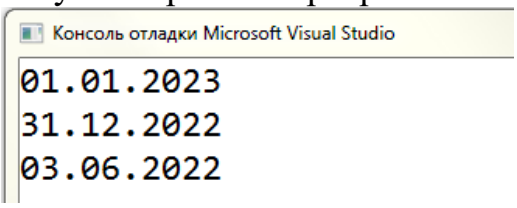
Насправді, якщо клас містить більше одного конструктора, це означає, що в класі є перевантажений конструктор. Клас може містити скільки завгодно конструкторів. Під час створення об'єкта завжди виконується один із них – той, у якого вказані відповідні параметри та їх тип.

Наприклад, в наступній програмі оголошується клас Date, який містить дату (число, місяць, рік) і виводить визначену дату на екран.

```
//Використання перевантажених конструкторів.
#include <iostream>
#include <time.h>
using namespace std;

class Date
{
 int day;
 int month;
 int year;

public:
 Date() // конструктор без параметрів
 {
       day=1;
       month = 1;
       year = 2023;
 }
// конструктор з параметрами число, місяць, рік
 Date(int Day, int Month, int Year)
 {
       day = Day;
```

```
        month = Month;
        year = Year;
 }
// конструктор з параметрамии для поточної дати
 Date(time_t t, tm *Pt)
 {
        day = Pt->tm_mday;
        month = Pt->tm_mon;
        year = 1900 + Pt->tm_year;
 }
 void Print()
 {
        if(day <10 && month<10)
          cout <<"0"<< day << ".0" << month << "." << year << endl;
        if (day < 10 && month >= 10)
              cout << "0" << day << "." << month << "." << year << endl;
        if (day >=10 && month < 10)
              cout << day << ".0" << month << "." << year << endl;
        if (day >= 10 && month >= 10)
              cout << day << "." << month << "." << year << endl;
 }
};

int main()
{
 Date date1;
 Date date2(31, 12, 2022);
 time_t t = time(NULL);
    tm *Pt = localtime(&t);
 Date date3(t, Pt);
 date1.Print();
 date2.Print();
 date3.Print();
}
```

Результат роботи програми:



Механізм перевантаження конструкторів сприяє зниженню рівня складності програм.

Зменшення кількості конструкторів

У прикладі з класом Date конструктор за замовчуванням насправді зайвий. Можна спростити цей клас. Замість конструктора за замовчуванням та

конструктора з параметрами Day, Month, Year можна написати один конструктор наступним чином:

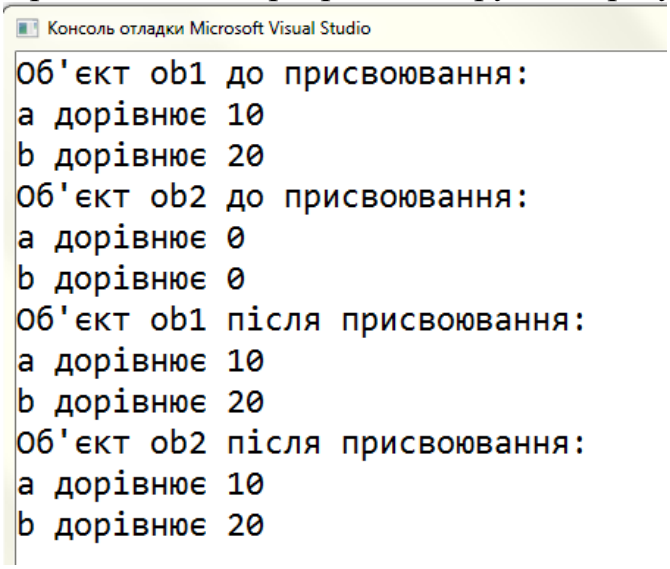
```
Date(int Day=1, int Month=1, int Year=2023)
 {
       day = Day;
       month = Month;
       year = Year;
 }
```

3.3 Присвоювання об'єктів.

Якщо два об'єкти мають однаковий тип (тобто обидва вони – об'єкти одного класу), то один об'єкт можна присвоїти іншому. Для присвоювання недостатньо, щоб два класи були фізично подібні. Імена класів, об'єкти яких беруть участь в операції присвоювання, повинні збігатися. Якщо один об'єкт присвоюється іншому, то за замовчуванням дані першого об'єкта порозрядно копіюються в другій. Присвоювання об'єктів демонструється в наступній програмі:

```
//Демонстрація присвоювання об'єктів.
class MyCl
{
 int a, b;

public:
 MyCl(int i, int j)
 { a = i, b = j; }
 void Print()
 {
       cout << "a дорівнює " << a << ' ' << endl;
       cout << "b дорівнює " << b << ' ' << endl;
 }

};

int main()
{
 setlocale(0, "Ukr");
 MyCl ob1(10, 20), ob2(0, 0);
 cout << "Об'єкт ob1 до присвоювання: " << endl;
 ob1.Print();
 cout << "Об'єкт ob2 до присвоювання: " << endl;
 ob2.Print();
 ob2 = ob1; // присвоювання об'єкта ob1 об'єкту ob2.
 cout << "Об'єкт ob1 після присвоювання: " << endl;
 ob1.Print();
 cout << "Об'єкт ob2 після присвоювання: " << endl;
 ob2.Print();
}
```

При виконанні програма генерує такі результати:



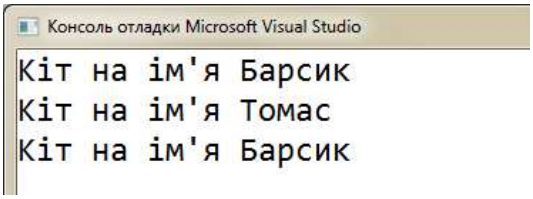
За замовчуванням всі дані з одного об'єкта присвоюються іншому шляхом створення побітової копії. (Іншими словами, створюється точний дублікат об'єкта.) Але, як буде показано нижче, оператор присвоювання можна перевантажити, визначивши власні операції присвоювання.

3.4 Передача об’єктів функціям

Об'єкт можна передати функції точно так же, як значення будь-якого іншого типу даних. Функції передається не сам об'єкт, а його копія. Отже, зміни, внесені в об'єкт при виконанні функції, не мають жодного впливу на об'єкт, який використовується в якості аргументу для функції. Цей механізм демонструється в наступній програмі.

```
class Cat {
 string name;
public:
 void SetName(string Name) { name = Name; }
 void Print() { cout << "Кіт на ім'я " << name<<endl; }
};
void f(Cat cat)
{
 cat.Print(); // Виводить ім'я.
 cat.SetName("Томас"); // Встановлює тільки локальну копію.
 cat.Print();  // Виводить локальну копію Томас.
}
int main()
{
 setlocale(0, "ukr");
 Cat cat;
 cat.SetName("Барсик");
 f(cat);
 cat.Print(); // Виводить ім'я Барсик. Значення не змінилося.
}
```

Ось як виглядають результати виконання цієї програми.



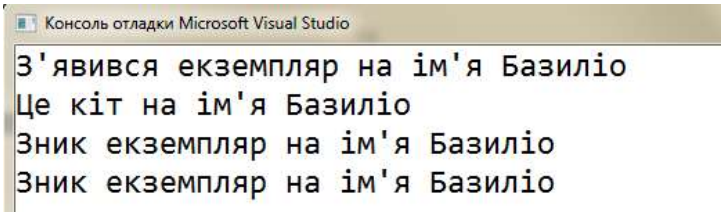
Як підтверджують ці результати, модифікація об'єкта cat в функції f () не впливає на об'єкт cat в функції main().

3.5 Конструктори, деструктори і передача об'єктів

Незважаючи на те що передача функціям нескладних об'єктів в якості аргументів – досить проста процедура, при цьому можуть відбуватися непередбачені події, що мають відношення до конструкторів і деструкторів. Щоб розібратися в цьому, розглянемо наступну програму.

```
class Cat {
 string name;
public:
 Cat(string Name) { name = Name;
      cout << "З'явився екземпляр на ім'я " << name << endl; }
 ~Cat() { cout << "Зник екземпляр на ім'я " << name << endl; }
 void Print() { cout << "Це кіт на ім'я " << name << endl; }
};

void f(Cat cat)
{
 cat.Print(); // Виводить ім'я.
}

int main()
{
 setlocale(0, "ukr");
 Cat cat("Базиліо");
 f(cat);
}
```

При виконанні ця програма виводить наступні несподівані результати.



Як бачите, тут виконується одне звернення до функції конструктора (при створенні об'єкта cat), але чомусь два звернення до функції деструктора. Давайте розбиратися, в чому тут справа.

При передачі об'єкта функції створюється його копія (і ця копія стає параметром в функції). Створення копії означає «народження» нового об'єкта. Коли виконання функції завершується, копія аргументу (тобто параметр) руйнується. Тут виникає відразу два питання. По-перше, чи викликається конструктор об'єкта при створенні копії? По-друге, чи викликається деструктор об'єкта при руйнуванні копії? Відповіді можуть здивувати вас.

Коли при виконанні функції створюється копія аргументу, звичайний конструктор не викликається! Замість цього викликається конструктор копії об'єкта. Конструктор копії визначає, як повинна бути створена копія об'єкта. Але якщо в класі явно не визначений конструктор копії, C++ надає його за замовчуванням. Конструктор копії за замовчуванням створює побітову (тобто ідентичну) копію об'єкта. Оскільки звичайний конструктор використовується для ініціалізації об'єкта, він не повинен викликатися для створення копії вже існуючого об'єкта. Такий виклик змінив би його вміст. При передачі об'єкта функції має сенс використовувати поточний стан об'єкта, а не його початковий стан.

Але коли функція завершується і руйнується копія об'єкта, яка використовується в якості аргументу, викликається деструктор цього об'єкта. Необхідність виклику деструктора пов'язана з виходом об'єкта з області видимості. Саме тому попередня програма мала два звернення до деструктора. Перше відбулося при виході з області видимості параметра функції f(Cat cat), а друге – при руйнуванні об'єкта cat в функції main () по завершенні програми.

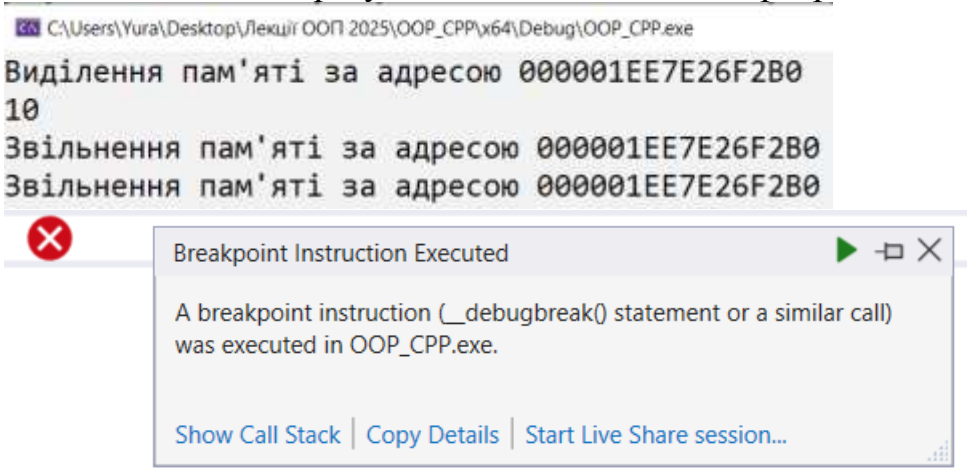
Отже, коли об'єкт передається функції як аргумент, звичайний конструктор не викликається. Замість нього викликається *конструктор копії,* який за замовчуванням створює побітову (ідентичну) копію цього об'єкта. Але коли ця копія руйнується (зазвичай при виході за межі області видимості по завершенні функції), обов'язково викликається деструктор.

3.6 Потенційні проблеми при передачі параметрів

Незважаючи на те що об'єкти передаються функцій «за значенням», тобто за допомогою звичайного механізму передачі параметрів, який теоретично захищає аргумент, тут все-таки можливий побічний ефект або навіть загроза для «життя» об'єкта, який використовується в якості аргументу. Наприклад, якщо об'єкт, який використовується як аргумент, вимагає динамічного виділення пам'яті та звільняє цю пам'ять при руйнуванні, його локальна копія при виклику деструктора звільнить ту ж саму область пам'яті, яка була виділена оригінальному об'єкту. І цей факт стає вже цілою проблемою, оскільки оригінальний об'єкт все ще використовує цю (вже звільнену) область пам'яті. Описана ситуація робить вихідний об'єкт «неповноцінним» і, по суті, непридатним для використання. Розглянемо наступну просту програму.

```
// Демонстрація проблеми, можливої при передачі об'єктів функцій.
class MyCl
{
 int *p;
public:
 MyCl(int i)
 {

      p = new int;
      *p = i;
      cout << "Виділення пам'яті за адресою " << p << endl;
 }

 ~MyCl()
 {
      cout <<  "Звільнення пам'яті за адресою " << p << endl;
      delete p;
 }
 int GetVal() { return *p; }
};

// При виконанні цієї функції і виникає проблема.
void GetInfo(MyCl ob)
{
 cout << ob.GetVal() << endl;
}

int main()
{
 setlocale(0, "Ukr");
 MyCl a(10);
 GetInfo(a);
}
```

Ось як виглядають результати виконання цієї програми:



Ця програма містить принципову помилку. І ось чому: при створенні в функції main() об'єкта a виділяється область пам'яті, адреса якої присвоюється

вказівнику p об’єкта a. При передачі функції GetInfo() об'єкт a копіюється в параметр ob. Це означає, що обидва об'єкти (a і ob) матимуть однакове значення для вказівника p.

Іншими словами, в обох об'єктах (в оригіналі та його копії) член даних p буде вказувати на одну і ту ж динамічно виділену область пам'яті. По завершенні функції GetInfo() об'єкт ob руйнується, і його руйнування супроводжується викликом деструктора. Деструктор звільняє область пам'яті, що адресується вказівником ob.p. Але ж ця (вже звільнена) область пам'яті – та ж сама область, на яку все ще вказує член даних (вихідного об'єкта) a.p! У наявності серйозна помилка.

Насправді справи йдуть ще гірше. По завершенні програми руйнується об'єкт a, і динамічно виділена (ще при його створенні) пам'ять звільняється вдруге. Справа в тому, що звільнення однієї і тієї ж області динамічно виділеної пам'яті вдруге вважається невизначеною операцією, яка, як правило (в залежності від того, як реалізована система динамічного розподілу пам'яті), викликає непоправну помилку.

3.7 Повернення об'єктів функціями

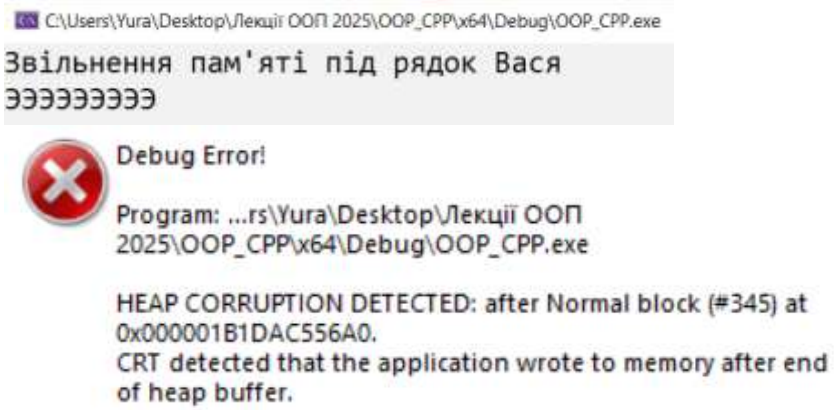
Якщо об'єкти можна передавати функціям, то «з таким же успіхом» функції можуть повертати об'єкти. Щоб функція могла повернути об'єкт, по-перше, необхідно оголосити в якості типу значення що повертається нею тип відповідного класу. По-друге, потрібно забезпечити повернення об'єкта цього типу за допомогою звичайної інструкції return. Розглянемо приклад функції, яка повертає об'єкт.

Щодо повернення об'єктів функціями важливо розуміти наступне. Якщо функція повертає об'єкт класу, вона автоматично створює тимчасовий об'єкт, який зберігає значення, що повертається. Саме цей об'єкт реально і повертається функцією. Після повернення значення об'єкт руйнується. Руйнування тимчасового об'єкта в деяких ситуаціях може викликати непередбачені побічні ефекти. Наприклад, якщо об'єкт, що повертається функцією, має деструктор, який звільняє пам'ять, що динамічно виділяється. Ця пам'ять буде звільнена навіть в тому випадку, якщо об'єкт, який одержує значення, що повертається функцією, все ще її використовує. Розглянемо програму:

```
// Помилка, що генерується при поверненні об'єкта функцією.
class Text {
     char* Ptext;
public:
     Text() { Ptext = nullptr; }
     ~Text() {
          if (Ptext)
          {
          cout << "Звільнення пам'яті під рядок " << Ptext << endl;
               delete[] Ptext;
          }
     }
```

```
	void Print() { cout << Ptext << endl; }

	// Завантаження рядка.
	void SetStr(char* str)
	{
		Ptext = new char[strlen(str) + 1];
		strcpy_s(Ptext, sizeof(Ptext), str);
	}
};

// Ця функція повертає об'єкт типу Text.
Text input()
{
	char instr[] = "Вася";
	Text str;
	str.SetStr(instr);
	return str;
}

int main()
{
	setlocale(0, "");
	Text ob;
	// Надаємо об'єкт ob, що повертається функцією input()
	ob = input(); // Ця інструкція генерує помилку!!!!
	ob.Print(); // Відображення "сміття".
	_getch();
}
```

Результати виконання цієї програми виглядають таким чином:



Коли функція повертає об'єкт, автоматично генерується невидимий тимчасовий об'єкт, який зберігає значення, що повертається. В даному випадку цей об'єкт просто є побітову копію об'єкта str, який є значенням, що повертається функцією. Отже, після повернення з функції виконується деструктор тимчасового об'єкта. Оскільки область пам'яті для зберігання рядка, що вводиться користувачем, вже була звільнена, при виконанні функції Print() на екран виведеться «сміття».

Для вирішення цієї проблеми замість повернення об'єкта з функції використовується повернення вказівника або посилання на об'єкт. Але це не завжди можливо. Ще один спосіб вирішення цієї проблеми включає використання конструктора копії, якому присвячений наступний розділ.

3.8 Створення і використання конструктора копії

Одним з найбільш важливих форматів перевантаженого конструктора є конструктор копії. Як було показано в попередніх прикладах, при передачі об'єкта функції або повернення об'єкту з функції можуть виникати проблеми. Один із способів уникнути цих проблем полягає у визначенні конструктора копії, який являє собою спеціальний тип перевантаженого конструктора.

Конструктор копії дозволяє управляти діями, які становлять процес створення копії об'єкта.

В основі розглянутих проблем лежить створення побітової копії об'єкта. Щоб запобігти їх виникненню, необхідно точно визначити, що повинно відбуватися, коли створюється копія об'єкта, і тим самим уникнути небажаних побічних ефектів. Цього можна домогтися шляхом створення конструктора копії.

Перш ніж докладніше знайомитися з використанням конструктора копії, важливо розуміти, що в C++ визначено два окремі види ситуацій, в яких значення одного об'єкта передається іншому. Першою такою ситуацією є присвоювання, а другий – ініціалізація. Ініціалізація може виконуватися трьома способами, тобто у випадках, коли:

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

Конструктор копії застосовується тільки до ініціалізації. Він не застосовується до присвоювання.

Ось як виглядає найпоширеніший формат конструктора копії:

```
ClassName(const ClassName & obj) {
 // тіло конструктора
}
```

Тут елемент obj означає посилання на об'єкт, який використовується для ініціалізації іншого об'єкта.

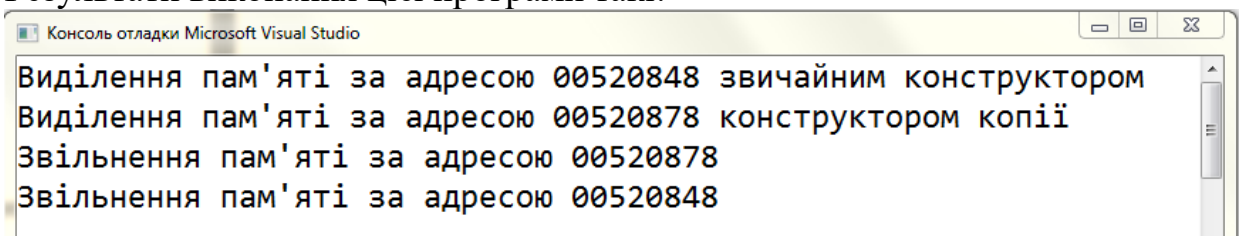
При передачі об'єкта функції в якості аргументу створюється копія цього об'єкта. Якщо в класі визначено конструктор копії, то саме він і викликається для створення копії.

Конструктор копії також викликається в разі, коли один об'єкт використовується для ініціалізації іншого.

Розглянемо наступну просту програму.

```
class MyCl
{
 int *p;

public:
 // звичайний конструктор.
 MyCl(int i)
 {
      p = new int;
      *p = i;
      cout << "Виділення пам'яті за адресою " << p;
      cout << " звичайним конструктором "<< endl;
 }
 // конструктор копії
 MyCl(const MyCl &obj)
 {
      p = new int;
      *p = *obj.p; // значення копії
      cout << "Виділення пам'яті за адресою " << p;
      cout << " конструктором копії" << endl;
 }
 ~MyCl()
 {
      cout << "Звільнення пам'яті за адресою " << p << endl;
      delete p;
 }
};
int main()
{
 setlocale(0, "Ukr");
 MyCl a(10); // Викликається звичайний конструктор.
 MyCl b = a; // Викликається конструктор копії.
}
```

Результати виконання цієї програми такі:



Слід мати на увазі, що конструктор копії викликається тільки в разі виконання ініціалізації. Наприклад, наступна послідовність інструкцій не викличе конструктор копії, визначений у попередній програмі:

```
MyCl a (2), b (3);
// ...
b = a;
```

Тут інструкція b = a виконує операцію присвоювання, а не операцію копіювання.

Конструктор копії також викликається при створенні тимчасового об'єкта, який є результатом повернення функцією об'єкта.

Початківці у програмуванні часто не розуміють, чому такий важливий конструктор копії. Для багатьох не відразу стає очевидним відповідь на питання: коли потрібен конструктор копії, а коли – ні. Ця ситуація часто виражається в такій формі: «А чи не існує більш простого способу?». Відповідь також непроста: і так, і ні!

Такі мови, як Java і C #, не мають конструкторів копії, оскільки в жодному з них не створюються побітові копії об'єктів. Справа в тому, 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 в якості параметрів функції або при поверненні їх з функцій в копіях об'єктів немає ніякої необхідності.

Той факт, що а ні Java, а ні C# не потребують конструкторів копії, робить ці мови простіше, але за простоту теж потрібно платити. Робота з об'єктами виключно за допомогою посилань (а не безпосередньо, як в C++) накладає обмеження на тип операцій, які може виконувати програміст.

3.9 Ключове слово this

Ключове слово *this* – це вказівник на поточний об'єкт даного класу. Відповідно через *this* ми можемо звертатися всередині класу до будь-яких його членів.

Для звернення до змінних використовується вказівник *this*. Причому після this ставиться не крапка, а стрілка ->.

У більшості випадків для звернення до членів класу навряд чи знадобиться ключове слово this. Але воно може бути необхідно, якщо параметри функції або змінні, які визначаються всередині функції, мають такі ж імена, як і змінні класу. Наприклад, щоб в конструкторі розмежувати параметри і змінні класу якраз і використовується вказівник this.

```
class P
{
 int x;
 int y;

public:
 P(int x, int y)
 {
      this->x = x;
      this->y = y;
 }
```

```
 void Show()
 {
       cout << "x: " << this->x << "\t y: " << y << endl;
 }
};

int main()
{
 P p1(20, 50);
 p1.Show();
}
```

В цьому прикладі визначено клас P, який представляє точку на площині. І для зберігання координат точки в класі визначено змінні x і y.

За допомогою this можна повертати поточний об'єкт класу:

```
class P
{
 int x;
 int y;

public:
    P(int x, int y)
    {
        this->x = x;
        this->y = y;
    }
    void Show()
    {
        cout << "x: " << this->x << "\t y: " << y << endl;
    }

    P Step(int x, int y)
    {
        this->x += x;
        this->y += y;
        return *this; // повертає поточний об'єкт класу.
    }
};
```

За допомогою this можна також повертати посилання поточний об'єкт класу:

```
    P &Step (int x, int y)
    {
        this->x += x;
        this->y += y;
        return *this;
    }
```

Тут метод Step за допомогою вказівника this повертає посилання на об'єкт поточного класу.

Резюме

1. Ключове слово friend надає функції, яка не є членом класу, доступ до його закритих членів.

2. Функція може бути «другом» декількох класів.

3. «Друг» одного класу може бути членом іншого класу.

4. Щоб перевантажити конструктор класу, досить оголосити його в усіх потрібних форматах і визначити кожну дію, пов'язану з відповідним форматом.

5. Коли при виконанні функції створюється копія аргументу, звичайний конструктор не викликається. Замість цього викликається конструктор копії об'єкта. Конструктор копії визначає, як повинна бути створена копія об'єкта. Але якщо в класі явно не визначений конструктор копії, C++ надає його за замовчуванням. Конструктор копії за замовчуванням створює побітову (тобто ідентичну) копію об'єкта.

6. При передачі об'єкта функції в якості аргументу створюється копія цього об'єкта. Якщо в класі визначено конструктор копії, то саме він і викликається для створення копії.

7. Конструктор копії також викликається в разі, коли один об'єкт використовується для ініціалізації іншого.

8. Конструктор копії застосовується тільки для ініціалізації. Він не застосовується для присвоювання об’єктів.

9. Конструктор копії також викликається при створенні тимчасового об'єкта, який є результатом повернення функцією об'єкта.

10. Ключове слово this – це вказівник на поточний об'єкт даного класу. Відповідно через this ми можемо звертатися всередині класу до будь-яких його членів.

Завдання до виконання лабораторної роботи

1. Визначити користувальницький клас у відповідності з варіантом завдання.
2. Створити звичайний конструктор, який виділяє динамічну пам’ять під змінні-вказівники.
3. Створити конструктор копії.
4. Деструктор звільняє динамічну пам’ять.
5. Створити дружню функцію у відповідності з варіантом завдання.

Варіант 1.

Створити базовий клас «комплексне число», такий, що:

1. його екземпляр містить вказівники на дійсну і уявну частину – змінні з плаваючою точкою;
2. його звичайний конструктор, який виділяє динамічну пам’ять під змінні-вказівники, створює екземпляр з відповідними значеннями параметрів для дійсної та уявної частини, а конструктор копії визначає, як повинна бути створена копія об'єкта;
3. його деструктор звільняє динамічно виділену пам’ять під дійсну і уявну частину;
4. його функції дозволяють отримувати і присвоювати значення дійсної та уявної частини;
5. функція Print() виводить на екран значення екземпляра у вигляді (0.00 ± i0.00);
6. дружня функція frd в якості параметрів отримує екземпляри двох «комплексних чисел», і повертає нове «комплексне число», дійсна і уявна частини якого дорівнюють сумам дійсних і уявних частин цих двох «комплексних чисел».

Варіант 2.

Створити клас «куля», такий, що:

1. його екземпляр містить вказівник на змінну зі значенням радіусу;
2. його звичайний конструктор, який виділяє динамічну пам’ять під змінні-вказівники, створює екземпляр з відповідним значенням радіусу, а конструктор копії визначає, як повинна бути створена копія об'єкта;
3. його деструктор звільняє динамічно виділену пам’ять під радіус кулі;
4. його функції дозволяють отримувати і присвоювати значення радіуса, та обчислювати значення об'єму;
5. функція Print() виводить на екран значення його радіусу і об'єму;
6. дружня функція frd в якості параметрів отримує екземпляри двох куль і повертає нову кулю, об’єм якої дорівнює сумі об’ємів цих двох куль. Значення радіусу нової кулі має бути перераховане.

Варіант 3.

Створити клас «циліндр», такий, що:

1. його екземпляр містить вказівники на змінну зі значенням радіусу і висоти;
2. його звичайний конструктор, який виділяє динамічну пам’ять під змінні-вказівники, створює екземпляр з відповідним значенням радіуса і висоти, а конструктор копії визначає, як повинна бути створена копія об'єкта;
3. його деструктор звільняє динамічно виділену пам’ять під радіус і висоту циліндра;
4. його функції дозволяють отримувати і присвоювати значення радіуса і висоти та обчислювати значення об'єму;
5. функція Print () виводить на екран значення його радіуса, висоти та об’єму;
6. дружня функція frd в якості параметрів отримує екземпляри двох циліндрів і повертає новий циліндр, об’єм якого дорівнює сумі об’ємів цих двох циліндрів, а радіус дорівнює максимальному серед цих двох циліндрів.

Варіант 4.

Створити клас «квадрат», такий, що:

1. його екземпляр містить вказівник на змінну зі значенням радіусу;
2. його звичайний конструктор, який виділяє динамічну пам’ять під змінні-вказівники, створює екземпляр з відповідним значенням сторони, а конструктор копії визначає, як повинна бути створена копія об'єкта;
3. його деструктор звільняє динамічно виділену пам’ять під сторону квадрата;
4. його функції дозволяють отримувати і присвоювати значення сторони, та обчислювати значення площі;
5. функція Print() виводить на екран значення сторони і площі квадрата;
6. дружня функція frd в якості параметрів отримує екземпляри двох квадратів і повертає новий квадрат, площа якого дорівнює сумі площ цих двох квадратів. Значення сторони нового квадрата має бути перераховане.

Варіант 5.

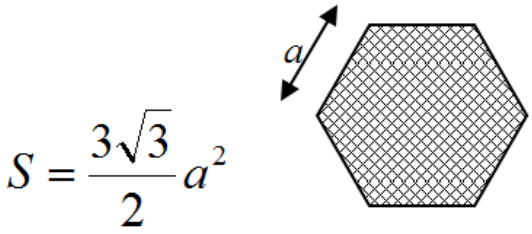
Створити клас «безпечний масив», такий, що:

1. його екземпляр містить вказівник на динамічний масив цілих чисел і змінну – розмір масиву;
2. його звичайний конструктор виділяє динамічну пам’ять під масив цілих чисел заданого розміру і заповнює масив псевдовипадковими числами в діапазоні від min до max, при цьому розмір масиву і діапазон значень його елементів задається параметрами конструктора;

3. його конструктор копії визначає, як повинна бути створена копія об'єкта;
4. його деструктор звільняє динамічно виділену пам’ять під масив;
5. його функції дозволяють отримувати і присвоювати значення елементів масиву за їх індексом, і виводять повідомлення про помилку, якщо індекс перевищує допустимий діапазон.
6. функція Print() виводить на екран значення елементів масиву цілих чисел;
7. дружня функція frd в якості параметрів отримує екземпляри двох масивів «зливає» два масива в один в масив сумарного розміру і повертає новий масив.

Варіант 6.

1. Створити клас «шайба» (правильний шестикутник, такий, що:



$$S = \frac{3\sqrt{3}}{2} a^2$$

a) його екземпляр містить розмір сторони a.
b) його конструктор без параметра створює екземпляр зі значенням 0, а конструктор з параметрами створює екземпляр з відповідним значенням a.
c) його методи дозволяють отримувати і присвоювати значення a і обчислювати значення площі.
d) функція print_sqrt () виводить на екран значення сторони і площі цієї фігури.
e) дружня функція frd в якості параметрів отримує екземпляри двох шайб і повертає нову шайбу, площа якої дорівнює сумі площ цих двох шайб. Значення сторони нової шайби має бути перераховане.

Тести для закріплення матеріалу

1. Що називається елементами класу?
   а) тільки члени-дані та члени-функції;
   б) тільки конструктор і деструктор;
   в) тільки члени Дані;
   г) тільки члени-функції.

2. this – це
   а) неявно визначений вказівник на поточний об'єкт класу;
   б) поточний клас;
   в) адреса поточного методу класу;
   г) неявно визначене посилання на поточний об'єкт класу.

3. Якщо клас містить конструктор за замовчуванням і конструктор з параметрами (C ++), ці конструктори
   а) викликаються послідовно в порядку оголошення;
   б) спочатку викликається конструктор за замовчуванням, потім - з параметрами;
   в) викликається тільки один з них;
   г) при наявності завжди викликається конструктор з параметрами.

4. Дружні функції можуть
   а) звертатися тільки до закритих даних класу;
   б) звертатися тільки до відкритими даними класу;
   в) звертатися тільки до захищених даних класу;
   г) звертатися до всіх елементів класу.

5. Функція, оголошена як дружня (C ++)
   а) має ключове слово friend в прототипі і в реалізації;
   б) має ключове слово friend в реалізації і не має в прототипі;
   в) має ключове слово friend в прототипі і не має в реалізації;
   г) має ключове слово friend перед викликом.

6. Коли не викликається конструктор копії?
   а) при передачі об’єкта як параметра функції;
   б) при виконанні операції присвоювання одного об’єкта іншому;
   в) при ініціалізації об’єкта;
   г) при поверненні об’єкта функцією.

7. Коли викликається конструктор копії?
   а) при передачі посилання на об’єкта як параметра функції;
   б) при виконанні операції присвоювання одного об’єкта іншому;
   в) при передачі об’єкта як параметра функції;
   г) при створенні об’єкта.

8. Вкажіть правильне визначення конструктор копії:

а)
```
ClassName(int x=0)
{
   P = new int;
   *P = x;
}
```

б)
```
ClassName(int x=size)
{
   P = new int;
   *P = x;
}
```

в)
```
ClassName(ClassName obj)
{
   P = new int;
   *P = *(obj.P);
}
```

г)
```
ClassName(const ClassName &obj)
{
   P = new int;
   *P = *(obj.P);
}
```

РОЗДІЛ 4. ПЕРЕВАНТАЖЕННЯ ОПЕРАТОРІВ

У С++ оператори є функціями, в яких значення операндів є параметрами функцій. Як відомо, функції в С++ можна перевантажувати, отже, оператори, як і інші функції, можна перевантажувати для типів даних, які визначаються програмістом. Принциповий виграш від перевантаження операторів полягає в тому, що воно дозволяє органічно інтегрувати нові типи даних в середовище програмування. По суті, перевантаження операторів є одним із проявів поліморфізму, коли програміст використовує один, звичний йому інтерфейс (оператор) для виконання різних операцій з новими типами даних.

Щоб перевантажити оператор, необхідно визначити значення нової операції для класу, до якого вона буде застосовуватися. Для цього створюється функція operator (операційна функція), яка визначає дію цього оператора. Загальний формат функції operator такий:

тип operator # (список аргументів)
{
операція_над_класом
}

Оператори перевантажуються за допомогою функції *operator*.

Тут оператор, що перевантажується, позначається символом "#", а елемент тип являє собою тип значення, що повертається заданої операцією.

Функція-оператор може бути членом класу чи не бути їм. Операторні функції, які не є членами класу, часто визначаються як його «друзі». Операторні функції-члени і функції, які не є членами класу, розрізняються за формою перевантаження. Кожен з варіантів ми розглянемо окремо. Варто відразу зазначити, що є функції-оператори, які можуть бути визначені лише як члени класу і є функції-оператори, які можуть бути визначені лише поза класом.

Перевантажити можна тільки оператори, які вже визначені в С++. Створити нові оператори не можна.

У мові С++ дозволено перевантажувати наступні оператори:

```
+     -     *     /     %     ^     &     |     ~
!     ,     =     <     >     <=    >=    ++    --
<<    >>    ==    !=    &&    ||    +=    -=    /=
%=    ^=    &=    |=    *=    <=    >>=   []    ()
->    ->*   new   new[] delete delete[]
```

Забороняється перевантажувати оператори

```
::   .*   .   ?
```

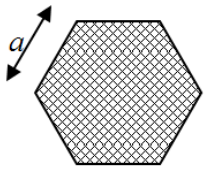
Крім того, забороняється перевантажувати будь-який оператор для вбудованих (стандартних) типів.

4.1 Перевантаження операторів з використанням функцій не членів класу

Якщо функція оператора визначена як окрема функція і не є членом класу, кількість параметрів такої функції збігається з кількістю операндів оператора.

Почнемо з простого прикладу. У наступній програмі створюється клас Honeycomb (бджолина сота), який визначає правильний шестикутник зі стороною a.

$$S = \frac{3\sqrt{3}}{2}a^2$$

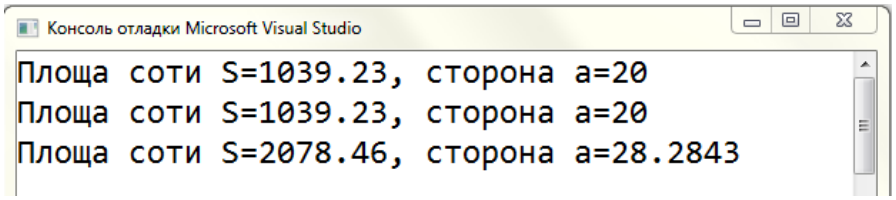


Для класу Honeycomb перевантажується оператор +. Уявимо собі, що дві бджоли вирішили об'єднати своє майно, для чого їм потрібно замість двох різних сот отримати одну з площею, що дорівнює сумі двох сот. Для цього потрібно обчислити сторону цієї фігури.

Отже, розглянемо уважно код цієї програми.

```
class Honeycomb
{
 double a, area;
public:
 Honeycomb(double a = 0)
 {
      this->a = a;
      area = 3 * sqrt(3)*a*a / 2;
 }
 // фунуція-оператор + не є членом класу, це дружня функція
 friend Honeycomb operator + (Honeycomb a, Honeycomb b);
 void Show()
 {
      cout << "Площа соти S=" << area << ", сторона a=" << a << endl;
 }
};
Honeycomb operator + (Honeycomb a, Honeycomb b)
{
 double S = a.area + b.area;
 // Знаходимо сторону нової фігури
 double side = sqrt(2 * S / (3 * sqrt(3)));
 Honeycomb NewHoney(side);
 return NewHoney;
}

int main()
{
 setlocale(0, "Ukr");
 Honeycomb h1(20);
 Honeycomb h2(20);
 Honeycomb hsum=h1 + h2;
 h1.Show(); h2.Show(); hsum.Show();
}
```

При виконанні ця програма генерує такі результати:



При виклику операторної функції +, визначеної поза класом, їй передаються два параметри: перший – той, що стоїть ліворуч, а другий - той, що стоїть праворуч.

З використанням функцій-не членів класу не можна перевантажувати такі оператори:

`=, (), [] , ->.`

Зверніть увагу на те, що функція operator + () повертає об'єкт типу Honeycomb. Незважаючи на те що вона могла б повертати значення будь-якого допустимого в C++ типу, той факт, що вона повертає об'єкт типу Honeycomb, дозволяє використовувати оператор "+" в таких складових виразах, як a + b + c. Частина цього виразу, a + b, генерує результат типу Honeycomb, який потім складається з об'єктом c. І якби ця частина виразу генерувала значення іншого типу (а не типу Honeycomb), такий складний вираз просто не працював би.

4.2 Перевантаження операторів з використанням функцій членів класу

При перевантаженні бінарного оператора з використанням функції-члена їй передається явно тільки один аргумент. Другий же неявно передається через this. Таким чином, в рядку

temp.x = x + op2.x;

Під членом x мається на увазі член this-> x, тобто член x зв'язується з об'єктом, який викликає дану операторну функцію. У всіх випадках неявно передається об'єкт, що вказується зліва від символу операції, який став причиною виклику операторної функції. Об'єкт, що розташовується праворуч від символу операції, передається цій функції як аргумент. У загальному випадку при використанні функції-члена для перевантаження унарного оператора параметри не використовуються взагалі, а для перевантаження бінарного – тільки один параметр. *Тернарний оператор «?» перевантажувати не можна*.

На відміну від оператора "+", оператор присвоювання призводить до модифікації одного зі своїх аргументів. (Перш за все, це становить саму суть присвоювання.) Оскільки функція operator = () викликається об'єктом, який розташований зліва від символу присвоювання (=), саме цей об'єкт і модифікується в результаті операції присвоювання. Після виконання цієї операції значення, що повертається перевантаженим оператором, містить об'єкт, який розташований зліва від символу присвоювання.

Необхідно, щоб функція operator = () повертала об'єкт, що адресується вказівником *this*, і щоб цей об'єкт був розташований зліва від оператора "=". Це дозволить виконати будь-який ланцюжок присвоювання. Наприклад, щоб можна було виконувати інструкції, подібні наступної:

a = b = c = d;

Операція присвоювання – це одне з найважливіших застосувань вказівника *this*.

Якщо оператор = не визначений, при виконанні операції присвоювання:

obj1 = obj2;

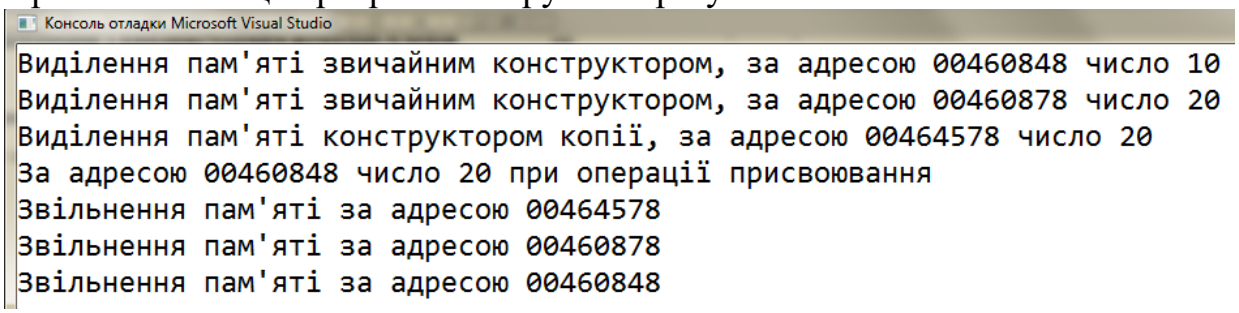
автоматично створюється побітова копія об'єкта obj2. Якщо в класі не застосовується динамічна пам'ять, яка звільняється деструктором, тоді можна оператор присвоювання не перевантажувати. Однак якщо конструктор виділяє, а деструктор звільняє динамічну пам'ять, необхідно перевантажувати оператор присвоювання так, щоб він виконував дії, подібні до тих, що виконує конструктор копії.

Розглянемо приклад:

```
class MyCl
{
 int *p;

public:

 // звичайний конструктор.
 MyCl(int i=0)
 {
      p = new int;
      *p = i;
      cout << "Виділення пам'яті звичайним конструктором,";
      cout << " за адресою " << p<<" число "<< *p << endl;
 }

 // конструктор копії
 MyCl(const MyCl &obj)
 {
      p = new int;
      *p = *obj.p; // значення копії
      cout << "Виділення пам'яті конструктором копії,";
      cout << " за адресою " << p << " число " << *p << endl;
 }
 ~MyCl()
 {
      cout << "Звільнення пам'яті за адресою " << p << endl;
      delete p;
 }
 // перевантаження оператора =
 MyCl & operator =(const MyCl &obj)
 {
      *p = *obj.p; // значення копії
      cout << "За адресою " << p << " число " << *p;
      cout << " при операції присвоювання" << endl;
      return *this;
 }
};
```

```
int main()
{
 setlocale(0, "Ukr");
 MyCl a(10); // Викликається звичайний конструктор.
 MyCl b(20); // Викликається звичайний конструктор.
 MyCl c = b; // Викликається  конструктор копії.
 a = b; // Викликається  перевантаженний оператор =
}
```

При виконанні ця програма генерує такі результати:



Можна також перевантажувати унарні оператори таки, як "++", "–", або унарні "–" і "+". При перевантаженні унарних оператора за допомогою функції-члена операторної функції жоден об'єкт не передається явно. Операція ж виконується над об'єктом, який генерує виклик цієї функції через неявно переданий покажчик this.

Розглянемо приклад:

```
// Перевантаження префіксної версії оператора "++".
class Vector
{
 double x, y;
public:

 Vector(double x = 0, double y=0)
 {
      this->x = x;
      this->y = y;
 }

 // Перевантаження префіксної версії оператора "++".
 Vector operator ++ ()
 {
      x++; // інкремент координат x, y
      y++;
      return *this;
 }
```

```
 // Перевантаження постфіксної версіяї оператора "++".
 Vector operator ++ (int notused)
 {
       Vector temp = *this; // збереження вихідного значення
       x++; // інкремент координат x, y
       y++;
       return temp; // повернення початкового значення
 }

 void Show()
 {
       cout << "x=" << x << ", y=" << y << endl;
 }
};


int main()
{
 Vector c(10, 20);
 // префіксна форма інкремента
 Vector a = ++c; // Об'єкт a отримує значення об'єкта c після його
 //інкрементування.
 a.Show(); // Тепер об'єкти a й c
 c.Show(); // мають однакові значення.
 // Постфіксна форма інкремента
 a = c++; // Об'єкт a отримує значення об'єкта c до його
 //інкрементування.
 a.Show(); // Тепер об'єкти a й c
 c.Show(); // мають різні значення.
}
```

Оператори інкременту і декременту мають як префіксну, так і постфіксну форми.

Параметр notused не використовується самою функцією. Він служить індикатором для компілятора, що дозволяє відрізнити префіксну форму оператора інкременту від постфіксної. (Цей параметр також використовується в якості ознаки постфіксної форми і для оператора декременту.)

4.3 Перевантаження операторів відношення і логічних операторів

Оператори відношення (наприклад, `==` або `<`) і логічні оператори (наприклад, `&&` або `||`) також можна перевантажувати, причому робити це зовсім неважко. Як правило, перевантажена операторна функція відношення повертає об'єкт класу, для якого вона перевантажується. А перевантажений оператор відношення або логічний оператор повертає одне з двох можливих значень: `true` або `false`. Це відповідає звичайному застосуванню цих операторів і дозволяє використовувати їх в умовних виразах.

Розглянемо приклад перевантаження оператора "==" для вже знайомого нам класу `Vector`.

```
// Перевантаження оператора "=="
bool operator == (Vector op2)
{
if ((x == op2.x) && (y == op2.y))
return true;
else
return false;
}
```

Якщо вважати, що операторна функція operator == () вже реалізована, наступний фрагмент коду абсолютно коректний.

```
Vector a, b;
// ...
if(a == b)
 cout << "a дорівнює b";
else
 cout << "a не дорівнює b";
```

Ряд операторів перевантажуються парами. Наприклад, якщо ми перевантажуємо оператор ==, необхідно також перевантажити і оператор !=. А для перевантаження оператора < треба також перевантажити функцію для оператора >. І якщо для C++ це правило носить рекомендаційний характер, то в C# недотримання цього правила призведе до помилки компіляції.

4.4 Перевантаження оператора індексації масивів ([])

Крім традиційних операторів C++ дозволяє перевантажувати і більш «екзотичні», наприклад, оператор індексації масивів `[].` У C++ (з точки зору механізму перевантаження) оператор `[]` вважається бінарним. Його можна перевантажувати тільки для класу і тільки з використанням функції-члена. Ось як виглядає загальний формат операторної функції-члена operator [ ] ().

```
тип operator [] (int індекс)
{
// ...
}
```

Припустимо, у нас визначено об'єкт ob, тоді вираз
`ob[3];` перетвориться в наступний виклик операторної функції operator []():

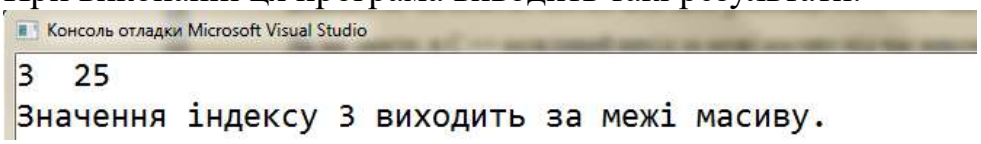
```
ob.operator [] (3);
```

Іншими словами, значення виразу, заданого в операторі індексації, передається операторної функції operator [] () в якості явно заданого аргументу. При цьому покажчик this буде вказувати на об'єкт ob, тобто об'єкт, який генерує виклик цієї функції.

Наприклад, наступна програма оснащена засобом контролю попадання в допустимий інтервал.

```
// Приклад організації безпечного масиву.
const int SIZE = 3;
class SmartArray {
 int A[SIZE];
public:
 SmartArray() {
      for (int i = 0; i < SIZE; i++)
           A[i] = i+1;
 }
 // Контроль попадання в допустимий інтервал для класу SmartArray.
 int & operator [] (int i)
 {
      if (i <0 || i> SIZE - 1) {
           cout << "Значення індексу ";
           cout << i << " виходить за межі масиву.";
           exit(1);
      }
      return A[i];
 }
};

int main()
{
 setlocale(0, "Ukr");
 SmartArray ob;
 cout << ob[2]; // Відображається число 2.
 cout << "  ";
 ob[2] = 25; // Оператор "[]" стоїть в лівій частині.
 cout << ob[2] << endl; // Відображається число 25.
 ob[3] = 44; // Генерується помилка
 // оскільки значення 3 виходить за межі масиву.
}
```

При виконанні ця програма виводить такі результати.



Тут функція operator [] () повертає значення і-го елемента масиву A. Таким чином, вираз ob[2] повертає число 3, яке відображається інструкцією cout. Ініціалізація масиву a за допомогою конструктора виконується лише в ілюстративних цілях.

Одна з переваг перевантаження оператора "[]" полягає в тому, що з його допомогою ми можемо забезпечити засіб реалізації безпечної індексації масивів. Як ви знаєте, в C ++ можливий вихід за межі масиву під час виконання програми без відповідного повідомлення (тобто без генерування повідомлення про

динамічну помилку). Але якщо створити клас, який містить масив, і дозволити доступ до цього масиву тільки через перевантажений оператор індексації "[]", то можна перехопити індекс, значення якого вийшло за дозволені межі.

```
ob [3] = 44;
```

операторною функцією operator [] () перехоплюється помилка порушення межи масиву, після чого програма тут же завершується, щоб не допустити будь-яких потенційно можливих руйнувань.

При створенні власних класів завжди має сенс експериментувати з перевантаженням операторів. Механізм перевантаження операторів можна використовувати для додавання нових типів даних в середовище програмування. Це один з найбільш потужних засобів С ++.

Резюме

1. Щоб перевантажити оператор, необхідно визначити значення нової операції для класу, до якого вона буде застосовуватися. Для цього створюється функція operator (операційна функція), яка визначає дію цього оператора.
2. При перевантаженні бінарного оператора з використанням функції-члена їй передається явно тільки один аргумент. Другий же неявно передається через this.
3. При перевантаженні унарних оператора за допомогою функції-члена операторної функції жоден об'єкт не передається явно. Операція ж виконується над об'єктом, який генерує виклик цієї функції через неявно переданий покажчик this.
4. З використанням функцій-не членів класу не можна перевантажувати такі оператори: =, (), [] , ->.
5. Функція operator «=» повертає об'єкт, що адресується вказівником this, і цей об'єкт має бути розташований зліва від оператора «=».

6. У С++ з точки зору механізму перевантаження оператор [] вважається бінарним.
7. Якщо перевантажується оператор ==, необхідно також перевантажити і оператор !=. А для перевантаження оператора < треба також перевантажити функцію для оператора >.
8. Механізм перевантаження операторів можна використовувати для додавання нових типів даних в середовище програмування.

Завдання до виконання лабораторної роботи

1. Визначити і реалізувати клас відповідно варіанту.
2. Визначити і реалізувати операції над даними цього класу відповідно варіанту.
3. Для всіх варіантів при перевантаженні операторів «інкремент» (++) або «декремент» (--) реалізувати як префіксну, так і постфіксну форму.

4. Для всіх варіантів перевантажити оператор << для виводу на консоль інформації про об’єкт.

Варіант 1.

Створити клас «трикутник», такий, що:

1. його екземпляр містить його вказівники на змінні зі значеннями трьох сторін трикутника;
2. його звичайний конструктор, який виділяє динамічну пам’ять під змінні-вказівники, створює екземпляр з відповідним значенням значеннями трьох сторін трикутника, а конструктор копії визначає, як повинна бути створена копія об'єкта;
3. його деструктор звільняє динамічно виділену пам’ять;
4. його функції дозволяють отримувати і присвоювати значення сторін і обчислювати площу (наприклад, за формулою Герона),

$$s = \sqrt{p(p-a)(p-b)(p-c)}, p = (a+b+c)/2$$

5. функція-оператор << перевантажена: виводить на екран значення трьох сторін і площі трикутника;
6. функція-оператор = модифікує об’єкт який розташований зліва від оператора =;
7. функція-оператор ++ збільшує всі сторони трикутника на одиницю.

Варіант 2.

Створити клас «куля», такий, що:

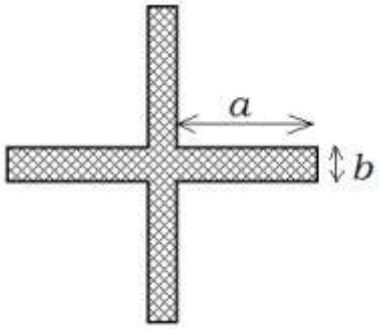
1. його екземпляр містить його вказівник на змінну зі значенням радіусу кулі;
2. його звичайний конструктор, який виділяє динамічну пам’ять під змінні-вказівники, створює екземпляр з відповідним значенням радіусу, а конструктор копії визначає, як повинна бути створена копія об'єкта;
3. його деструктор звільняє динамічно виділену пам’ять;
4. його функції дозволяють отримувати і присвоювати значення радіусу і обчислювати об’єм кулі

$$V = \frac{4\pi R^3}{3};$$

5. функція-оператор << перевантажена і виводить на екран значення радіусу і об’єму кулі;
6. функція-оператор = модифікує об’єкт який розташований зліва від оператора =;
7. функція-оператор + для двох куль створює нову кулю сумарного об’єму, радіус нової кулі має бути перевизначений.

Варіант 3.

Створити клас «хрест», такий, що:



1. його екземпляр містить його вказівник на змінні зі значеннями *a* і *b*;
2. його звичайний конструктор, який виділяє динамічну пам’ять під змінні-вказівники, створює екземпляр з відповідним значенням *a* і *b*, а конструктор копії визначає, як повинна бути створена копія об'єкта;
3. його деструктор звільняє динамічно виділену пам’ять;
4. його функції дозволяють отримувати і присвоювати значення *a* і *b* і обчислювати площу цієї фігури (формулу виведіть самостійно);
5. функція-оператор << перевантажена і виводить на екран значення розміри *a* і *b* і площу цієї фігури;
6. функція-оператор = модифікує об’єкт який розташований зліва від оператора =;
7. функція-оператор ++ збільшує значення змінних *a* і *b* на одиницю.

Варіант 4.

Створити клас «комплексне число», такий, що:

1. його екземпляр містить його вказівник на змінну зі значенням дійсної і уявної частини;
2. його звичайний конструктор, який виділяє динамічну пам’ять під змінні-вказівники, створює екземпляр з відповідним значенням дійсної і уявної частини, а конструктор копії визначає, як повинна бути створена копія об'єкта;
3. його деструктор звільняє динамічно виділену пам’ять;
4. операція «+» перевантажена: для двох комплексних чисел створює нове комплексне число, дійсна і уявна частина якого дорівнює сумі дійсних і уявних частин двох комплексних доданків;
5. функція-оператор ++ перевантажена як префіксна так і постфіксна і збільшує на одиницю дійсну і уявну частину комплексного числа.
6. функція-оператор = перевантажена і модифікує об’єкт який розташований зліва від оператора =;
7. функція-оператор << перевантажена і виводить на екран значення дійсної і уявної частини комплексного числа.

Варіант 5.

Створити клас «безпечний масив», такий, що:

1. його екземпляр містить вказівник на динамічний масив цілих чисел і змінну – розмір масиву;

2. його звичайний конструктор виділяє динамічну пам’ять під масив цілих чисел заданого розміру і заповнює масив нулями, при цьому розмір масиву задається параметром конструктора;
3. його конструктор копії визначає, як повинна бути створена копія об'єкта;
4. його деструктор звільняє динамічно виділену пам’ять під масив;
5. функція-оператор «=» модифікує об’єкт який розташований зліва від оператора «=»;
6. функція-оператор «[ ]» дозволяє отримувати і присвоювати значення елементів масиву за їх індексом, і виводить повідомлення про помилку, якщо індекс перевищує допустимий діапазон;
7. Операція << перевантажена і виводить на екран елементи масиву.

Тести для закріплення матеріалу

1. Перевантаження функцій - це

а) використання одного імені для декількох функцій з різними типами значень, що повертаються;

б) перевантаження деструкторів;

в) використання одного імені для декількох функцій з різними списками параметрів;

г) перевантаження директив.

2. Для перевантаження операторів використовують ключове слово

а) new;

б) operacia;

в) operator;

г) назва оператора.

3. До перевантаження функцій можна віднести

а) перевантаження конструкторів;

б) перевантаження деструкторів;

в) перевантаження модифікаторів;

г) перевантаження директив.

4. Коли перевантажується оператор, перевантаження діє

а) для всіх класів, які є в програмі;

б) тільки для класу, в якому оператор визначається;

в) для всіх випадків використання оператора в програмі;

г) також на стандартне визначення оператора з некласовим змінними.

5. Вибрати вірну інструкцію перевантаження операції «унарний мінус» методом класу Point

а) Point & Point :: operator - () {x = -x; y = -y; return * this; }
б) Point :: operator - () {x = -x; y = -y; return * this; }
в) Point & operator - () {x = -x; y = -y; }
г) operator Point - () {x = -x; y = -y; return * this; }

6. Функцію-оператор можна задати тільки як
   а) метод класу, дружню функцію, звичайну функцію;
   б) метод класу;
   в) метод класу, дружню функцію;
   г) звичайну функцію.
7. Коли для перевантаження операторів не можна використовувати функції-члени?
   а) коли справа від оператора розташований екземпляр іншого класу;
   б) коли зліва від оператора розташований екземпляр іншого класу;
   в) коли функція-оператор повертає значення *this;
   г) коли функція-оператор нічого не повертає (значення void).
8. Коли для перевантаження операторів необхідно використовувати тільки функції-члени?
   а) коли справа від оператора розташований екземпляр іншого класу;
   б) коли зліва від оператора розташований екземпляр іншого класу;
   в) коли функція-оператор повертає значення *this;
   г) коли функція-оператор нічого не повертає (значення void).

РОЗДІЛ 5. ВІДНОСИНИ МІЖ КЛАСАМИ

Жоден з об'єктів навколишнього нас світу не існує сам по собі. Птахи літають тому, що є повітря, на яке спираються їхні крила. Кожен з нас пов'язаний з масою інших людей різноманітними родинними, професійними та іншими зв'язками, які передбачають різні типи відносин.

Так само і класи пов'язані між собою. І щоб в повній мірі оволодіти ООП, необхідно зрозуміти суть цих відносин і навчитися їх ідентифікувати.

5.1 Спадкування

Спадкування (узагальнення) – це відношення між більш загальною сутністю, яка називається суперкласом, та її конкретним втіленням, що називається підкласом.

Спадкування – один із трьох фундаментальних принципів об'єктно-орієнтованого програмування, оскільки саме завдяки йому можливе створення ієрархічної класифікації. Використовуючи наслідування, можна створити клас, який визначає найбільш загальні характери всіх пов'язаних із ним елементів. Цей клас потім може бути успадкований іншими, спеціалізованими класами з додаванням до кожного з них своїх, унікальних особливостей.

У стандартній термінології мови C++ клас, що успадковується, називається базовим. Клас, який використовує базовий клас, називається похідним. Похідний клас можна використовувати як базовий для іншого похідного класу. Таким шляхом буде багаторівнева ієрархія класів.

Базовий клас успадковується похідним класом. Для цього базовий клас визначається при оголошенні похідного. Найкраще почати з прикладу (Рис. 5.1)

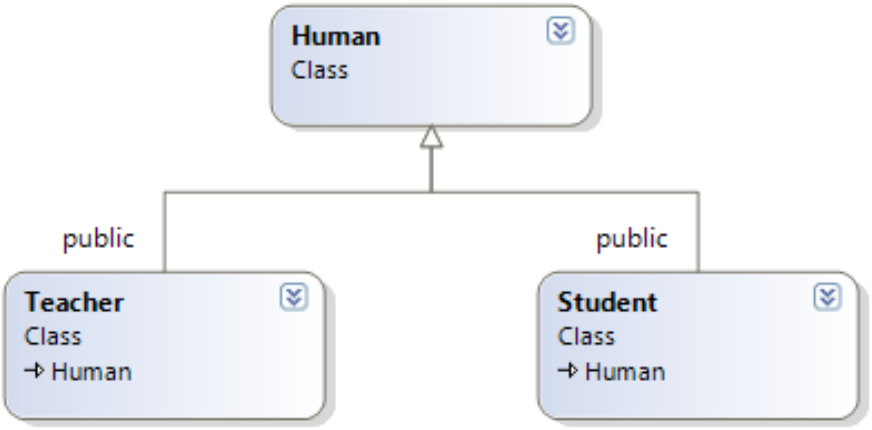


Рис.5.1. Похідні класи Teacher і Student є нащадками класу Human.

Розглянемо клас Human, який в найзагальніших рисах визначає будь-яку людину. Кожна людина має рік народження та ім’я. Саме ці дані зберігають відповідні поля класу Human.

```
class Human
{
 int  year_of_birth; // рік народження
 string name; // ім'я
 public:
    void SetYear(int year) { year_of_birth = year; }
    int GetYear () { return year_of_birth; }
    void SetName(string text) { name = text; }
    string GetName () { return name; }

};
```

Це загальне визначення людини можна використовувати для визначення інших осіб. Наприклад, студента і викладача. В наступному фрагменті шляхом спадкування класу Human створюється клас Student (студент).

```
class Student : public Human
{
 int kurs; // курс
 string group; // група
 public:
    void SetKurs(int num) { kurs = num; }
    int GetKurs () { return kurs; }
    void SetGroup(string text) { group = text; }
    string GetGroup() { return group; }
 void Show ();
};
```

Той факт, що клас `Student` успадковує клас `Human`, означає, що клас `Student` успадковує весь вміст класу `Human`. До вмісту класу `Student` клас `Human` додає поле `kurs i group`, а також функції-члени, які визначають ці поля.

Загальний формат для забезпечення спадкування має такий вигляд.

клас ім'я_похідного_класа: доступ ім'я _базового_класу
{
  тіло нового класу
}

Тут елемент *доступ* необов'язковий. При необхідності він може бути виражений одним з модифікаторів доступу: *public*, *private* або *protected*. Детальніше про ці модифікатори буде сказано в цьому розділі. А поки в визначеннях всіх успадкованих класів будемо використовувати модифікатор *public*. Це означає, що всі public - члени базового класу також будуть public - членами похідного класу. Отже, в попередньому прикладі члени класу `Student` мають доступ до відкритих функцій-членів класу `Human`, як ніби ці функції були оголошені в тілі класу `Human`. Однак клас `Student` не має доступу до private - членів класу `Human`. Так, для класу `Student` закритий доступ до члена даних (поля) name і year_of_birth.

Розглянемо програму, яка використовує механізм спадкування для створення двох похідних класів від класу Human: Student і Teacher.

```
// Демонстрація Спадкування.
// Визначаємо базовий клас людина
class Human
{
      int  year_of_birth; // рік народження
      string name; // ім'я
public:
      void SetYear(int year) { year_of_birth = year; }
      int GetYear() { return year_of_birth; }
      void SetName(string text) { name = text; }
      string GetName() { return name; }
};

// Визначаємо клас студент
class Student : public Human
{
      int kurs; // курс
      string group; // група
public:
      void SetKurs(int num) { kurs = num; }
      int GetKurs() { return kurs; }
      void SetGroup(string text) { group = text; }
      string GetGroup() { return group; }
      void Show();
};

// Визначаємо клас викладач
class Teacher : public Human
{
      int hours; // піднавантаження на рік.
      string discipline; // дисципліна
public:
      void SetHours(int num) { hours = num; }
      int GetHours() { return hours; }
      void SetDiscipline(string text) { discipline = text; }
      string SetDiscipline() { return discipline; }
      void Show();
};

void Student::Show()
{
      cout << "Студент групи " << group << " " << GetName() << endl;
      cout << "навчається на " << kurs << " курсі" << endl;
      cout << "Рік народження " << GetYear() << endl << endl;
}
```

```
void Teacher::Show()
{
	cout << GetName() << " викладає дисципліну \n" << discipline <<
endl;
	cout << "з піднавантаженням " << hours << " годин" << endl << endl;
}
int main()
{
	setlocale(0, "Ukr");
	Student s;
	Teacher t;
	s.SetName("Борис Джонсон");
	s.SetYear(2003);
	s.SetGroup("ІПЗ 240016");
	s.SetKurs(2);
	t.SetName("Билл Гейтс");
	t.SetYear(1955);
	t.SetHours(600);
	t.SetDiscipline("об'єктно-орієнтоване програмування");
	s.Show();
	t.Show();
}
```

При виконанні ця програма генерує такі результати:

Microsoft Visual Studio Debug Console

```
Студент групи ІПЗ 240016 Борис Джонсон
навчається на 2 курсі
Рік народження 2003

Билл Гейтс викладає дисципліну
об'єктно-орієнтоване програмування
з піднавантаженням 600 годин
```

Як видно за результатами виконання цієї програми, основна перевага спадкування полягає в тому, що воно дозволяє створити базовий клас, який потім можна включити до складу більш спеціалізованих класів.

5.2 Управління доступом до членів базового класу

Якщо один клас успадковує інший, члени базового класу стають членами похідного. Статус доступу членів базового класу в похідному класі визначається модифікатором доступу, що використовуються для спадкування базового класу. Модифікатор доступу базового класу виражається одним з ключових слів: *public*, *private* або *protected*. Якщо модифікатор доступу не вказано, то за замовчуванням використовується модифікатор *private*, якщо мова йде про спадкування типу *class*. Якщо успадковується тип *struct*, то при відсутності явно заданого модифікатора доступу за замовчуванням використовується модифікатор *public*.

Якщо базовий клас успадковується як *public*-клас, його *public*-члени стають *public*-членами похідного класу.

У всіх випадках *private*-члени базового класу залишаються закритими в рамках цього класу і не доступні для членів похідного.

Наприклад, в наступній програмі public-члени класу base стають public-членами класу derived. Отже, вони будуть доступні і для інших частин програми.

```
class Base {
int i, j;
public:
void Set (int a, int b) {i = a; j = b; }
void Show () {cout << i << " " << j << " "; }
};

class Derived: public Base {
int k;
public:
Derived (int x) {k = x; }
void ShowK () {cout << k << " "; }
};

int main ()
{
Derived ob (3);
ob.Set (1, 2); // доступ до членів класу base
ob.Show (); // доступ до членів класу base
ob.ShowK (); // доступ до члена класу derived
}
```

Оскільки функції Set() і Show() (члени класу Base) успадковані класом Derived як public-члени, їх можна викликати для об'єкта типу Derived в функції main(). Оскільки члени даних і та j визначені як private-члени, вони залишаються закритими в рамках свого класу Base.

Якщо базовий клас успадковується як private-клас, його public-члени стають private-членами похідного класу.

5.3 Використання захищених членів

Член класу може бути оголошений не тільки відкритим (public) або закритим (private), але і захищеним (protected). Крім того, базовий клас в цілому може бути успадкований з використанням модифікатора protected. Ключове слово protected додано в C ++ для надання механізму спадкування більшої гнучкості.

Закриті члени базового класу не доступні ніяким іншим частинам програми, включаючи і похідні класи. Однак з захищеними членами все інакше. Якщо базовий клас успадковується як public-клас, захищені члени базового класу стають захищеними членами похідного класу, тобто доступними для похідного класу. Отже, використовуючи модифікатор protected, можна створити члени класу, які закриті в рамках свого класу, але які може успадкувати похідний клас, причому з отриманням доступу до них.

Розглянемо наступний приклад програми.

```
class Base {
protected:
 int i, j; // Ці члени закриті в класі base, але доступні
              //для класу derived.
public:
 void Set(int a, int b) { i = a; j = b; }
 void Show() { cout << i << " " << j << " "; }
};

class Derived : public Base {
 int k;
public:
 // Клас derived має доступ до членів класу base і i j.
 void SetK() { k = i * j; }
 void ShowK() { cout << k << ""; }
};

int main()
{
 Derived ob;
 ob.Set(2, 3); // OK, класу Derived це дозволено.
 ob.Show(); // OK, класу Derived це дозволено.
 ob.SetK();
 ob.ShowK();
}
```

При оголошенні члена класу відкритим (з використанням ключового слова public) до нього можна отримати доступ з будь-якої іншої частини програми. Якщо член класу оголошується закритим (за допомогою модифікатора private), до нього можуть отримувати доступ тільки члени того ж класу. Більш того, до закритих членам базового класу не мають доступу навіть похідні класи. Якщо ж член класу оголошується захищеним (protected-членом), до нього можуть отримувати доступ тільки члени того ж або похідних класів. Таким чином, модифікатор protected дозволяє успадковувати члени, але залишає їх закритими в рамках ієрархії класів.

Якщо базовий клас успадковується з використанням ключового слова public, його public-члени стають public-членами похідного класу, а його protected-члени – protected-членами похідного класу.

Якщо базовий клас успадковується з використанням модифікатора protected, його public і protected-члени стають protected-членами похідного класу.

Якщо базовий клас успадковується з використанням ключового слова private, його public і protected-члени стають private-членами похідного класу.

Во всіх випадках private-члени базового класу залишаються закритими в рамках цього класу і не успадковуються.

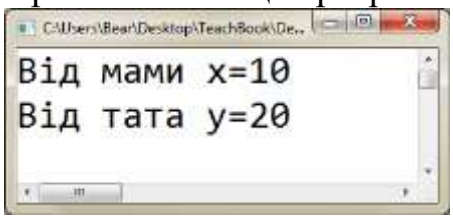
5.4 Множинне спадкування

Похідний клас може успадковувати два або більше базових класів (так зване множинне спадкування). Наприклад, у студента є двоє батьків – мамо і тато. Як відомо, студент має різні властивості від обох батьків. Промоделюємо цю ситуацію за допомогою множинного спадкування.

В цій короткій програмі клас Student успадковує обидва класу Mamo і Tato.

```
// Приклад використання декількох базових класів.
class Mamo {
protected:
int x;
public:
void ShowX () {cout <<"Від мами x="<< x <<endl; }
};

class Tato {
protected:
int y;
public:
void ShowY () {cout <<"Від тата y="<< y <<endl; }
};

// Спадкування двох базових класів.
class Student: public Mamo, public Tato {
public:
void Set(int i, int j) {x = i; y = j; }
};

int main ()
{
setlocale(0, "Ukr");
Student ob;
ob.Set(10, 20); // член класу derived
ob.ShowX(); // функція з класу Mamo
ob.ShowY(); // функція з класу Tato
}
```

При виконанні ця програма генерує такі результати:



На рисунку 5.2 показана ієрархічна схема цих класів.

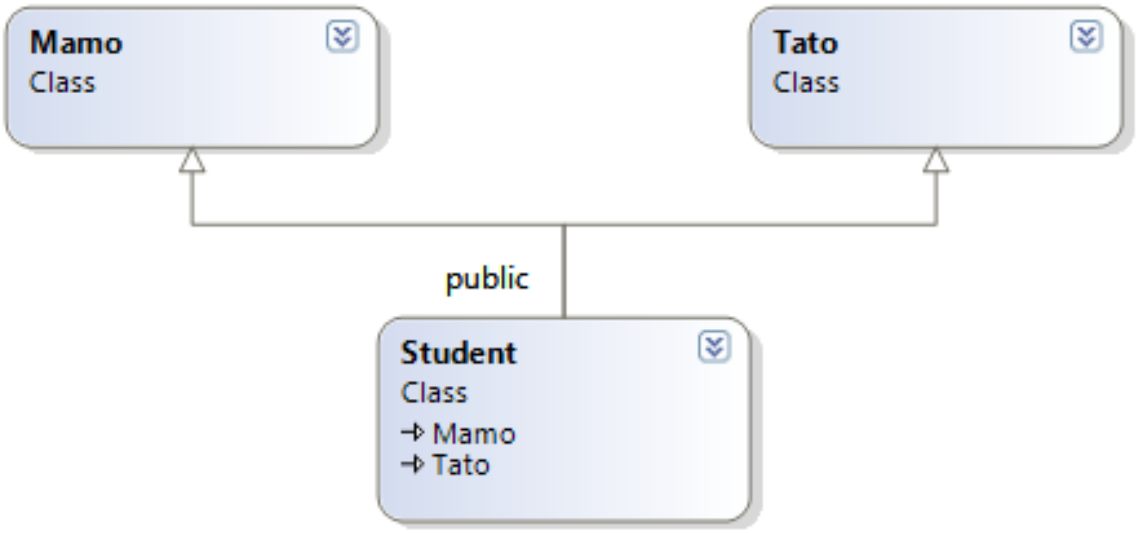


Рис. 5.2. Клас Student успадковує обидва класа Mamo і Tato.

Як видно з цього прикладу, щоб забезпечити спадкування декількох базових класів, необхідно через кому перерахувати їх імена у вигляді списку. При цьому потрібно вказати модифікатор доступу для кожного успадкованого базового класу.

У відомого польського письменника-фантаста Станіслава Лема є оповідання, де герой потрапляє на планету, на якій мешкали п'ятистатеві істоти. У створенні потомства брали участь мама, дада, гага, фафа і хаха. Спробуйте самі змоделювати цю ситуацію, використовуючи механізм множинного спадкування.

5.5 Віртуальні базові класи

При наслідуванні декількох базових класів в програму може бути внесений елемент невизначеності. У попередньому прикладі студент є спадкоємцем двох класів – мамо і тато, але кожен із цих класів, у свою чергу, є спадкоємцем класу людина (Human). Така схема успадкування зображена на діаграмі (рис. 5.3) і носить назву «ромбічне успадкування».

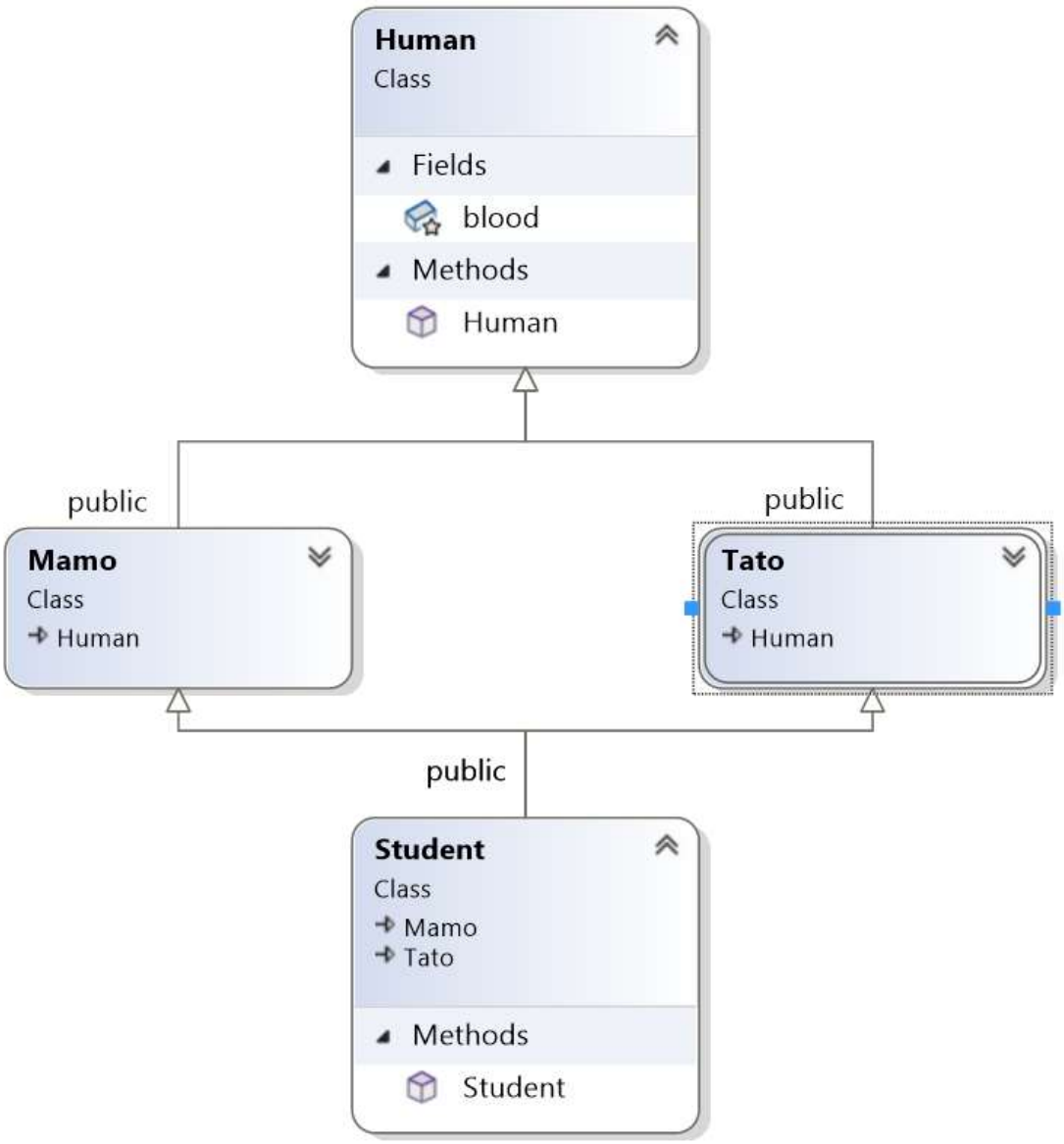


Рис. 5.3. Схема успадкування на діаграмі класів.

Розглянемо наступний приклад:

Кожна людина має певну групу крові, це змінна blood в базовому класі Human. Клас Mamo і клас Tato успадковують різні крупи крові, бо вони – різні люди і в кожної з них тече своя кров. Але яку групу крові отримає від своїх батьків наш студент? Поки на це питання немає відповіді і це показано в наступної програмі:

```
#include <time.h>
using namespace std;
// Приклад використання декількох базових класів.

class Human
{
protected:
 int blood;
public:
```

```
 Human()
 {
       blood = rand() % 4;
       cout << "З'явилася людина, її група крові " <<blood<< endl;
 }
};

class Tato : public  Human //
{
public:
 Tato()
 {
       cout << "Це тато, його група крові "<<blood << endl;
 }
};

class Mamo : public  Human //
{
public:
 Mamo()
 {
       cout << "Це мамо, її група крові " << blood << endl;
 }
};

class Student : public Tato, public Mamo
{
public:
 Student()
 {
       cout << "Студент  народився. Група крові ???" << endl;
 }
};

int main()
{
 setlocale(0, "");
 srand((unsigned)time(0));
    Student vasya;
}
```

Результат роботи програми:

```
З'явилася людина, її група крові 2
Це тато, його група крові 2
З'явилася людина, її група крові 0
Це мамо, її група крові 0
Студент  народився. Група крові ???
```

Однак, коли ми спробуємо змінити код, щоб вивести групу крові студента, отримаємо таку помилку:

```
Student()
{
    cout << "Студент  народився. Група крові "<<blood << endl;
}
```

abc	E0266	"Student::blood" is ambiguous
⊗	C2385	ambiguous access of 'blood'

(Помилка C2385 неоднозначний доступ до blood)

Обидва класи Mamo і Tato наслідують клас Human. Але клас Student успадковує як клас Mamo, так і клас Tato. В результаті в об'єкті типу Student присутні дві копії класу Human, в такому виразі

```
cout << "Студент  народився. Група крові "<<blood << endl;
```

не ясно, на яку саме копію змінної blood тут дано посилання: на змінну blood, успадковану від класу Mamo або від класу Tato? Оскільки в класі Student присутні обидві копії класу Human, то в ньому існують і два члена blood!

Тому-то ця інструкція і є невизначеною. Є два способи виправити попередню програму. Перший полягає в застосуванні оператора дозволу контексту (дозволу області видимості), за допомогою якого можна «вручну» вказати потрібну змінну blood:

```
Student()
{
cout << "Студент  народився. Група крові "<<Mamo::blood << endl;
}
```

Наразі невизначеність ліквідована і студент завжди успадковує свою групу крові від матері.

```
З'явилася людина, її група крові 0
Це тато, його група крові 0
З'явилася людина, її група крові 2
Це мамо, її група крові 2
Студент  народився. Група крові 2
```

Застосування оператора "::" дозволяє програмі «ручним способом» вибрати версію класу Human (успадковану класами Mamo і Tato). Але після такого рішення виникає цікаве питання: а що, якщо в дійсності потрібна тільки одна копія класу Human? Чи можна якимось чином запобігти включення двох копій в клас Student? Відповідь, як, напевно, ви здогадалися, позитивна. Це рішення досягається за допомогою віртуальних базових класів.

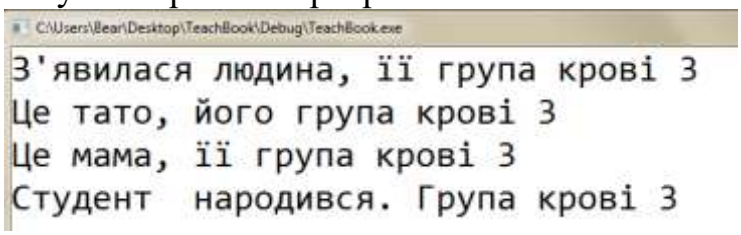
Якщо два (або більше) класа є нащадками загального базового класу, ми можемо запобігти включення декількох його копій в об'єкти цих похідних класів, що реалізується шляхом оголошення базового класу при його спадкуванні віртуальним. Для цього достатньо випередити ім'я успадкованого базового класу ключовим словом virtual.

Для ілюстрації цього процесу наведемо ще одну версію попередньої програми. На цей раз клас Student містить тільки одну копію класу Human.

```
// Ця програма використовує віртуальні базові класи.
class Human
{
protected:
 int blood;
public:
 Human()
 {
      blood = rand() % 4;
      cout << "З'явилася людина, її група крові " <<blood<< endl;
 }
};

class Tato : virtual public  Human //
{
public:
 Tato()
 {
      cout << "Це тато, його група крові "<<blood << endl;
 }
};

class Mamo : virtual public  Human //
{
public:
 Mamo()
 {
      cout << "Це мама, її група крові " << blood << endl;
 }
};

class Student : public Tato, public Mamo
{
public:

 Student()
 {
      cout << "Студент  народився. Група крові "<<blood << endl;
 }
};
```

```
int main()
{
 setlocale(0, "");
 Student vasya;
}
```

Результат роботи програми:



Тепер обидва класи Mamo і Tato успадковують клас Human як віртуальний, і тому при будь-якому множинному наслідуванні в похідний клас буде включена лише одна його копія. Отже, в класі Student присутня лише одна копія класу Human, а інструкція

```
cout << "Студент  народився. Група крові "<<blood << endl;
```

тепер абсолютно допустима і не містить ніякої неоднозначності. Але зверніть увагу, що тепер і мамо і тато – це не дві різних людини, а всього лише одна. Що ж, таке трапляється після довгих років спільного сімейного життя.

Різниця між звичайним базовим і віртуальним класами стає очевидною тільки тоді, коли цей базовий клас успадковується більше одного разу. (Рис. 5.3.) Якщо базовий клас оголошується віртуальним, то тільки один його екземпляр буде включений в об'єкт похідного класу. В іншому випадку в цьому об'єкті буде присутній кілька його копій.

5.6 Залежність

Розглянемо ще один вид відносин між класами – залежність.

Залежність виникає тоді, коли реалізація класу одного об'єкта залежить від специфікації операцій класу іншого об'єкта. І якщо зміниться специфікація операцій цього класу, нам неминуче доведеться вносити зміни і в залежний клас.

Наприклад, операція «Відтворення», що реалізується класом-медіаплеєром, залежить від операції «Декомпресія», що реалізовується класом кодек. Якщо специфікація операції «Декомпресія» зміниться, доведеться змінювати код медіаплеєра.

Ось так залежність між класами зображується в UML (Рис. 5.4)

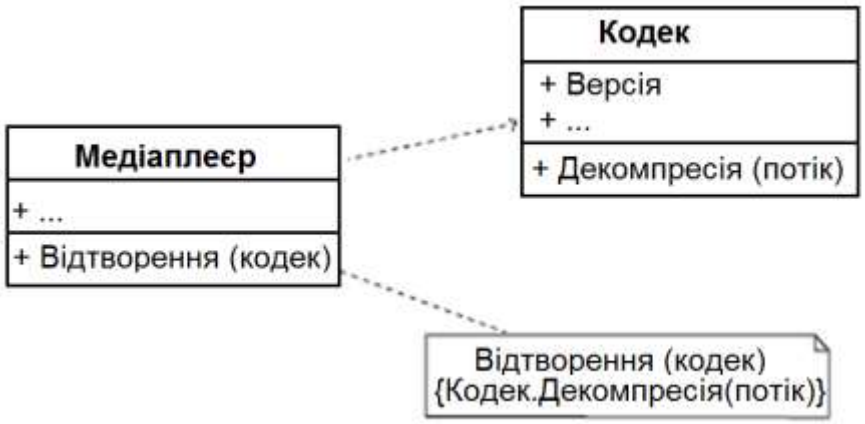


Рис. 5.4. Залежність між класами.

Варто відзначити, що залежності на діаграмах зображують далеко не завжди, а тільки в тих випадках, коли їх відображення є важливим для розуміння моделі. Часто залежності лише маються на увазі, бо логічно випливають з природи класів.

5.7 Асоціація. Агрегація і композиція

Інший вид відносин між об'єктами – це асоціація. Це просто зв'язок між об'єктами, по якому можна між ними переміщатися (Рис. 5.5).

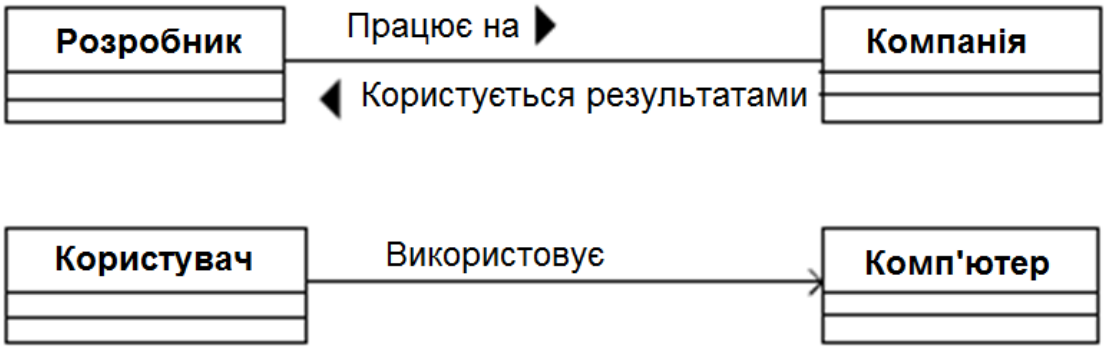


Рис. 5.5. Зв'язок між об'єктами типу асоціація.

Асоціація може мати ім'я, яке показує природу відносин між об'єктами, при цьому в імені може вказуватися напрям читання зв'язку за допомогою трикутного маркера. Односпрямована асоціація може зображуватися стрілкою.

Крім напряму асоціації, ми можемо вказати на діаграмі ролі, які кожен клас відіграє в даному відношенні, і кратність, тобто кількість об'єктів, пов'язаних відношенням (Рис. 5.6).

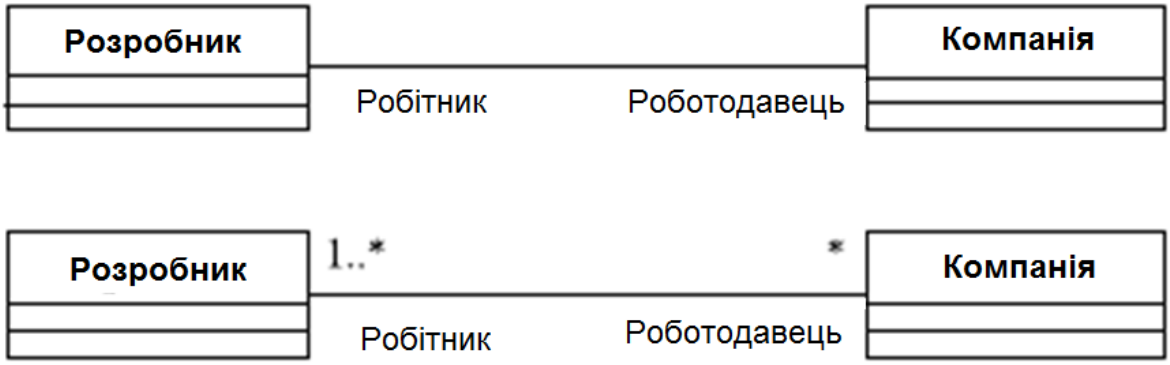


Рис. 5.6. Асоціація. Ролі і кратність.

Насправді, в реальності зв'язки бувають «просто зв'язками» вкрай рідко. Зазвичай при найближчому розгляді під асоціацією розуміється більш складне відношення між класами, наприклад, зв'язок типу «частина-ціле».

Такий вид асоціації називається асоціацією з агрегуванням. У цьому випадку один клас має більш високий статус (ціле) і складається з нижчих за статусом класів (частин).

При цьому виділяють просте і композитне агрегування і говорять про агрегацію і композицію.

Проста агрегація передбачає, що частини, відділені від цілого, можуть продовжувати своє існування незалежно від нього.

Під композитним ж агрегуванням розуміється ситуація, коли ціле володіє своїми частинами і їх час життя відповідає часу життя цілого, тобто незалежно від цілої частини існувати не можуть.

На рисунку 5.7 показано відношення типу агрегація і композиція з використанням зафарбованого та не зафарбованого ромбика.

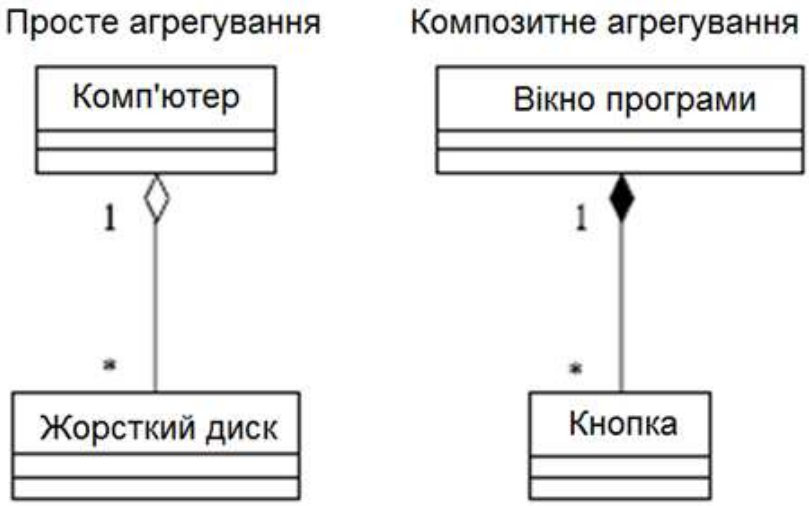


Рис. 5.7. Просте і композитне агрегування (агрегація і композиція).

Наступний програма показує приклад агрегації і композиції:

```
// Приклад агрегації і композиції
class Brain
{
public:
	int Volume;
	void Think()
	{
		cout << "я мислю, отже, я існую" << endl;
	}
};

class Human
{
public:
	int year_of_birth;
	Brain brain;   // містить мозок
	void Think()
	{
brain.Think(); // людина сама по собі думати не може, думає мозок
	}
};

class Student : public Human
{
public:
	string pib;
	string arrdess;
	void Print()
	{
		cout << "Студент " << pib << " каже: ";
		brain.Think();
	}
};

class Group
{
	Student* p;
	int kilkist;
public:
	Group(int kilkist)
	{
		this->kilkist = kilkist;
		p = new Student[kilkist];
	}

	~Group()
	{
		delete[]p;
	}
```

```
        Student& operator [] (int i)
        {
                return p[i];
        }

    };

int main()
{
        setlocale(0, "");
        Group group(4);
        group[0].pib = "Кір Стармер";
        group[1].pib = "Дональд Трамп";
        group[2].pib = "Емануель Макрон";
        group[3].pib = "Фрідріх Мерц";
        for (int i = 0; i < 4; i++)
                group[i].Print();
       _getch();
}
```

Результат роботи програми:

```
C:\Users\Yura\Desktop\Лекції ООП 2025\OOP_CPP\x64\Debug\OOP_CPP.exe
Студент Кір Стармер каже: я мислю, отже, я існую
Студент Дональд Трамп каже: я мислю, отже, я існую
Студент Емануель Макрон каже: я мислю, отже, я існую
Студент Фрідріх Мерц каже: я мислю, отже, я існую
```

Кожна людина має мозок, але мозок не може функціонувати без людини. Тому клас Brain (мозок) є композитним для класу Human (людина).

Людина сама по собі думати не може, думає мозок. Коли викликається метод Think (думати) для об'єкта класу Human або успадкованого від нього класу Student, цей метод викликає метод Think (думати) у мозку (в класі Brain). Така ситуація називається «делегування» і це, до речі, один із паттернів проектування.

Кожен студент є людиною, тобто нащадком класу Human. Це відношення типа спадкування. Крім того, студент є членом групи, де може бути кілька студентів, але студент може існувати і без групи. Тому клас Student є агрегованим для класу Group (група).

На рис. 5.8. показані відносини між цими класами.

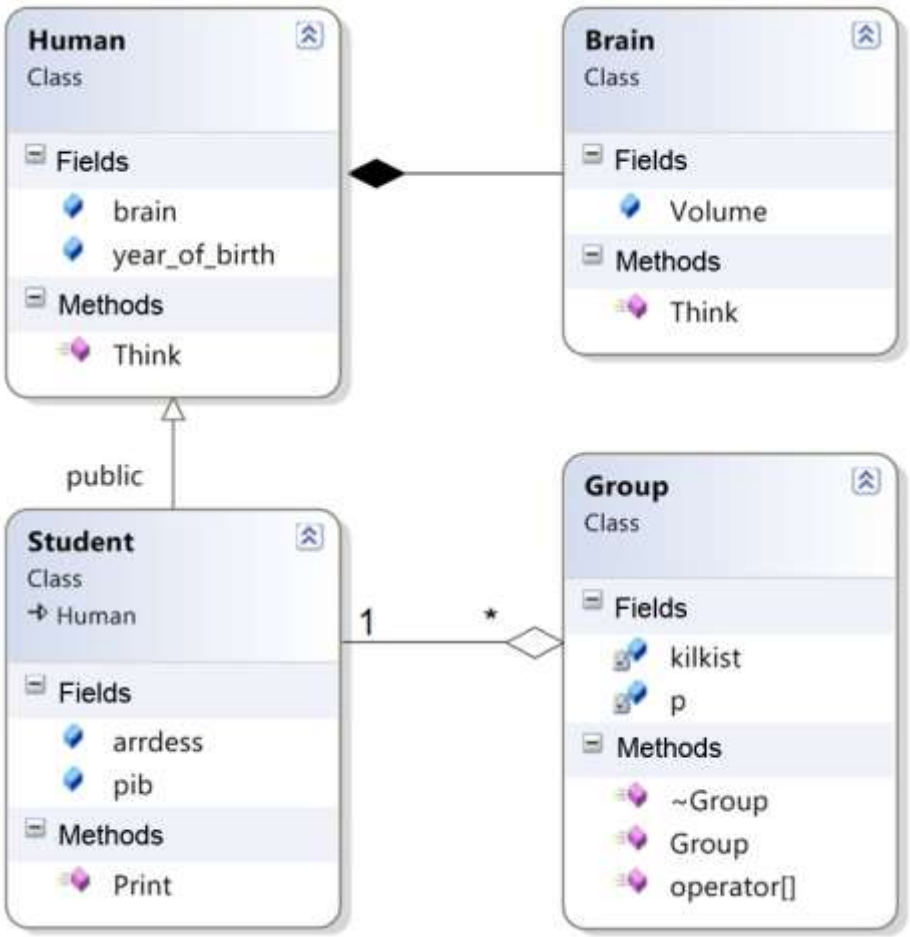


Рис. 5.8. Відносини між класами типа спадкування, агрегація і композиція.

5.8 Створення діаграм класів засобами Visual Studio

Для створення діаграм класів у Visual Studio, відкрийте вікно Solution Explorer (рисунок 5.9), виберіть рядок з іменем проекту і далі, клацнувши правою кнопкою «миші» увійдіть у вікно додавання нового елемента у проект.

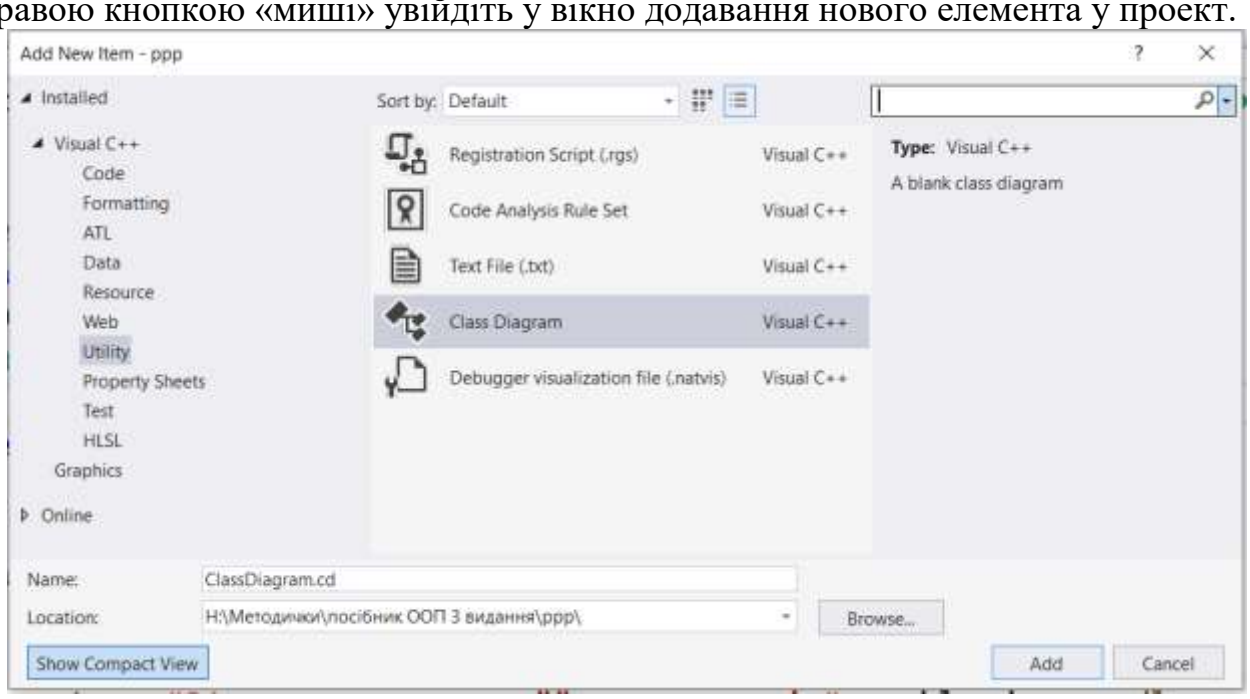


Рис. 5.9. Додавання діаграми класів у проект

У вікні Solution Explorer оберіть рядок з іменем проекту і далі натисніть View->View Class Diagram (рисунок 5.10).

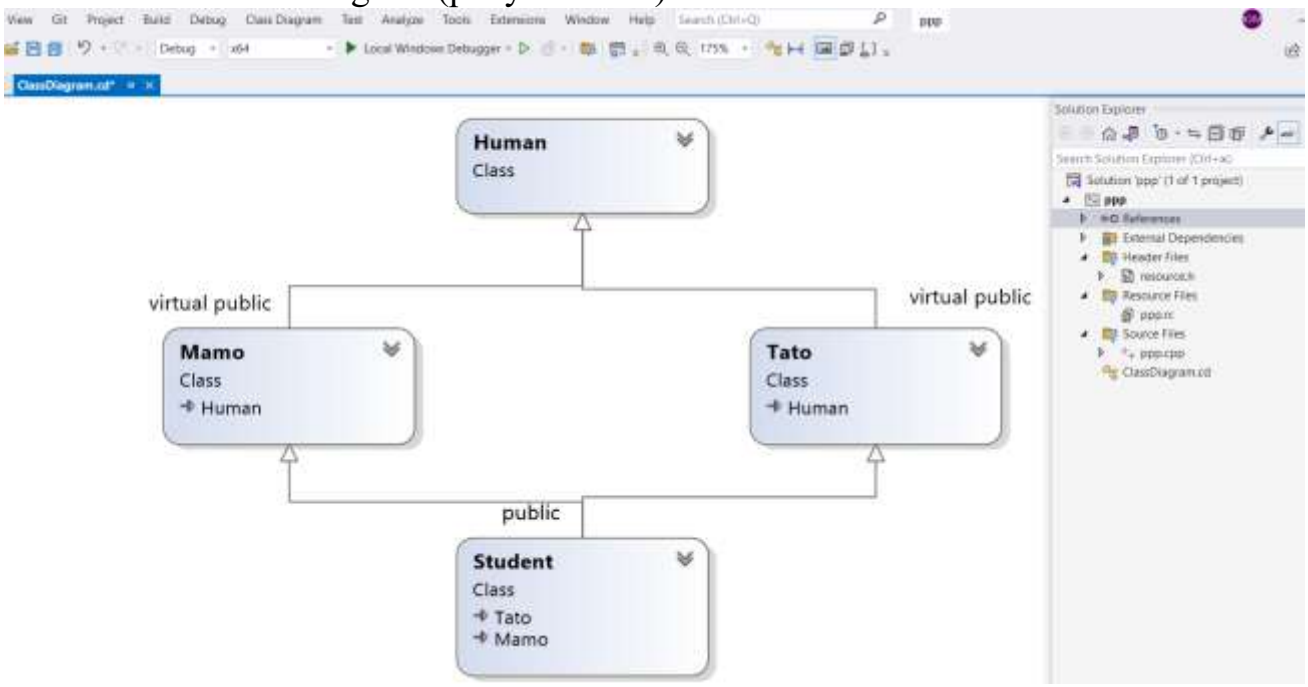


Рис. 5.10. Вигляд вікна Visual Studio для діаграми класів

Діаграму класів можна відредагувати. Розкриємо показ усіх членів всіх класів проекту (рисунок 5.11)

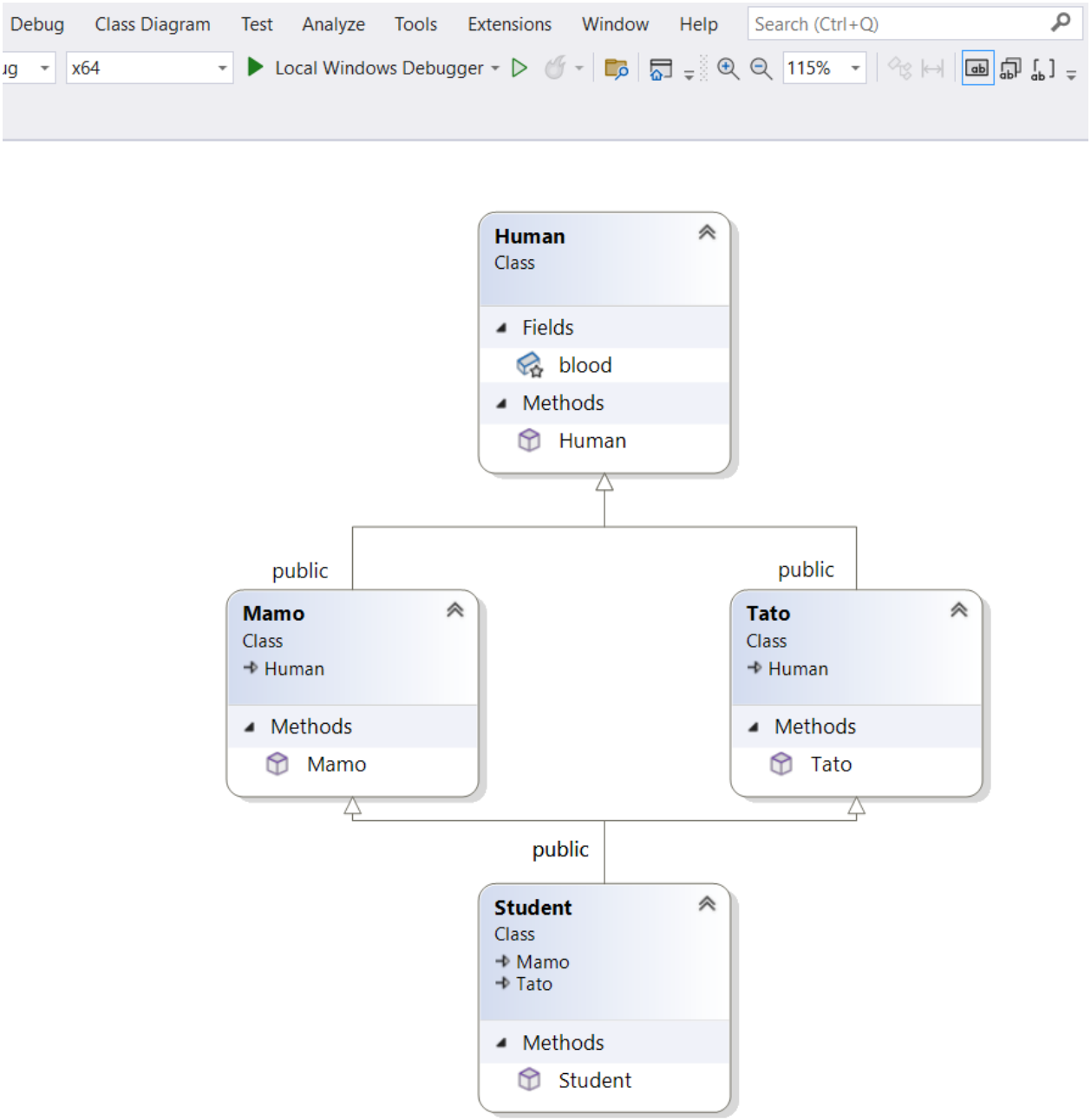


Рис. 5.11. Вигляд діаграми класів із деталізацією показу усіх членів класів

Резюме

1. Використовуючи спадкування, можна створити клас, який визначає найбільш загальні характеристики всіх пов'язаних з ним елементів. Цей клас потім може бути успадкований іншими, спеціалізованими класами з додаванням в кожен з них своїх, унікальних особливостей.

2. Якщо один клас успадковує інший, члени базового класу стають членами похідного. Статус доступу членів базового класу в похідному класі визначається модифікатором доступу, що використовуються для спадкування базового класу.

3. Використовуючи модифікатор protected, можна створити члени класу, які закриті в рамках свого класу, але які може успадкувати похідний клас, причому з отриманням доступу до них.

4. Якщо базовий клас успадковується з використанням ключового слова public, його public-члени стають public-членами похідного класу, а його protected-члени – protected-членами похідного класу.

5. Якщо базовий клас успадковується з використанням модифікатора protected, його public і protected-члени стають protected-членами похідного класу.

6. Якщо базовий клас успадковується як private-клас, його public-члени стають private-членами похідного класу.

7. Похідний клас може успадковувати два або більше базових класів (так зване множинне спадкування).

8. Якщо базовий клас оголошується віртуальним, то тільки один його екземпляр буде включений в об'єкт похідного класу. Різниця між звичайним базовим і віртуальним класами стає очевидною тільки тоді, коли цей базовий клас успадковується більше одного разу.

9. Зв’язок між класами називається асоціацією з агрегуванням, коли один клас має більш високий статус (ціле) і складається з нижчих за статусом класів (частин).

10. Проста агрегація передбачає, що частини, відділені від цілого, можуть продовжувати своє існування незалежно від нього.

11. Під композитним агрегуванням розуміється ситуація, коли ціле володіє своїми частинами, які незалежно від цілої частини існувати не можуть.

Завдання до лабораторної роботи

1. Визначити ієрархію класів (відповідно до варіанта). Для визначення ієрархії класів зв'язати ставленням спадкування класи, наведені для заданого варіанта, з цих класів обрати один, який буде стояти на чолі ієрархії.

2. Реалізувати класи. Визначити в класах всі необхідні конструктори і деструктор.

3. Написати демонстраційну програму, в якій створюються об'єкти різних класів.

4. Побудувати діаграму класів.

Варіанти завдань:

1. студент, викладач, персона, зав кафедрою;
2. службовець, персона, робітник, інженер;
3. робочий, кадр, інженер, головний інженер;
4. дуб, рослина, дерево, очерет;
5. організація, страхова компанія, будівна компанія, метробуд;
6. журнал, книга, друковане видання, посібник;
7. тест, іспит, випускний іспит, випробування;
8. місце, область, місто, мегаполіс;
9. іграшка, продукт, товар, молочний продукт;
10. квитанція, накладна, документ, чек;
11. автомобіль, поїзд, транспортний засіб, таксі;
12. двигун, двигун внутрішнього згоряння, дизель, турбореактивний двигун;
13. республіка, монархія, королівство, держава;
14. ссавці, парнокопитні, птахи, тварина;
15. судно, пароплав, вітрильник, бригантина.

Тести для закріплення матеріалу

1. Простим називається спадкування, при якому:
   а) похідний клас має тільки члени Дані, не має членів-методів;
   б) похідний клас має один базовий клас;
   в) похідний клас має тільки один конструктор;
   г) базовий клас не має віртуальних методів.

2. Якщо в базових класів є загальний предок, то похідний від цих базових класів клас успадковує два примірники полів предка. Щоб уникнути цього, треба
   а) використовувати конструктори з різними параметрами;
   б) визначити клас предка як статичний;
   в) при спадкуванні загального предка визначити його як віртуальний клас;
   г) використовувати оператор :: для доступу до екземплярів предка

3. Якщо при множині наслідуванні в базових класах є однойменні елементи і конфлікт ідентифікаторів, він усувається за допомогою операції
   а):
   б) ->
   в) &
   г) ::

4. Чи можна використовувати модифікатори доступу в представленому нижче прикладі?
```
class D: A, protected B, public 3 {}
```
   а) можна було б тільки у випадку одного базового класу;
   б) можна використовувати один модифікатор для всіх базових класів;
   в) не можна;
   г) можна.
5. Якщо записано
```
class A {public: void f () {cout << 1; }};
class B: public A
  {
     public: void f () {cout << 2; }
  };
```
   то що буде надруковано в результаті виконання коду?
```
B b; A & a = b; a.f ();
```
   а) 1
   б) 2 1
   в) 2
   г) 1 2
6. Скільки похідних класів можна отримати з базового класу?
   а) необмежену кількість
   б) кількість похідних класів визначається кількістю базових класів
   в) визначається наявністю абстрактного класу
   г) визначається розміром стеку

7. Який правильний виклик функції базового класу з об'єкта похідного класу, якщо в похідному класі ця функція була заміщена?
   а) base.Function Name ();
   б) base :: Function Name ();
   в) FunctionName ();
   г) таку функцію викликати не можна.

8. Прокоментуйте код: class A: public B {}
   а) в конструкторі класу А ініціалізується член В;
   б) клас А містить в секції public член В;
   в) клас А похідний від базового класу В;
   г) інше.

РОЗДІЛ 6. ВІРТУАЛЬНІ ФУНКЦІЇ І ПОЛІМОРФІЗМ

Одним з трьох основних принципів об'єктно-орієнтованого програмування є поліморфізм. Стосовно до C++ поліморфізм є термін, який використовується для опису процесу, в якому різні реалізації функції можуть бути доступні за допомогою одного і того ж імені. З цієї причини поліморфізм іноді характеризується фразою «один інтерфейс, багато методів». Це означає, що до всіх функцій-членів загального класу можна отримати доступ одним і тим же способом, незважаючи на можливі відмінності в конкретних діях, пов'язаних з кожною окремою операцією.

У C++ поліморфізм підтримується як під час виконання, так в період компіляції програми. Перевантаження операторів і функцій – це приклади поліморфізму, що відноситься до часу компіляції. Але, не дивлячись на могутність механізму перевантаження операторів і функцій, він не в змозі вирішити всі задачі, які виникають в реальних застосуваннях об'єктно-орієнтованої мови програмування. Тому в C++ також реалізований поліморфізм під час виконання на основі використання похідних класів і віртуальних функцій.

6.1 Вказівники на похідні типи

Фундаментом для динамічного поліморфізму служить вказівник на базовий клас. Вказівники на базові і похідні класи пов'язані такими відносинами, які не властиві вказівникам інших типів. Вказівник одного типу, як правило, не може вказувати на об'єкт іншого типу. Однак вказівники на базові класи і об'єкти похідних класів – виключення з цього правила. У C ++ вказівник на базовий клас також можна використовувати для посилання на об'єкт будь-якого класу, успадкованого від базового. Припустимо, що у нас є базовий клас Animal і класи Cat, God, Frog і Cow, які успадковані від класу Animal. (Рис. 6.1.) У C++ вказівник, що був оголошений як вказівник на базовий клас Animal, може бути також вказівником на похідні класи Cat, God, Frog і Cow.

```
Animal *A[4]; // вказівники на базовий клас
Cat B; Dog C; Frog D; Cow E;
A[0] = &B; // вказівник на базовий клас посилається на
//об'єкт класу Cat
A[1] = &C; // вказівник на базовий клас посилається на
//об'єкт класу Dog
A[2] = &D; // вказівник на базовий клас посилається на
//об'єкт класу Frog
A[3] = &E; // вказівник на базовий клас посилається на
//об'єкт класу Cow
```

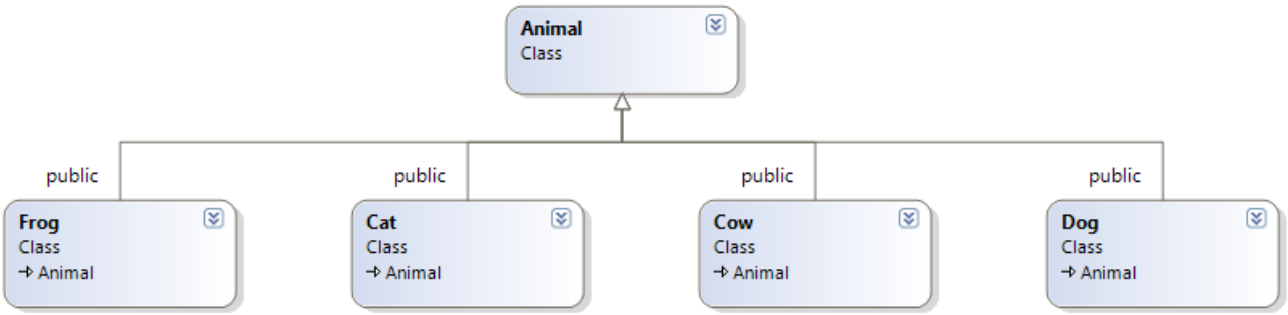


Рис. 6.1. Класи Cat, God, Frog і Cow успадковані від класу Animal.

Крім того, необхідно розуміти, що хоча «базовий» вказівник можна використовувати для доступу до об'єктів будь-якого похідного типу, зворотне твердження не вірно. Іншими словами, використовуючи вказівник на похідний клас, не можна отримати доступ до об'єкта базового типу. На рис. 6.2. зображено вікно повідомлень про помилки під час компіляції програми.

```
Cat* P;
Animal A;
P = &A;
```

Error List

Entire Solution | 2 Errors | 0 Warnings | 0 of 1 Message

Code	Description
E0513	a value of type "Animal *" cannot be assigned to an entity of type "Cat *"
C2440	'=': cannot convert from 'Animal *' to 'Cat *'

Рис. 6.2. Помилки під час компіляції програми.

Той факт, що вказівник на базовий тип можна використовувати для посилання на будь-який об'єкт, успадкованій від базового, надзвичайно важливий і принциповий для C++. Як буде показано нижче, ця гнучкість є ключовим моментом для способу реалізації динамічного поліморфізму в C ++.

Подібно вказівниками, *посилання* на базовий клас також можна використовувати для доступу до об'єкта похідного типу. Ця можливість особливо часто застосовується при передачі аргументів функцій. Параметр, який має тип посилання на базовий клас, може приймати об'єкти базового класу, а також об'єкти будь-якого іншого типу, успадкованого від нього.

6.2 Віртуальні функції

Динамічний поліморфізм можливий завдяки поєднанню двох засобів: спадкування і віртуальних функцій.

Віртуальна функція – це функція, яка оголошується в базовому класі з використанням ключового слова virtual і перевизначається в одному або декількох похідних класах. Таким чином, кожен похідний клас може мати власну версію віртуальної функції.

Коли віртуальна функція викликається через вказівник (або посилання) на базовий клас, C++ визначає, яку саме версію віртуальної функції потрібно викликати, по типу об'єкта, що адресується цим вказівником. Причому слід мати на увазі, що це рішення приймається під час виконання програми. Отже, при посиланні на різні об'єкти будуть викликатися і різні версії віртуальної функції. Саме за типом об'єкта, що адресується, (а не по типу самого вказівника) визначається, яка версія віртуальної функції буде виконана. Таким чином, якщо базовий клас містить віртуальну функцію і якщо з цього базового класу успадковані два (або більше) інших класу, то при адресації різних типів об'єктів через вказівник на базовий клас будуть виконуватися і різні версії віртуальної функції. Аналогічний механізм працює і при використанні посилання на базовий клас.

Щоб оголосити функцію віртуальної, досить випередити її оголошення ключовим словом virtual.

Функція оголошується віртуальною в базовому класі за допомогою ключового слова virtual. При перевизначені віртуальної функції в похідному класі ключове слово virtual повторювати не потрібно (хоча це не буде помилкою).

Атрибут virtual передається «у спадок».

Якщо функція оголошується як віртуальна, вона залишається такою незалежно від того, через скільки рівнів похідних класів вона може пройти.

Якщо віртуальна функція перевизначається в похідному класі, її називають перевизначеною.

Прототипи віртуальної функції і її перевизначень повинні бути абсолютно однаковими. Якщо прототипи будуть різними, то така функція буде просто вважатися перевантаженою, і її «віртуальна сутність» втратиться.

І ще одне важливе зауваження: *конструктор не успадковується*. Конструктор базового класу створює об’єкт, який «живе» в об'єкті похідного класу. Тому деструкторам дозволяється бути віртуальними, а конструкторам – ні!

```
class Cat : public
{
public:
    virtual ~Cat()
    {
    }
```

```
class Cat : public Animal
{
public:
    virtual Cat()
    {
    }
```

Вище показано два фрагмента коду, зліва правильний, а справа - з помилкою.

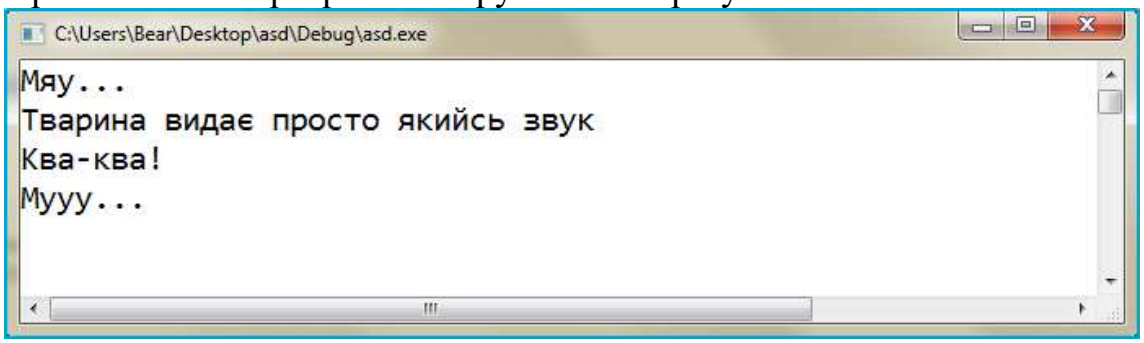
Клас, який включає віртуальну функцію, називається поліморфним класом. Цей термін також застосовується до класу, який успадковує базовий клас, що містить віртуальну функцію.

Розглянемо наступну коротку програму, в якій демонструється використання віртуальних функцій.

```
class Animal
{
public:
 virtual void GetSound(){
 cout<<"Тварина видає просто якийсь звук "<<endl;
 }
};

class Cat : public Animal
{
public:
 void GetSound() // Перевизначення функції GetSound()
                  //для класу Cat
 {
      cout<<"Мяу..."<<endl;
 }
};
class Dog : public Animal
{
public:
 /*void GetSound() // Перевизначення функції GetSound()
                    //для класу Dog закоментоване
 {
      cout<<"Гав!"<<endl;
 }*/
};
class Frog : public Animal
{
public:
 void GetSound() // Перевизначення функції GetSound()
                  //для класу Frog
 {
      cout<<"Ква-ква!"<<endl;
 }
};

class Cow : public Animal
{
public:
 void GetSound() // Перевизначення функції GetSound()
                         //для класу Cow
 {
      cout<<"Муууу..."<<endl;
 }
};

int main()
{
setlocale(0,"Ukr");
Animal *A[4]; // вказівники на базовий клас
Cat B; Dog C; Frog D; Cow E;
```

```
A[0] = &B; // вказівник на базовий клас посилається на
//об'єкт класу Cat
A[1] = &C; // вказівник на базовий клас посилається на
//об'єкт класу Dog
A[2] = &D; // вказівник на базовий клас посилається на
//об'єкт класу Frog
A[3] = &E; // вказівник на базовий клас посилається на
//об'єкт класу Cow
for(int i=0; i<4; i++)
 A[i]->GetSound(); // поліморфний виклик!!!
}
```

При виконанні програма генерує ось такі результати:



Якщо похідний клас не перевизначає віртуальну функцію, то використовується функція, що визначена в базовому класі.

Як бачимо, в похідному класі Dog функція `GetSound()` не перевизначається и тоді викликається функція базового класу.

Поліморфізм забезпечує можливість деякому узагальненому класу визначати функції, які будуть використовувати всі похідні від нього класи, причому похідний клас може визначити власну реалізацію деяких або всіх цих функцій. Іноді ця ідея виражається в такий спосіб: базовий клас диктує загальний інтерфейс, який буде мати будь-який об'єкт, успадкований від цього класу, але дозволяє при цьому похідному класу визначити метод, який використовується для реалізації цього інтерфейсу. Ось чому для опису поліморфізму часто використовується фраза «один інтерфейс, багато методів».

Чому ж так важливий загальний інтерфейс з безліччю реалізацій? Відповідь знову повертає нас до основної причини виникнення об'єктно-орієнтованого програмування: такий інтерфейс дозволяє програмісту впоратись з дедалі більшою складністю програм. Наприклад, якщо коректно розробити програму, то можна бути впевненим в тому, що до всіх об'єктів, успадкованим від базового класу, можна буде отримати доступ єдиним (загальним для всіх) способом, не дивлячись на те, що конкретні дії у одного похідного класу можуть відрізнятися від дій у іншого.

Розробнику похідного класу не потрібно заново винаходити елементи, вже існуючі в базовому класі. Більш того, відділення інтерфейсу від реалізації дозволяє створювати бібліотеки класів, написанням яких можуть займатися сторонні організації. Коректно реалізовані бібліотеки повинні надавати загальний інтерфейс, який програміст може використовувати для виведення

класів відповідно до своїх конкретних потреб. Таким чином можна визначити поліморфізм як один інтерфейс для безлічі реалізацій (Рис. 6.3)

Рис. 6.3. Поліморфізм – це один інтерфейс для безлічі реалізацій.

Коли потрібен віртуальний деструктор

Розглянемо приклад із застосуванням динамічної пам’яті. Конструктор базового класу (Parent) виділяє динамічну пам’ять, а деструктор її звільняє. Від класу Parent успадковується похідний клас Child, його конструктор теж виділяє динамічну пам’ять, а деструктор її звільняє. Так саме робить конструктор і деструктор класу SubChild, похідний від класу Child. Відносини між цими класами показані на рис. 6.4.

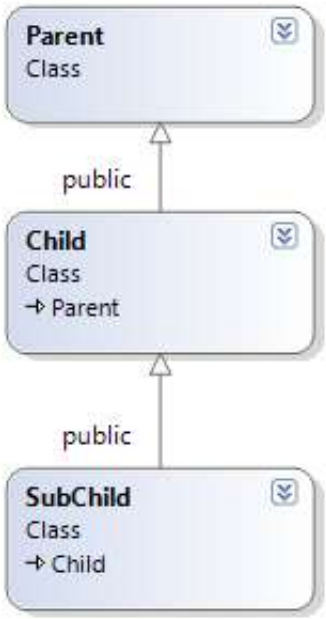
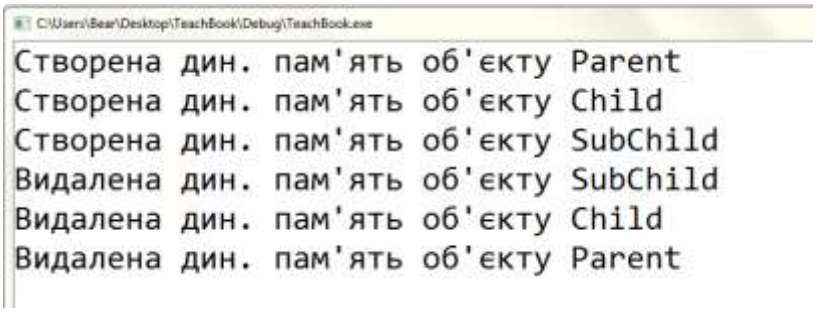


Рис. 6.4. Відносини між класами з віртуальними деструкторами.

У наведеному тексті програми «віртуальність» деструктора базового класу успадковується деструкторами похідних класів:

```
class Parent
{
public:
 Parent()
 {
      cout << "Створена дин. пам'ять об'єкту Parent" << endl;
 }
 virtual ~Parent()
 {
      cout << "Видалена дин. пам'ять об'єкту Parent" << endl;
 }
};
class Child : public Parent
{
public:
 Child()
 {
      cout << "Створена дин. пам'ять об'єкту Child" << endl;
 }
 ~Child()
 {
      cout << "Видалена дин. пам'ять об'єкту Child" << endl;
 }
};
class  SubChild : public Child
{
public:
 SubChild()
 {
      cout << "Створена дин. пам'ять об'єкту SubChild" << endl;
 }
 ~SubChild()
 {
      cout << "Видалена дин. пам'ять об'єкту SubChild" << endl;
 }
};

int main() {
 setlocale(0, "");
 Parent *P2 = new SubChild();
 delete P2;
}
```

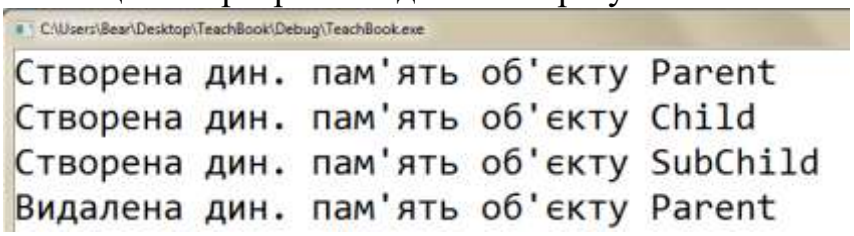
І результат отримуємо такий:



А тепер приберемо ключове слово virtual для деструктора класу Parent:

```
virtual ~Parent()
{
    cout << "Видалена дин. пам'ять об'єкту Parent" << endl;
}
```

Після цього програма видає такий результат:



Тобто динамічна пам’ять звільняється не повністю. Це випадок, коли деструктор обов'язково має бути віртуальним.

6.3 Суто віртуальні функції та абстрактні класи

У багатьох випадках взагалі немає сенсу давати визначення віртуальної функції в базовому класі. Наприклад, в базовому класі Animal (з попереднього прикладу) визначення функції GetSound() – це просто вивід тексту без особливого сенсу, тому, що неможливо собі уявити, який звук видає просто тварина.

Конкретний звук завжди видає конкретна тварина, тому функцію GetSound() обов'язково треба перевизначити для всіх похідних класів.

Суто віртуальна функція (pure virtual function) – це віртуальна функція, яка не має визначення в базовому класі.

Суто віртуальна функція – це функція, яка оголошена в базовому класі, але не має в ньому ніякого визначення. Тому будь-який похідний тип повинен визначити власну версію цієї функції, адже у нього просто немає ніякої можливості використовувати версію з базового класу (через її відсутність). Щоб оголосити суто віртуальну функцію, використовується наступний загальний формат:

```
virtual тип ім’я_функції (список_параметрів) = 0;
```

Тут під елементом тип мається на увазі тип значення, що повертається функцією, а елемент ім’я_функції- її ім'я. Позначення = 0 є ознакою того, що функція тут оголошується як суто віртуальна.

Оголосивши функцію суто віртуальною, програміст створює умови, при яких похідний клас просто змушений мати визначення власної її реалізації. Без цього компілятор видасть повідомлення про помилку.

```
class Animal
{
public:
     virtual void GetSound() = 0; // Суто віртуальна функція
};
class Cat : public Animal
{
public:
void GetSound() // Перевизначення функції GetSound() для класу Cat
     {
          cout << "Мяу..." << endl;
     }
};
class Dog : public Animal
{
public:
/*void GetSound() // Перевизначення функції GetSound() закоментоване

     {
          cout<<"Гав!"<<endl;
     }*/
};

int main()
{
     setlocale(0, "Ukr");
     Animal* a = new Dog();
}
```

Наприклад, при компіляції програми, представлений вище, з'являється ось таке вікно повідомлень (Рис. 6.5):

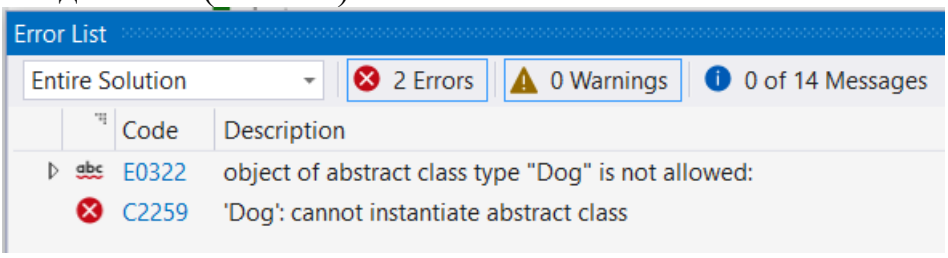


Рис. 6.5. Повідомлення під час компіляції програми.

Якщо клас має хоча б одну суто віртуальну функцію, його називають абстрактним. Абстрактний клас характеризується однією важливою особливістю: у такого класу не може бути об'єктів. Абстрактний клас можна використовувати тільки в якості базового, з якого будуть успадковуватися інші

класи. Причина того, що абстрактний клас не можна використовувати для створення об'єктів, лежить, безумовно, в тому, що хоча б одна з його функцій не має визначення. Але навіть якщо базовий клас є абстрактним, його все одно можна використовувати для оголошення вказівників і посилань, які необхідні для підтримки динамічного поліморфізму.

6.4 Класи-інтерфейси

Поняття класів-інтерфейсів

При проектуванні певних архітектурних рішень програмного забезпечення використовують так звані інтерфейсні класи. Цей підхід особливо ефективний при командній розробці складних проектів, що складаються з багатьох слабо пов'язаних класів і модулів. У C++, на відміну C#, немає спеціального типу інтерфейс. Інтерфейси C# будуть детально розглянуті в частині 2 «Об’єктно-орієнтоване програмування на мові C#».

У C++ клас-інтерфейс оголошується у вигляді базового абстрактного класу.

Існує традиція назви класів-інтерфейсів розпочинати з букви I. Рекомендується дотримуватися цієї традиції і в інших об'єктно-орієнтованих мовах , таких як Java і в C#.

Приклад оголошкння класу-інтерфейсу:

```
class IMovable
{
public:
     virtual void move() = 0;
};
```

Клас-інтерфейс записується тільки у вигляді оголошення. Не може бути створений об’єкт-екземпляр даного класу.

Реалізація (implemetation) функцій інтерфейсу покладається на похідні класи. Похідні від класу-інтерфейсу класи зобов’язані імплементувати всі його методи.

```
class Person : public IMovable
{
public:
     void Move() {
          cout << "Людина йде пішки" << endl;
     }
};

class Driver : public IMovable
{
public:
     void Move() {
          cout << "Водій веде машину" << endl;
     }
}
```

Згідно канонів ООП клас-інтерфейс може мати тільки public функції-члени (методи). Не може містити дані-члени (поля), конструктор та деструктор, але треба зазначити, що даний перелік основних властивостей класівінтерфейсів є досить умовним. Так для коректної реалізації поліморфізму необхідно визначити у базовому класі-інтерфейсі віртуальний деструктор, але ж клас-інтерфейс не повинен його мати? Тим не менше, якщо ввести у класс-інтерфейс віртуальний деструктор, програма буде успішно скомпільована.

Динамічна зміна поведінки

Існують системи, поведінка яких може визначатися відповідно до одного алгоритму з деякого сімейства. Всі алгоритми цього сімейства є спорідненими: призначені для вирішення спільних завдань, мають однаковий інтерфейс для використання і відрізняються тільки реалізацією (поведінкою).

Об'єктно-орієнтований дизайн такої програми може бути побудований на ідеї використання поліморфізму. В результаті отримуємо набір родинних класів із загальним інтерфейсом і різними реалізаціями алгоритмів. На рисунку 6.6. показано поліморфні класи, що визначають за допомогою единого інтерфейсу різні реалізації алгоритмів стисення даних.

Рис. 6.6. Загальний інтерфейс і різні реалізації алгоритмів

```
#include <iostream>
#include<conio.h>
using namespace std;


class Compressor
{
public:
     virtual void Compression() = 0;
};
```

```
class ZipCompression : public Compressor
{
public:
     void  Compression()
     {
     cout << "Цей метод реалізує алгоритм ZIP-компресії" << endl;
     }
};

class RarCompression : public Compressor
{
public:
     void  Compression()
     {
     cout << "Цей метод реалізує алгоритм RAR-компресії" << endl;
     }
};



class ArjCompression : public Compressor
{
public:
     void  Compression()
     {
     cout << "Цей метод реалізує алгоритм ARJ-компресії" << endl;
     }
};
int main()
{
     setlocale(0, "Ukr");
     Compressor * p;
     p = new ZipCompression();
     p->Compression();
     p = new RarCompression();
     p->Compression();
     p = new ArjCompression();
     p->Compression();
     _getch();
}
```

Представленому підходу властиві такі недоліки:

1. Реалізація алгоритму жорстко прив'язана до його підкласу, що ускладнює підтримку і розширення такої системи.
2. Система, що побудована на основі наслідування, є статичною. Замінити один алгоритм на інший в ході виконання програми вже неможливо.

І тут виникає ідея, яка полягає в тому, щоб перенести в окрему ієрархію класів всі деталі, пов'язані з реалізацією алгоритмів. Для програми стиснення

файлів абстрактний базовий клас ICompression цієї ієрархії оголошує інтерфейс, загальний для всіх алгоритмів і використовуваний класом Compressor.

```
// оголошуємо інтерфейс, загальний для всіх алгоритмів компресії
class ICompression
{
public:
     virtual void  Compress() = 0;
     virtual ~ICompression() {} // приклад віртуального деструктора
};
```

Похідні класи ZIP_Compression, ARJ_Compression і RAR_Compression його реалізують відповідно до тих чи інших алгоритмом.

Клас Compressor містить вказівник на об'єкт абстрактного типу ICompression і призначений для переадресації запитів конкретного алгоритму з боку користувача.

```
class Compressor
{
     ICompression* p; // вказівник на інтерфейс
public:
     Compressor(ICompression* p)
     {
          this->p = p;
     }
     ~Compressor()
     {
          delete p;
     }
     void ExecuteCompress()
     {
          p->Compress();
     }
     void ChangeAlgorythm(ICompression* p)
     {
          this->p = p;
     }
};

class ZipCompression : public ICompression
{
public:
     void  Compress()
     {
     cout << "Цей метод реалізує алгоритм ZIP-компресії" << endl;
     }
};
```

```
class RarCompression : public ICompression
{
public:
     void  Compress()
     {
     cout << "Цей метод реалізує алгоритм RAR-компресії" << endl;
     }
};

class ArjCompression : public ICompression
{
public:
     void  Compress()
     {
     cout << "Цей метод реалізує алгоритм ARJ-компресії" << endl;
     }
};


class MyCompression : public ICompression
{
public:
     void  Compress()
     {
     cout << "Цей метод реалізує власний алгоритм компресії" << endl;
     }
};
```

Підхід, застосований у цьому прикладі, є ні що інше, як використання одного з патернів проектування під назвою Strategy (стратегія) [5].

Патерн Strategy переносить в окрему ієрархію класів всі деталі, пов'язані з реалізацією алгоритмів.

Для заміни одного алгоритму іншим досить переналаштувати покажчик на об'єкт потрібного типу.

```
Compressor* comp = new Compressor(new ZipCompression());
comp->ExecuteCompress();

comp = new Compressor(new RarCompression());
comp->ExecuteCompress();
```

Можна гнучно змінювати алгоритм:

```
comp->ChangeAlgorythm(new MyCompression());
comp->ExecuteCompress();
```

На рисунку 6.7. показано діаграма класів для патерна Strategy.

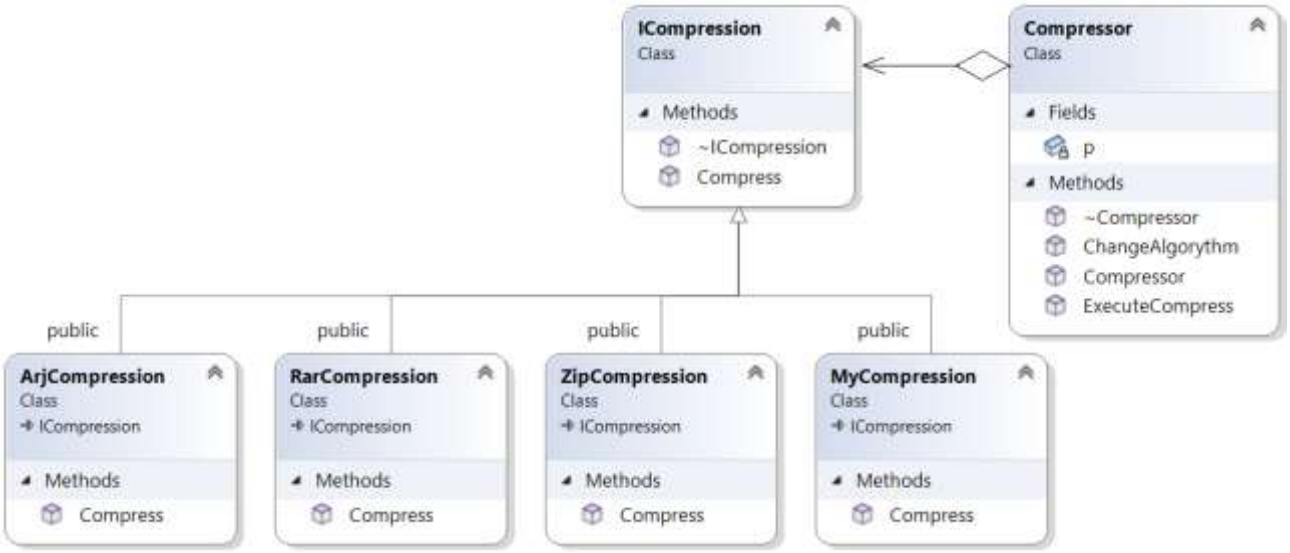


Рис. 6.7. Діаграма класів для патерна Strategy

Переваги патерна Strategy

1. Систему простіше підтримувати і модифікувати, через те, що сімейство алгоритмів перенесено в окрему ієрархію класів.
2. Патерн Strategy надає можливість заміни одного алгоритму іншим в процесі виконання програми.
3. Патерн Strategy дозволяє приховати деталі реалізації алгоритмів від клієнта.

Недоліки патерна Strategy

1. Для правильного налаштування системи користувач повинен знати про особливості всіх алгоритмів.
2. Кількість класів в системі, побудованої із застосуванням паттерна Strategy, зростає.

6.5 Динамічна ідентифікація типів і оператори приведення типу

С ++ реалізує поліморфізм за допомогою використання ієрархії класів, віртуальних функцій і вказівників на базові класи.

Вказівник на базовий клас можна використовувати для посилання на об’єкти як базового класу, так і на об’єкти будь-якого похідного класу.

Отже, не завжди заздалегідь відомо, на об'єкт якого типу буде посилатися вказівник на базовий клас в довільний момент часу.

Це з'ясується тільки при виконанні програми – при використанні одного із засобів динамічної ідентифікації типів (run-time type identification – RTTI).

Для отримання типу об'єкта під час виконання програми використовується оператор **typeid**. Для цього необхідно включити в програму заголовок

```
#include <typeinfo>
```

Найпоширеніший формат використання оператора typeid такий:
typeid(object)

Тут елемент object означає об'єкт, тип якого потрібно отримати. Можна виконувати не тільки вбудований тип, але і тип класу, створеного програмістом. Оператор typeid повертає посилання на об'єкт типу type_info, який описує тип об'єкта object.

У класі **type_info** визначені наступні public-члени:

bool operator == (const type_info &ob);

bool operator !=(const type_info &ob);

bool before(const type_info &ob);

const char *name();

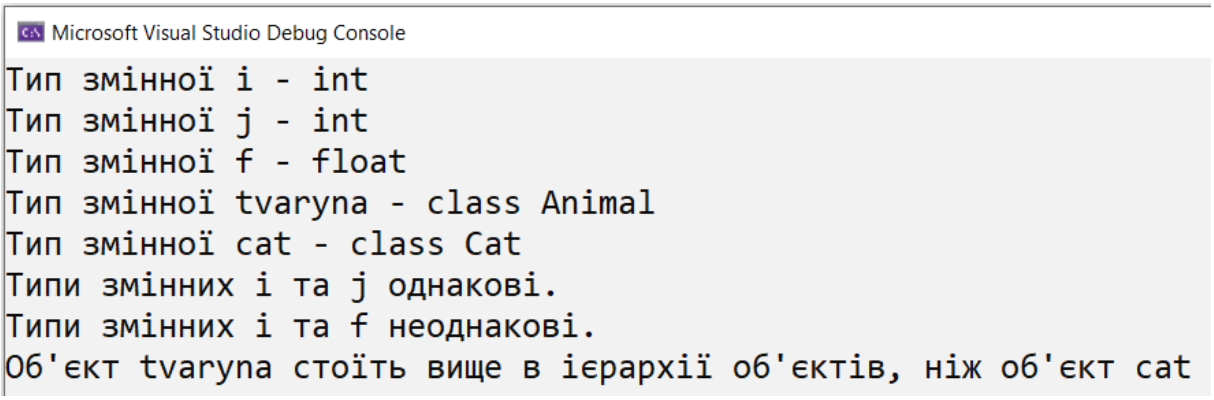
Перевантажені оператори «==» і «! =» служать для порівняння типів.

Функція before() повертає true, якщо викликаючий об'єкт стоїть вище в ієрархії об'єктів, ніж об'єкт, який використовується в якості параметра.

Розглянемо простий приклад використання оператора typeid:

```
class Animal {
 // . . .
};
class Cat : public Animal
{

};
int main()
{
 setlocale(0, "Ukr");
 int i, j;
 float f;
 Animal tvaryna;
 Cat cat;
 cout << "Тип змінної i - " << typeid (i).name() << endl;
 cout << "Тип змінної j - " << typeid (j).name() << endl;
 cout << "Тип змінної f - " << typeid (f).name() << endl;
 cout << "Тип змінної tvaryna - " << typeid (tvaryna).name() << endl;
 cout << "Тип змінної cat - " << typeid (cat).name() << endl;
 if (typeid (i) == typeid (j))
      cout << "Типи змінних i та j однакові." << endl;
 if (typeid (i) != typeid (f))
      cout << "Типи змінних i та f неоднакові." << endl;
 if (typeid (tvaryna).before(typeid (cat)))
 {
      cout << "Об'єкт tvaryna стоїть вище в";
      cout <<" ієрархії об'єктів, ніж об'єкт cat" << endl;
 }
}
```

При виконанні програма генерує ось такі результати:



Якщо оператор typeid застосовується до вказівника на поліморфний базовий клас, він автоматично повертає тип реального об'єкта, на який той вказує: будь то об'єкт базового класу або об'єкт класу, виведеного з базового. Отже, оператор typeid можна використовувати для динамічного визначення типу об'єкта, що адресується вказівником на базовий клас.

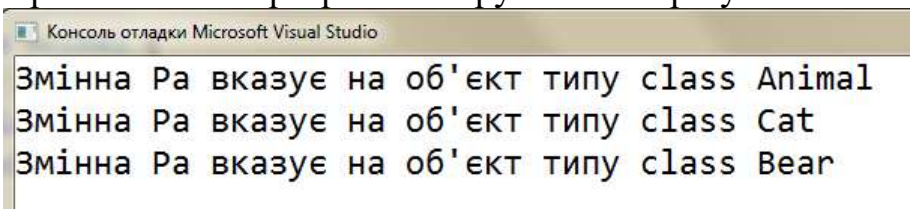
Слід пам'ятати, що вказівник обов'язково повинен бути розіменований. Коли цей розіменованний вказівник виявляється нульовим, з’являється спеціальний виняток bad_typeid.

Приклад застосування оператора typeid до ієрархії поліморфних класів:

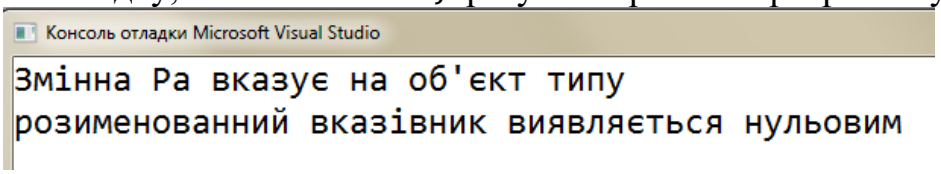
```
class Animal {
 // . . .
public:
 virtual void fun() {} // робимо клас Animal  поліморфним
};
class Cat : public Animal
{ };
class Bear : public Animal
{ };

int main() {
 setlocale(LC_ALL, "ukr");
 Animal *Pa, tvaryna;
 Cat cat;
 Bear bear;
 Pa = &tvaryna; // Pa = NULL;
 //ПАМ'ЯТКА: Вказівник обов'язково повинен бути розименован.
 //коли цей розименованний вказівник виявляється нульовим,
 //створено спеціальний виняток bad_typeid
 try {
       cout << "Змінна Pa вказує на об'єкт типу ";
       cout << typeid (*Pa).name() << endl;
       Pa = &cat;
       cout << "Змінна Pa  вказує на об'єкт типу ";
       cout << typeid (*Pa).name() << endl;
       Pa = &bear;
       cout << "Змінна Pa  вказує на об'єкт типу ";
       cout << typeid (*Pa).name() << endl;
 }
```

```
 catch (...)
 {
 cout << endl << "розименованний вказівник виявляється нульовим ";
 }
}
```

При виконанні програма генерує ось такі результати:



У випадку, коли `Pa = NULL;` результат роботи програми буде таким:



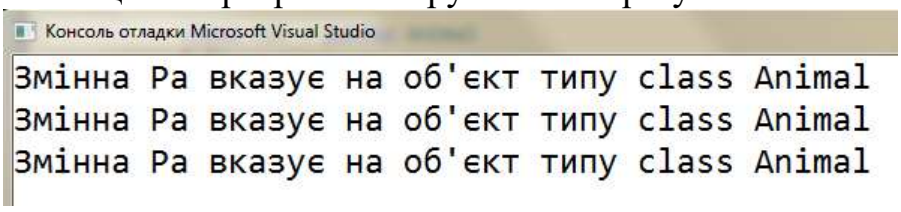
Якщо оператор typeid застосовується до вказівника на базовий клас поліморфного типу, тип реально адресуємого об'єкта, як підтверджують ці результати, буде визначено під час виконання програми.

У всіх випадках застосування оператора typeid до вказівника на не поліморфну ієрархію класів буде отримано вказівник на базовий тип, тобто те, на що цей вказівник реально вказує, визначити не можна.

У попередній програмі в базовому класі Animal приберемо ключове слово virtual:

```
class Animal {
    // . . .
public:
    virtual void fun() {} // робимо клас Animal не поліморфним
};
```

Після цього програма генерує ось такі результати:



Посилання на об'єкти ієрархії поліморфних класів працюють подібно вказівникам. Якщо оператор typeid застосовується до посилання на поліморфний клас, він повертає тип об'єкта, на який воно реально посилається, і це може бути об'єкт не базового, а похідного типу. Описаний засіб найчастіше використовується при передачі об'єктів функціям за посиланням.

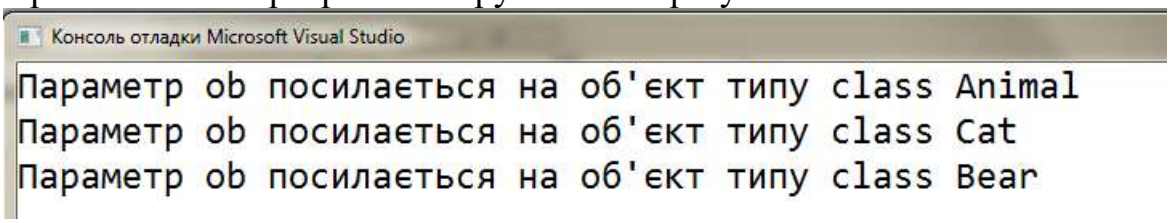
Наприклад, в наступній програмі функція WhatType () оголошує параметр-посилання на об'єкти базового класу Animal. Це означає, що функції WhatType () можна передавати посилання на об'єкти типу Cat або Bear або посилання на

об'єкти будь-яких класів, похідних від базового класу Animal. Оператор typeid, застосований до такого параметру, поверне реальний тип об'єкта, переданого функції.

Наступний приклад демонструє застосування оператора typeid до посилального параметру:

```
class Animal {
 // . . .
public:
 virtual void fun(){} // робимо клас Animal поліморфним
};
class Cat : public Animal
{ };
class Bear : public Animal
{ };

// застосування оператора typeid до посилального параметру:
void WhatType(Animal & ob)
{
 cout << "Параметр ob посилається на об'єкт типу ";
 cout << typeid (ob).name() << endl;
}

int main()
{
 setlocale(0, "Ukr");
 Animal tvaryna;
 Cat cat;
 Bear bear;
 WhatType(tvaryna);
 WhatType(cat);
 WhatType(bear);
}
```

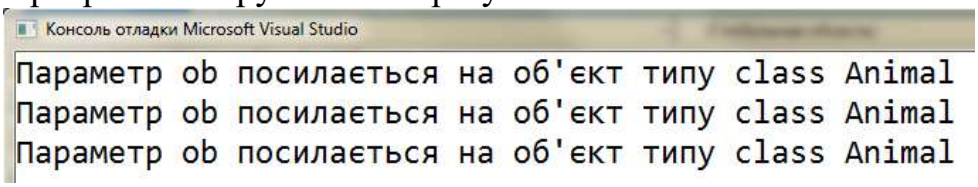
При виконанні програма генерує ось такі результати:



Після внесення в код зміни:

```
class Animal {
 // . . .
public:
 void fun() {} // робимо клас Animal не поліморфним (без virtual)
};
```

Програма генерує ось такі результати:



Оператори приведення типів. Оператор dynamic_cast

Оператор **dynamic_cast** виконує операцію приведення поліморфних типів під час виконання програми. Загальний формат застосування оператора dynamic_cast такий:

dynamic_cast <type> (expr)

Тут елемент type означає новий тип, який є метою виконання цієї операції, а елемент expr- вираз, що приводиться до цього нового типу. Тип type повинен бути представлений вказівником або посиланням, а вираз expr має приводитися до вказівника або посилання.

Таким чином, оператор dynamic_cast можна використовувати для перетворення вказівника одного типу у вказівник іншого або посилання одного типу на посилання іншого.

Цей оператор в основному використовується для динамічного виконання операцій приведення типу серед поліморфних типів.

Розглянемо простий приклад. Припустимо, що клас Animal - поліморфний, а клас Cat – похідний від класу Animal.

```
class Animal
{
 virtual void f() {}; // робимо клас поліморфним
};
class Cat : public Animal
{
public:
 void CatOnly() {
       cout << "Це об'єкт класу Cat."<<endl;
 }
};

int main()
{
 setlocale(0, "Ukr");
 Animal *Pa, animal;
 Cat *Pc, cat;
 // Використання оператора dynamic_cast
 Pa = &animal;
 Pc = dynamic_cast <Cat *> (Pa); // помилка
 if (Pc)
       Pc->CatOnly();
 else
   cout << "Приведення об'єкта типу Animal до типу Cat не виконано." << endl;
 Pa = &cat;
 Pc = dynamic_cast <Cat *> (Pa);
```

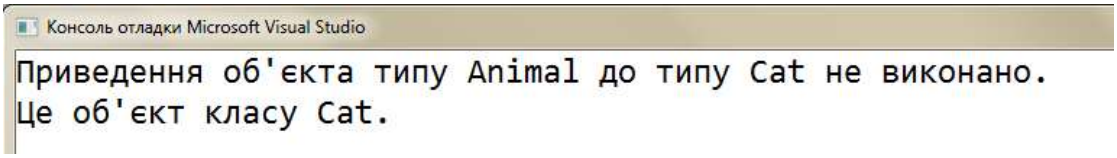
```
 if (Pc)
      Pc->CatOnly();
 else
   cout << endl << "Помилка, приведення типу має бути реалізовано!";
}
```

В наступному фрагменті коду

```
 Pa = &animal;
 Pc = dynamic_cast <Cat *> (Pa); // помилка
```

спроба зробити операцію приведення типу буде невдалою, оскільки Pa в дійсності вказує на об'єкт базового класу Animal, і неправомірно приводити вказівник на базовий клас до типу вказівника на похідний, якщо адресується їм об'єкт не є насправді об'єктом похідного класу.

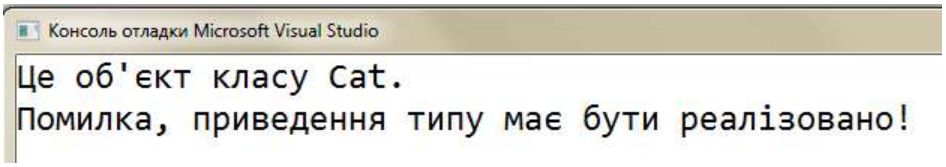
Маємо такий результат:



Якщо змінимо код:

```
Pa = &cat; // &animal; тепер вказує на об’єкт похідного класу
Pc = dynamic_cast <Cat *> (Pa);

if (Pc)
      Pc->CatOnly();
else
      cout << "Приведення до типу Cat не виконано." << endl;
 Pa = &animal; //а тут навпаки тепер вказує на об’єкт похідного класу
```

Результат буде таким:



Як бачите, використання оператора dynamic_cast спрощує логіку, необхідну для перетворення вказівника на базовий клас в вказівник на похідний клас. Ось як виглядають результати виконання цієї програми.

6.6 Обробка виняткових ситуації

Виняткова ситуація (або виключення) – це помилка, яка виникає під час виконання програми.

Використовуючи підсистему обробки виняткових ситуацій мови C++, з такими помилками успішно можна впоратися.

При виникненні виняткових ситуацій під час роботи програми автоматично викликається так званий обробник виключень.

Обробка винятків – це системні засоби, за допомогою яких програма може впоратися з помилками під час виконання.

Управління C++ – механізмом обробки винятків ґрунтується на трьох ключових словах: try, catch і throw. Вони утворюють взаємопов'язану підсистему, в якій використання одного з них передбачає застосування іншого.

Загальний формат try і catch блоків виглядає так:

```
try {
       // try-блок (блок коду, який підлягає перевірці на наявність помилок)
}
catch (type1 arg) {
    // catch-блок (обробник виключення типу type1)
}
catch {type2 arg) {
    // catch-блок (обробник виключення типу type2)
}
catch {type3 arg) {
    // catch-блок (обробник виключення типу type3)
}
// ...
catch (typeN arg) {
  // catch-блок (обробник виключення типу typeN)
}
```

З одним **try**-блоком може бути пов'язана не одна, а кілька catch-інструкцій. Яка саме з них буде виконана, визначається типом винятку.

Щоб виняток був перехоплений, необхідно забезпечити його "викидання" в try-блоці.

Загальний формат інструкції throw виглядає так:

```
throw exception;
```

Розглянемо простий приклад обробки виключень засобами мови C++.

```
#include <iostream>
using namespace std;
void main()
{
    setlocale(0, "Ukr");
    cout << "Початок" << endl;
    try {
        // початок try-блока
        cout << "У try - блоці" << endl;
        throw 99; // генерування помилки
        cout << "Ця інструкція не буде виконана" << endl;
    }
    catch (int i)
    {
        // перехоплення помилки
```

```
        cout << "Перехоплення виключення. Його значення дорівнює ";
        cout<< i << endl;
    }
    cout << "Кінець" << endl;
}
```

```
Выбрать C:\Users\Yura\Desktop\Лекції ООП 2025\OOP_CPP\x64\Debug\OOP_CPP.exe
Початок
У try - блоці
Перехоплення виключення. Його значення дорівнює 99
Кінець
```

Розглянемо уважно код цієї програми

Тут **try**-блок містить три інструкції, а інструкція **catch** (int i) призначена для обробки винятку целочисленного типу. У цьому try-блоці виконуються тільки дві з трьох інструкцій: cout і **throw**.

Після генерування винятку управління передається **catch**-виразу, при цьому виконання try-блоку припиняється. Необхідно розуміти, що **catch**-інструкція не викликається, а просто з неї триває виконання програми після "викиду" виключення.

Тип винятку повинен збігатися з типом, заданим в catch-інструкції. Наприклад, якщо в попередній програмі тип int, зазначений в catch-вираженні, замінити типом double, то виняток перехоплене не буде, і станеться аварійне завершення програми.

```
cout << "Початок" << endl;
try {
    // початок try-блока
    cout << "У try - блоці" << endl;
    throw 99; // генерування помилки
    cout << "
}
catch (double
{
    // перехо
```

Exception Unhandled

Unhandled exception at 0x00007FFF13BC5339 in OOP_CPP.exe: Microsoft C++ exception: int at memory location 0x00000038D736FD84.

Show Call Stack | Copy Details | Start Live Share session...

Exception Settings

Такі результати виконання цієї програми пояснюються тим, що виключення цілочисленого типу не перехоплюється інструкцією catch (double i)

Виняток, згенерований функцією, викликаної з try-блоку, може бути перехоплений цим же try-блоком. Розглянемо наступну програму:

```
#include <iostream>
using namespace std;
 void Xtest(int test)
     {
       cout << "В функції Xtest(), значення test дорівнює: "<< test <<
endl;
          if(test==2)
           throw test;
```

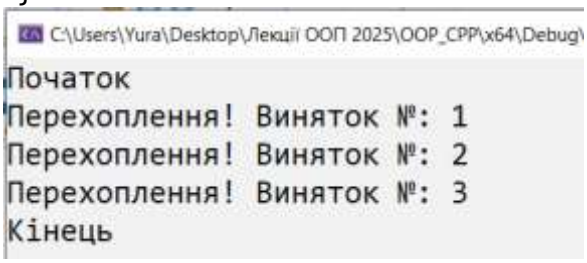
```
}
void main()
{
      setlocale(0, "Ukrainian");
  // Генерування виключення з функції, що викликається з try-блоку.
  cout << "Початок"<<endl;
  try {
       cout << "В try-блоке "<<endl;
      Xtest (0);
       Xtest (1);
       Xtest (2);
      }
   catch (int i) {
    // перехоплення помилки
   cout << "Перехоплення виключення. Його значення дорівнює: ";
   cout << i << endl;
   }
 cout << "Кінець";
}
```

```
C:\Users\Yura\Desktop\Лекції ООП 2025\OOP_CPP\x64\Debug\OOP_CPP.exe
Початок
В try-блоке
В функции Xtest(), значение test равно: 0
В функции Xtest(), значение test равно: 1
В функции Xtest(), значение test равно: 2
Перехоплення виключення. Його значення дорівнює: 2
Кінець
```

Блок try може бути локалізований в рамках функції. В цьому випадку при кожному її виконанні запускається і обробка виключень, пов'язана з цією функцією. Розглянемо наступну просту програму.

```
//Функціонування блоків try / catch поновлюється при кожному вході в
функцію
void Xhandler(int test)
     {
      try {

        if(test) throw test;
      }
       catch(int i) {
       cout << "Перехоплення! Виняток №: " << i << endl;
      }
     }

     void main()
     {
       cout << "Початок"<<endl;
       Xhandler (1);
       Xhandler (2);
       Xhandler (0);
```

```
    Xhandler (3);

   cout << "Кінець";
  }
```

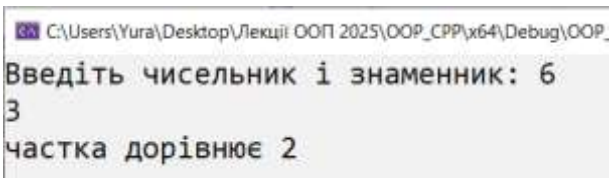


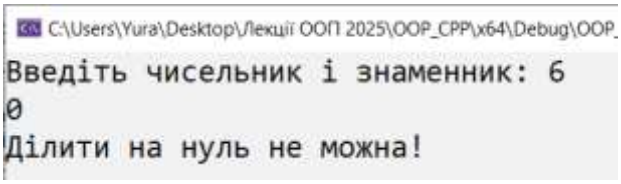
Перехоплення винятків класового типу

Виняток може мати будь-який тип, в тому числі і тип класу, створеного програмістом. У реальних програмах більшість винятків мають саме тип класу, а не вбудований тип.

В наступному прикладі інформація, що міститься в об'єкті класу винятків, дозволяє спростити обробку винятків.

```
// Використання класу винятків.
class MyException {
      public:
        string Message;
        MyException()
          { Message =""; }
        MyException(string s)
          { Message =s;}
     };

     void main()
     {
         int a, b;
         try {
         cout << "Введіть чисельник і знаменник: ";
         cin >> a >> b;
           if( !b)
                  throw MyException("Ділити на нуль не можна!");
           else
               cout << "частка дорівнює " << a/b << endl;
            }
         catch (MyException e) {
         // перехоплення помилки
         cout << e.Message << "";
           }
   }
```

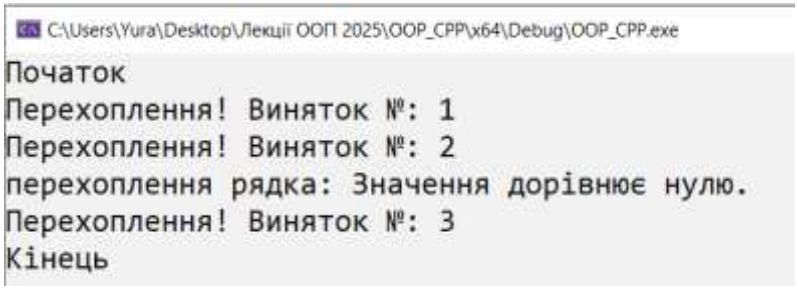




Використання декількох catch-інструкцій

Як згадувалося вище, з try-блоком можна пов'язувати не одну, а кілька catch-інструкцій. Насправді саме така практика і є звичайною. Але при цьому всі catch-інструкції повинні перехоплювати виключення різних типів.

```
// Тут може бути перехоплення виключень різних типів
   void Xhandler(int test)
   {
        try
        {
        if(test)
             throw test;
        else
             {
                  string t="Значення дорівнює нулю.";
                  throw t;
             }
        }
        catch (int i)
        {
        cout << "Перехоплення! Виняток №: " << i << endl;
        }
         catch(string str)
           {
         cout << "перехоплення рядка: ";
         cout << str << endl;
         }
    }

    void main()
    {
      cout << "Початок"<<endl;
      Xhandler (1);
      Xhandler (2);
      Xhandler (0);
      Xhandler (3);
     cout << "Кінець";
    }
```

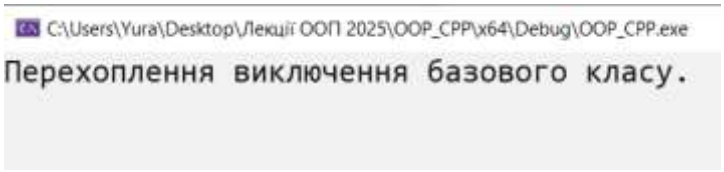


Перехоплення винятків базового класу

Як виконуються catch-інструкції, пов'язані з похідними класами? catch-вираз для базового класу "відреагує збігом" на виключення будь-якого похідного типу (тобто типу, виведеного з цього базового класу).

Отже, якщо потрібно перехоплювати виключення як базового, так і похідного типів, в catch-послідовності catch-інструкцію для похідного типу необхідно помістити перед catch-інструкцією для базового типу. В іншому випадку catch-вираз для базового класу буде перехоплювати (крім "своїх") і виключення всіх похідних класів.

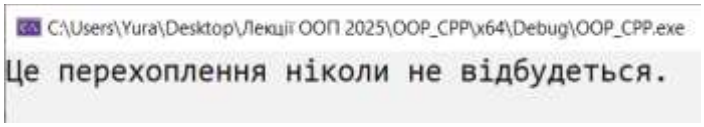
```
// Перехоплення винятків базових і похідних типів
class B {
};

class D : public B {
};

void main()
{
	setlocale(0, "");
	D derived;
	try {
		throw derived;
	}
	catch (B b) {
		cout << "Перехоплення виключення базового класу." << endl;
	}
	catch (D d) {
		cout << "Це перехоплення ніколи не відбудеться." << endl;
	}

}
```



Оскільки тут об'єкт derived - це об'єкт класу D, який виведений з базового класу B, то виключення типу derived буде завжди перехоплюватися першим catch-виразом; друга ж catch-інструкція при цьому ніколи не виконається.

Щоб виправити ситуацію, достатньо поміняти порядок проходження цих catch-інструкцій на протилежний.

```
catch (D d) {
     cout << "Це перехоплення ніколи не відбудеться." << endl;
}
catch (B b) {
     cout << "Перехоплення виключення базового класу." << endl;
}
```



Крім розглянутих, існують і інші C ++ - засоби обробки винятків, які створюють певні зручності для програмістів.

Перехоплення всіх винятків

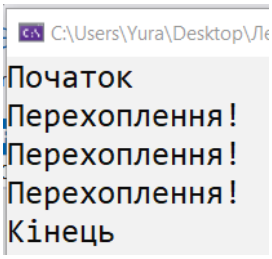
Іноді має сенс створити обробник для перехоплення всіх винятків, а не винятків тільки певного типу. Для цього достатньо використовувати такий формат catch-блоку:

```
catch (...) {
     // обробка всіх винятків
}
```

Крапки в круглих дужках забезпечують збіг з будь-яким типом даних. Використання формату catch (...) ілюструється в наступній програмі.

```
// Тут може бути перехоплення винятків різних типів
void Xhandler(int test)
{
     try
     {
          if (test==0) throw test; // генерує int- винятки
          if (test == 1) throw 'a'; // генерує char-винятки
          if (test == 2) throw 1.56; // генерує double-винятки
     }
     catch (...) // перехоплення всіх винятків
     {
          cout << "Перехоплення!"<< endl;
     }

}
```

```
void main()
{
	setlocale(0, "");
	cout << "Початок" << endl;
	Xhandler(0);
	Xhandler(1);
	Xhandler(2);
	cout << "Кінець";

	_getch();
}
```



Резюме

1. У C++ вказівник на базовий клас також можна використовувати для посилання на об'єкт будь-якого класу, виведеного з базового.
2. Подібно вказівниками, посилання на базовий клас також можна використовувати для доступу до об'єкта похідного типу.
3. Використовуючи вказівник на похідний клас, не можна отримати доступ до об'єкта базового типу.
4. Віртуальна функція – це функція, яка оголошується в базовому класі з використанням ключового слова virtual і перевизначається в одному або декількох похідних класах. Прототипи віртуальної функції і її перевизначень повинні бути абсолютно однаковими.
5. Клас, який включає віртуальну функцію, називається поліморфним класом.
6. Якщо похідний клас не перевизначає віртуальну функцію, то використовується функція, певна в базовому класі.
7. Суто віртуальна функція – це віртуальна функція, яка не має визначення в базовому класі.
8. Клас, який містить хоча б одну чисто віртуальну функцію, називається абстрактним, у такого класу не може бути об'єктів.
9. Для отримання типу об'єкта під час виконання програми використовується оператор typeid. Якщо оператор typeid застосовується до вказівника на базовий клас поліморфного типу, тип реально адресуємого об'єкта буде визначено під час виконання програми.

10. Оператор dynamic_cast виконує операцію приведення поліморфних типів під час виконання програми. оператор dynamic_cast можна використовувати для перетворення вказівника одного типу у вказівник іншого або посилання одного типу на посилання іншого.

11. Виняткова ситуація (або виключення) – це помилка, яка виникає під час виконання програми.

12. Управління C++ – механізмом обробки винятків ґрунтується на трьох ключових словах: try, catch і throw. Вони утворюють взаємопов'язану підсистему, в якій використання одного з них передбачає застосування іншого.

Завдання до лабораторної роботи

1. Визначити ієрархію класів (відповідно до варіанта). З переліку класів вибрати один, який буде стояти на чолі ієрархії. Переважно це має бути абстрактний клас.
2. Реалізувати класи. Створити поліморфну функцію (віртуальну або суто віртуальну) відповідно варіанту.
3. Написати демонстраційну програму, в якій створюються об'єкти різних класів і поміщаються в масив або в список, після чого масив або список проглядається через виклик поліморфної функції.
4. Якщо базовий клас не абстрактний, зробити відповідну поліморфну функцію не віртуальною і подивитися, що буде. Пояснити отриманий результат.

Варіанти завдань:

1. Трикутник, геометрична фігура, трапеція, рівносторонній трикутник. Поліморфна функція повертає значення площі відповідної фігури.
2. Куля, усічений прямий конус, геометричне тіло, конус. Поліморфна функція повертає значення об’єму відповідного геометричного тіла. (Формули для обчислення знайти самостійно)
3. Куля, геометричне тіло, паралелепіпед, тетраедр. Поліморфна функція повертає значення площі поверхні тіла.
4. Кінь, слон, тура, ферзь, шахова фігура. Поліморфна функція приймає два параметра: перший-позиція фігури на шахівниці (наприклад, g2), а другий параметр – поле, куди треба піти (наприклад, h3), і виводить повідомлення – чи можливий такий хід, чи ні;
5. Кінь, слон, тура, ферзь, шахова фігура. Поліморфна функція приймає два параметра: перший – поле, де стоїть король противника (наприклад, g2), а другий параметр – позиція фігури на шахівниці (наприклад, h3), і виводить повідомлення – чи оголошений королю шах, чи ні;
6. Організація, страхова компанія, будівна компанія, метробуд. Поліморфна функція для страхової компанії виводить прізвище страхового агента, кількість полісів і загальну суму, для будівної компанії - прізвище виконроба, загальний кошторис (суму в грн) і метраж (загальну площу кв.м), для метробуд – назва маршруту, кількість пасажирів, відстань в км.;

7. Ціле беззнакове число, двійкове число, десяткове число, шістнадцяткове число. Поліморфна функція виводить на екран число відповідно як двійкове, десяткове або шістнадцяткове.

8. Базовий клас – трикутник (поля – довжини двох сторін і кут між ними), похідні – рівносторонній, рівнобедрений, прямокутний. Поліморфна функція виводить на екран значення площі та периметра.

9. Двигун, вічний двигун (perpetuum mobile), двигун внутрішнього згоряння, електродвигун. Поліморфна функція для двигуна внутрішнього згоряння виводить робочий об'єм двигуна (л), потужність (л.с) і витрата бензину на 100 км., для електродвигуна - потужність (квт), напруга (в) і кількість фаз, ну а для perpetuum mobile – придумайте самі, щоб показати, що такий не існує.

10. Опір, активний опір, ємнісний опір, індуктивний опір. Поліморфна функція повертає значення опіру для відповідного типу і виводить повідомлення про тип опіру. Ланцюг змінного струму включає в себе певну кількість активних, ємнісних і індуктивних елементів.

Довідка:

активний опір $R = \frac{\rho l}{S}$; де р - питомий опір провідника, l - довжина провідника, S - площа перерізу провідника;

ємнісний опір $X_c = \frac{1}{\omega C}$; де ω -циклічна частота змінного струму (рад/с), C – ємність;

індуктивний опір $X_L = \omega L$, де L - індуктивність.

Тести для закріплення матеріалу

1. Що з перерахованого є прикладом поліморфізму
   а) наявність в класі декількох деструкторів;
   б) використання віртуальних функцій;
   в) створення декількох об'єктів класу;
   г) всі відповіді невірні

2. Віртуальна функція (virtual) - це
   а) метод, який не має коду в реалізації;
   б) метод, оголошений в базовому класі і який може бути перевизначений в похідному;
   в) статичний метод абстрактного класу;
   г) метод, оголошений в базовому класі і який має бути перевизначений в похідному;.

3. Вкажіть правильне оголошення чисто віртуального методу (C++):
   а) virtual void f (int) = 0;

б) virtual void f (int) = '\ 0';
в) virtual void f (int);
г) virtual clear void f (int);

4. Якщо віртуальний метод перевизначений в похідному класі, то об'єкти похідного класу
а) не можуть отримати доступ до методу базового класу;
б) можуть отримати доступ до методу базового класу за допомогою операції ->
в) можуть отримати доступ до методу базового класу за допомогою операції ::
г) можуть отримати доступ до методу базового класу за допомогою операції .

5. Якщо в базовому класі метод оголошений як віртуальний, то в похідному класі метод з таким же ім'ям і набором параметрів буде
а) звичайним методом;
б) константним методом;
в) статичним методом;
г) віртуальним методом

6. Якщо є абстрактний клас A і похідний від цього класу клас A1 то яка з записів свідомо невірна?
(Відповідь вважається вірної, якщо відзначені всі правильні варіанти відповідей.)
а) A * a = new A;
б) A1 a1;
в) A a;
г) A *a=new A1

7. Якщо записано
```
class A {
public: virtual void f () {cout << 1; }
};
 class B: public A
  {
    public:
         void f () {cout << 2; }
 };
```
то що буде надруковано в результаті виконання коду?
```
B b; A & a = b; a.f ();
```
а) 1
б) 2 1
в) 2
г) 1 2

8. Абстрактний клас (C++):
   а) містить хоча б один суто віртуальний метод;
   б) є класом з константними даними;
   в) містить віртуальні члени-дані;
   г) містить тільки суто віртуальні методи

9. Якою командою ініціюється виключення?
   а) try;
   б) catch;
   в) throw;
   г) exept.

10. Є наступний код:

```
try {
    throw 99; // генерування помилки
    cout << "Ця інструкція не буде виконана" << endl;
}
catch (double i)
{
    // перехоплення помилки
    cout << "Перехоплення виключення. Його значення дорівнює ";
    cout<< i << endl;
}
```

В результаті виконання цього коду буде виведено:
   а) Перехоплення виключення. Його значення дорівнює 99;
   б) Ця інструкція не буде виконана
   в) Unhandled exception
   г) Помилка компіляції.

РОЗДІЛ 7. ШАБЛОНИ

Шаблон – це одне з найскладніших і потужних засобів в C++. Шаблони дозволяють досягти одну з найважчих цілей в програмуванні – створити багаторазово використовуваний код.

Використовуючи шаблони, можна створювати узагальнені функції і класи. В узагальненій функції (або класі) тип даних, що обробляється нею (ним), задається як параметр.

7.1 Узагальнені функції

Узагальнена функція – це функція, що перевантажує сама себе.

Узагальнена функція створюється за допомогою ключового слова *template*. Звичайне значення слова template (шаблон) точно відображає мету його застосування в C++. Це ключове слово використовується для створення шаблону (або оболонки), який описує дії, що виконуються функцією. Компілятору ж залишається «доповнити відсутні деталі» відповідно до заданого значенням параметра.

Загальний формат визначення шаблонної функції має такий вигляд:

template <class T Type> тип ім'я функції (список параметрів)

```
 {
   // тіло функції
 }
```

Визначення узагальненої функції починається з ключового слова template.

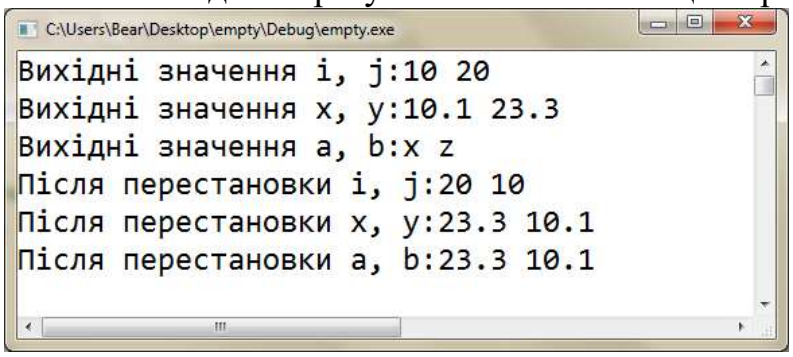
Тут елемент Ttype є «заповнювач» для типу даних, які обробляються функцією. Це ім'я може бути використано в тілі функції. Але воно означає всього лише заповнювач, замість якого компілятор автоматично підставить реальний тип даних при створенні конкретної версії функції. І хоча для завдання узагальненого типу в template-оголошенні за традицією застосовується ключове слово class, можна також використовувати ключове слово typename.

У наступному прикладі створюється узагальнена функція, яка міняє місцями значення двох змінних, що використовуються при її виклику.

```
// Визначення шаблонної функції.
template <class X> void swapargs (X & a, X & b)
{
X temp;
temp = a;
a = b;
b = temp;
}

int main ()
{
int i = 10, j = 20;
double x = 10.1, y = 23.3;
char a = 'x', b = 'z';
cout << "Вихідні значення i, j:" << i << " " << j <<endl;
```

```
cout << "Вихідні значення x, y:" << x << " "<< y << endl;
cout << "Вихідні значення a, b:" << a << " " << b << endl;
swapargs(i, j); // перестановка цілих чисел
swapargs(x, y); // перестановка значень з плаваючою точкою
swapargs(a, b); // перестановка символів
cout << "Після перестановки i, j:" << i << " " << j <<endl;
cout << "Після перестановки x, y:" << x << " "<< y << endl;
cout << "Після перестановки a, b:" << x << " "<< y << endl;
}
```

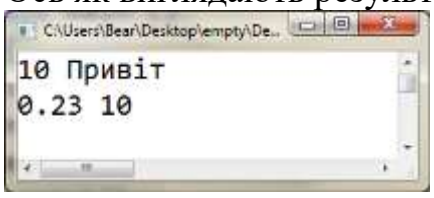
Ось як виглядають результати виконання цієї програми:



У template-інструкції можна визначити кілька узагальнених типів даних, використовуючи список елементів, розділених комою. Наприклад, в наступній програмі створюється шаблонна функція з двома узагальненими типами.

```
// Визначення шаблонної функції з двома узагальненими типами.
template <class type1, class type2>
void myfunc (type1 x, type2 y)
{
cout << x << " " << y <<endl;
}
int main ()
{
setlocale(0,"Ukr");
myfunc (10, "Привіт");
myfunc (0.23, 10L);
}
```

Ось як виглядають результати виконання цієї програми:



У цьому прикладі при виконанні функції main (), коли компілятор генерує конкретні екземпляри функції myfunc(), наповнювачі типів type1 і type2 замінюються спочатку парою типів даних int і char *, а потім парою double і long відповідно.

Узагальнені функції подібні перевантаженим функцій, але мають більше обмежень щодо застосування. При перевантаженні функцій в тілі кожної з них

зазвичай задаються різні дії. Але узагальнена функція повинна виконувати одну і ту ж дію для всіх версій - відмінність між версіями полягає тільки в типі даних.

Основні властивості параметрів шаблону функції

1. Імена параметрів шаблону повинні бути унікальними у всьому визначенні шаблону.
2. Список параметрів шаблону не може бути порожнім.
3. У списку параметрів шаблону може бути декілька параметрів, і кожному з них має передувати ключове слово `class`.
4. Ім'я параметра шаблона має всі права імені типу у визначеної шаблоном функції.
5. Визначена за допомогою шаблону функція може мати будь-яку кількість непараметрізованих формальних параметрів. Може бути параметризованим і значення, що повертається функцією.
6. У списку параметрів прототипу шаблону імена параметрів не зобов'язані збігатися з іменами тих же параметрів у визначенні шаблону.
7. При конкретизації параметризованої функції необхідно, щоб при виклику функції типи фактичних параметрів, відповідні однаково параметризованим формальним параметрам, були однакові.

7.2 Узагальнені класи

Крім узагальнених функцій, можна також визначити узагальнений клас. Для цього створюється клас, в якому визначаються всі використовувані їм алгоритми; при цьому реальний тип оброблюваних в ньому даних буде поставлено як параметр при створенні об'єктів цього класу.

Узагальнені класи особливо корисні в разі, коли в них використовується логіка, яку можна узагальнити.

Загальний формат оголошення узагальненого класу має такий вигляд:

template <class Ttype> class ім'я_класу {

...

}

Тут елемент Ttype є «заповнювач» для імені типу, який буде заданий при реалізації класу. При необхідності можна визначити кілька узагальнених типів даних, використовуючи список елементів, розділених комами.

Створивши узагальнений клас, можна створити його конкретний екземпляр, використовуючи наступний загальний формат:

ім'я_класу <тип> ім’я_об'єкту;

Тут елемент тип означає ім'я типу даних, які будуть оброблятися екземпляром узагальненого класу. Функції-члени узагальненого класу автоматично є узагальненими. Тому вам не потрібно використовувати ключове слово template для явного визначення їх такими.

Як і в разі узагальнених функцій, визначення узагальненого класу дає не сам клас, а лише схему (шаблон) для його побудови. Створення конкретного

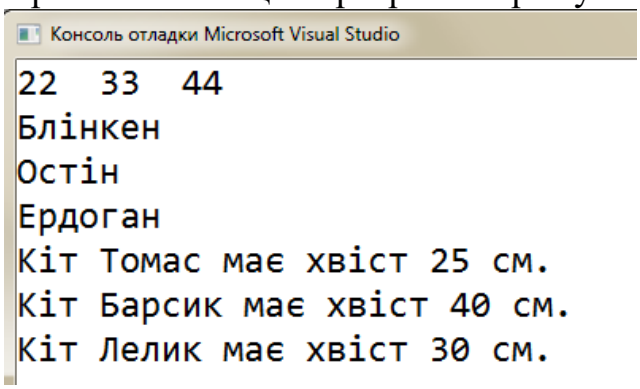
класу за узагальненим визначенням параметризованого класу називають конкретизацією шаблону (template specialization). Один і той самий узагальнений клас можна конкретизувати за допомогою різних типів даних, стандартних і програмованих класів, визначених у програмі чи в бібліотеках (користувацьких або стандартних). Кожен зі згенерованих таким способом класів має власне осібне ім’я, що складається з двох частин: імені узагальненого класу й імені фактичного типу.

У наступній програмі клас SmartArray (вперше представлений в розділі 4.4) перероблений в узагальнений. Це означає, що його можна використовувати для об'єктів будь-якого типу. Можна також створювати масив для зберігання даних, тип яких створено програмістом. В цьому прикладі узагальнений клас створюється створює масив цілих чисел, рядків и об’єктів користувацького класу Cat.

```
//Приклад шаблона класу
const int SIZE = 3;
class Cat
{
 string name; // ім'я кота
 int tail; // довжина хвоста кота
public:
 Cat()
 {
      this->name = "Барсик";
      this->tail = 40;
 }
 Cat(string name, int tail)
 {
      this->name = name;
      this->tail = tail;
 }
 void Print()
 {
 cout << "Кіт " << name << " має хвіст " << tail << " см." << endl;
 }
};
template <class T> class SmartArray
{
 T A[SIZE];
public:
 // Контроль попадання в допустимий інтервал для класу SmartArray.
 T & operator [] (int i)
 {
 if (i <0 || i> SIZE - 1) {
      cout << "Значення індексу виходить за межі масиву.";
           exit(1);
      }

 return A[i];
```

```
 }
};

int main()
{
 setlocale(0, "Ukr");
 SmartArray<int> ob_int;
 ob_int[0] = 22;
 ob_int[1] = 33;
 ob_int[2] = 44;
 for (int i = 0; i < 3; i++)
      cout << ob_int[i] << "  ";
 cout << endl;
 SmartArray<string> ob_str;
 ob_str[0] = "Блінкен";
 ob_str[1] = "Остін";
 ob_str[2] = "Ердоган";
 for (int i = 0; i < 3; i++)
      cout << ob_str[i] << endl;
 SmartArray<Cat> ob_cat;
 ob_cat[0] = *(new Cat("Томас", 25));
 ob_cat[1] = *(new Cat(); // конструктор без параметрів
 ob_cat[2] = *(new Cat("Лелик", 25));
 for (int i = 0; i < 3; i++)
      ob_cat[i].Print();
}
```

При виконанні цієї програми отримуємо наступні результати:



У наведеній програмі використано такі конкретизації узагальненого класу SmartArray: масиви цілих чисел `SmartArray<int>`, рядків зі стандартної бібліотеки `SmartArray<string>` і масив котів, визначених в класі `Cat`. Усі три класи різні, ніяк не пов'язані один з одним. Спільне для них лише те, що їхні визначення та реалізації автоматично згенеровано компілятором з одного й того самого узагальненого класу. При модифікації програм обробки масивів достатньо лише змінювати один файл, що містить шаблон параметризованого класу. Модифікація коду шаблону автоматично призводитиме до модифікації кожної з конкретизацій під час її повторної компіляції.

На прикладі класу SmatrArray неважко переконатися, що узагальнені функції і класи являють собою потужні засоби, які допоможуть збільшити

ефективність роботи програміста, оскільки вони дозволяють визначити загальний формат об'єкта, який можна потім використовувати з будь-яким типом даних. Узагальнені функції і класи позбавляють вас від стомлюючої праці по створенню окремих реалізацій для кожного типу даних, що підлягають обробці єдиним алгоритмом. Цю роботу зробить за вас компілятор: він автоматично створить конкретні версії визначеного вами класу.

7.3 Стандартна бібліотека шаблонів

Бібліотека STL (Standard Template Library) – це набір шаблонних класів і функцій загального призначення. Ядро стандартної бібліотеки шаблонів включає три основних елемента: **контейнери**, **алгоритми** і **ітератори**. Вони працюють спільно один з іншим, надаючи тим самим готові рішення різних завдань програмування.

STL спирається не тільки на контейнери, алгоритми та ітератори, а й інші стандартні компоненти. Основними з них є розподільники пам'яті, предикати та функції порівняння.

Розподільник пам'яті керує виділенням пам'яті контейнера.

Деякі алгоритми та контейнери використовують спеціальний тип функції, що називається предикатом (predicate). Існує два варіанти предикатів: унарний та бінарний. Унарний предикат приймає один аргумент, а бінарний – два. Ці функції повертають значення true/false, але точні умови, які змусять їх повернути справжнє чи хибне значення, визначаються програмістом.

Контейнери – це об'єкти, що містять інші об'єкти.

Існує кілька різних типів контейнерів. Наприклад, клас **vector** визначає динамічний масив, клас **queue** створює двосторонню чергу, клас **list** забезпечує роботу з лінійним списком.

Ці контейнери називаються послідовними контейнерами і є базовими в STL.

Крім базових, бібліотека STL визначає асоціативні контейнери, які дозволяють ефективно знаходити потрібні значення на основі заданих ключових значень (ключів).

Наприклад, клас **map** забезпечує зберігання пар «ключ-значення» і надає можливість знаходити значення за заданим унікальним ключем.

Кожен контейнерний клас визначає набір функцій, які можна застосовувати до даного контейнера. Наприклад, контейнер списку включає функції, призначені для виконання вставки, видалення та об'єднання елементів. А стек включає функції, які дозволяють поміщати значення в стек і витягувати їх з стека.

Ітератори

У бібліотеці STL для доступу до елементів в якості посередника використовується узагальнена абстракція, іменована ітератором. Ітератори подібні до вказівників.

Ітератори – це об'єкти, які тією чи іншою мірою діють подібно до вказівників. Вони дозволяють циклічно опитувати вміст контейнера практично

так, як це робиться за допомогою вказівника при циклічному опитуванні елементів масиву. Існує п'ять типів ітераторів:

1. довільного доступу (random access) – зберігають та зчитують значення, дозволяють організувати довільний доступ до елементів контейнера;
2. двонаправлені (bidirectional) – зберігають та зчитують значення, забезпечують інкрементно-декрементне переміщення;
3. однонаправлені (forward) – зберігають та зчитують значення, забезпечують лише інкрементне переміщення;
4. вхідні (input) – зчитують, але не записують значення, забезпечують лише інкрементне переміщення;
5. вихідні (output) – записують, але не зчитують значення, забезпечують лише інкрементне переміщення.

Кожен контейнер підтримує «свій» вид ітератора, який являє собою «модернізований» інтелектуальний вказівник, «знає» як отримати доступ до елементів конкретного контейнера.

Алгоритми

Алгоритми обробляють дані, що містяться в контейнерах. Їх можливості включають засоби ініціалізації, сортування, пошуку і перетворення вмісту контейнерів. Багато алгоритмів працюють з заданим діапазоном елементів контейнера.

Незважаючи на те що кожен контейнер забезпечує підтримку власних базових операцій, стандартні алгоритми дозволяють виконувати більш розширені або більш складні дії. Вони також дозволяють працювати з двома різними типами контейнерів одночасно.

Для отримання доступу до алгоритмів бібліотеки STL необхідно включити в програму заголовок **<algorithm>.**

Існують кілька класів алгоритмів:

1. алгоритми, що не модифікують;
2. алгоритми, що модифікують;
3. алгоритми видалення;
4. алгоритми сортування;
5. алгоритми для впорядкованих діапазонів;
6. чисельні алгоритми.

У бібліотеці STL визначено безліч алгоритмів. Всі ці алгоритми являють собою шаблонні функції. Це означає, що їх можна застосовувати до контейнера будь-якого типу.

Одна з найпопулярніших операцій, яку можна виконати для будь-якої послідовності елементів, – підрахувати їхню кількість. Для цього можна використовувати один із алгоритмів: count() або count_if().

Розглянемо приклад використання алгоритму count_if(). При виклику цього алгоритму окремим параметром треба вказати логічну функцію, що визначає критерій відбору елементів – предикат (в прикладі це функція pred).

```
#include <iostream>
#include <conio.h>
```

```
#include <algorithm>
#include <vector>
using namespace std;

class Student
{
public:
	Student(string name, double score)
	{
		Name = name;
		Score = score;
	}
	string Name;
	double Score;
};

bool pred(const Student& p) // предикат
{
	return p.Score < 60;
}



int main() {
	setlocale(0, "");
	vector<Student> students = { Student("Білл Гейтс", 75),
	Student("Дональд Туск", 75), Student("Ілон Маск", 45),
	Student("Дональд Трамп", 35), Student("Емануель Макрон", 65),
	Student("Фрідріх Мерц", 79), Student("Кір Стармер", 95) };
	int bad = count_if(students.begin(), students.end(), pred);
	for (auto s : students)
	{
		cout << s.Name << " " << s.Score << endl;
	}
	cout << "Кількість неуспішних студентів " << bad;
}
```

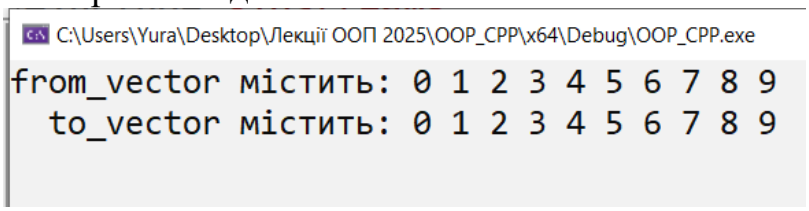
На екран виводиться такий результат:

```
Microsoft Visual Studio Debug Console
Білл Гейтс 75
Дональд Туск 75
Ілон Маск 45
Дональд Трамп 35
Емануель Макрон 65
Фрідріх Мерц 79
Кір Стармер 95
Кількість неуспішних студентів 2
```

Наступна програма використовує функцію std :: copy як для копіювання вмісту одного вектора в інший, так і для виведення на екран:

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <iterator>
using namespace std;
void main()
{
    setlocale(0, "Ukr");
    vector<int> from_vector;
    for (int i = 0; i < 10; i++) {
        from_vector.push_back(i);
    }
    cout << "from_vector містить: ";
    copy(from_vector.begin(), from_vector.end(),
ostream_iterator<int>(cout, " "));
    cout << endl;

    vector<int> to_vector(10);
    copy(from_vector.begin(), from_vector.end(), to_vector.begin());

    cout << "  to_vector містить: ";
    copy(to_vector.begin(), to_vector.end(),
ostream_iterator<int>(cout, " "));
    cout << endl;
}
```

На екран виводиться:



Демонстрація асоціативного контейнера

Асоціативний контейнер (асоціативний масив) – це масив, у якого індекси його елементів, на відміну від звичайних масивів, можуть бути будь-якого типу, а не тільки цілими типу int.

Так, асоціативний масив дуже зручно використовувати як словник. До речі, в мові C# подібний контейнер так і називається Dictionary (словник)

Програма, що демонструє використання словника:

```cpp
#include <iostream>
#include <map>
using namespace std;

int main()
{
	setlocale(0, "");
	map<string, string> dic; // словник
	map<char, int>::iterator ii; // це вказівник!!
	dic["cat"] = "кіт";
	dic["bear"] = "ведмідь";
	dic["wolf"] = "вовк";
	dic["cow"] = "корова";
	dic["elephant"] = "слон";
	cout << "Enter an English word: ";
	string word;
	cin >> word;
	cout << "Переклад: " << dic[word];
}
```

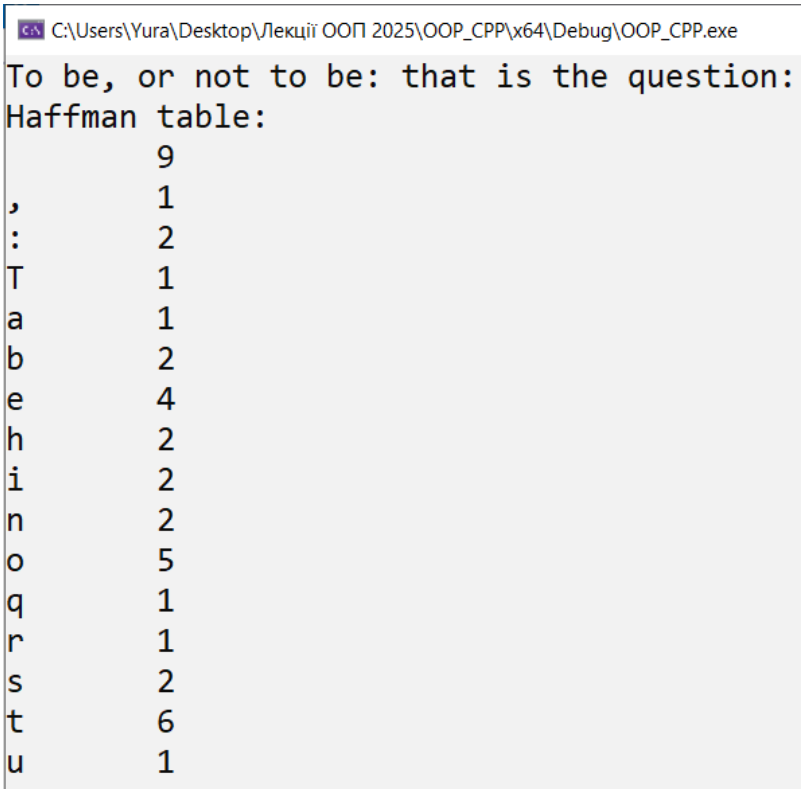
Результат роботи програми:

```
C:\Users\Yura\Desktop\Лекції ООП 2025\OOP_CPP\x64\Debug\OOP_CPP.exe
Enter an English word: cat
Переклад: кіт
```

Ще один приклад використання асоціативного масиву. Так, для реалізації алгоритму стиснення тексту методом Хаффмана, треба спочатку підрахувати, скільки разів кожен символ зустрічається у тексті.

```cpp
int main()
{
	setlocale(0, "");
	char text[] = "To be, or not to be: that is the question:";
	map<char, int> m;
	map<char, int>::iterator ii; // це вказівник!!
	for (int j = 0; j < strlen(text); j++)
		m[text[j]]++;
	cout << text << endl << "Haffman table:" << endl;
	for (ii = m.begin(); ii != m.end(); ii++)
		cout << ii->first << "\t" << ii->second << endl;

}
```

Результат роботи програми:



Основні властивості шаблонів класів

1. Компонентні функції параметризованого класу автоматично є параметризованими. Їх не обов'язково оголошувати як параметризрвані за допомогою template.
2. Дружні функції, які описуються в параметризованому класі, не є автоматично параметризованими функціями, тобто за замовчуванням такі функції є дружніми для всіх класів, які організовуються за даним шаблоном.
3. Якщо friend-функція містить у своєму описі параметр типу параметризованого класу, то для кожного створеного за даним шаблоном класу є власна friend-функція.
4. В рамках параметризованого класу не можна визначити friend-шаблони (дружні параметризрвані класи).
5. З одного боку, шаблони можуть бути похідними (успадковуватися) як від шаблонів, так і від звичайних класів, з іншого боку, вони можуть використовуватися в якості базових для інших шаблонів або класів.
6. Шаблони функцій, які є членами класів, не можна описувати як virtual.
7. Локальні класи не можуть містити шаблони в якості своїх елементів.

Резюме

1. Шаблони дозволяють досягти одну з найважчих цілей в програмуванні – створити багаторазово використовуваний код.
2. Використовуючи шаблони, можна створювати узагальнені функції і класи.
3. Узагальнена функція – це функція, що перевантажує сама себе.
4. Узагальнена функція створюється за допомогою ключового слова template.
5. Узагальнена функція повинна виконувати одну і ту ж дію для всіх версій – відмінність між версіями полягає тільки в типі даних.
6. Узагальнені класи особливо корисні в разі, коли в них використовується логіка, яку можна узагальнити.
7. Загальний формат оголошення узагальненого класу має такий вигляд: template <class Ttype> class ім'я_класу {...}
8. Узагальнені функції і класи дозволяють визначити загальний формат об'єкта, який можна потім використовувати з будь-яким типом даних.

Завдання до лабораторної роботи

1. Створити шаблон заданого класу. Визначити конструктори, деструктор, перевантажену операцію присвоювання ("=") та операції, що задані у варіанті завдання.
2. Написати програму тестування, в якій перевіряється використання шаблону для стандартних типів даних.
3. Визначити користувальницький клас, який буде використовуватися в якості параметра шаблону. Визначити в класі необхідні функції і перевантажені операції.
4. Написати програму тестування, в якій перевіряється використання шаблону для користувальницького типу.

Клас реалізувати як динамічний масив. Для цього визначення класу повинно мати такі поля:

- Покажчик на початок масиву;
- Максимальний розмір масиву;
- Поточний розмір масиву.

Щоб у вас не виникало проблем, акуратно працюйте з константними об'єктами. наприклад:

конструктор копіювання слід визначити так:

```
MyTmp (const MyTmp & ob);
```

операцію присвоювання перевантажити так:

```
MyTmp & operator = (const MyTmp & ob);
```

Для шаблонів масивів в якості стандартних типів використовувати цілі і дійсні типи. Для користувальницького типу взяти клас "Комплексне число" Complex.

```
class Complex
{
  int real; // дійсна частина
  int imag; // уявна частина
```

```
public;
// Необхідні функції і перевантажені операції
};
```

Тестування повинне бути виконане для всіх типів даних і для всіх операцій.

Варіанти завдань:

1. Клас - одновимірний масив . Додатково перевантажити наступні операції:
   << вивід на консоль;
   == порівняння.

2. Клас - одновимірний масив. Додатково перевантажити наступні операції:
   [ ] Оператор індексу;
   == порівняння.

3. Клас - одновимірний масив. Додатково перевантажити наступні операції:
   << вивід на консоль;
   + об'єднати два масива;

4. Клас - одновимірний масив . Додатково перевантажити наступні операції:
   [ ] Оператор індексу;
   + злиття двох масивів .

5. Клас - одновимірний масив. Додатково перевантажити наступні операції:
   << вивід на консоль;
   = = Перевірка на рівність масивів

6. Клас - односпрямований список list. Додатково перевантажити наступні операції:
   + Додати елемент в кінець списку (list + item);
   - - Видалити елемент з початку (--list);
   = = Перевірка на рівність.

7. Клас - односпрямований список list. Додатково перевантажити наступні операції:
   + - Додати елемент в початок списку (list + item);
   - - Видалити елемент з початку списку (--list);
   ! = - Перевірка на нерівність.

8. Клас - односпрямований список list. Додатково перевантажити наступні операції:
   + Додати елемент в кінець списку (list + item);
   -- Видалити елемент з кінця списку (типу list--);
   ! = Перевірка на нерівність.

9. Клас - односпрямований список list. Додатково перевантажити наступні операції:
   [] - Доступ до елементу в даній точці, наприклад:
   int i; Type c;
   list L;
   c = L [i];
   + Об'єднати два списки;

10. Клас - стек stack. Додатково перевантажити наступні операції:
   + Додати елемент в стек;
   -- Витягти елемент з стека;

Тести для закріплення матеріалу

1. Що описується в прикладі для класу
T T :: T (const T &) {}?
   а) абстрактний клас;
   б) шаблон функції;
   в) константних метод;
   г) конструктор копіювання.

2. Оголошення шаблону функції починається з ключового слова
   а) type;
   б) function;
   в) template;
   г) pattern.

3. Що описується в прикладі для класу
   template <class X> void swapargs(X &a, X &b)
   а) абстрактний клас;
   б) шаблон функції;
   в) константних метод;
   г) конструктор копіювання.

4. Чи можна шаблони функцій, які є членами класів, описувати як virtual?
   а) шаблони функцій, які є членами класів, можна описувати як virtual;
   б) можна тільки як abstract
   в) шаблони функцій, які є членами класів, не можна описувати як virtual;
   г) шаблони функцій не можуть бути членами класів.

5. Компонентні функції параметризованого класу обов'язково оголошувати як параметризрвані за допомогою template?

а) Їх не обов’язково оголошувати як параметризрвані.

б) Їх обов’язково оголошувати як параметризрвані.

в) При їх оголошенні як параметризрвані виникне помилка компіляції.

г) Виникне помилка Runtime error.

6. Дружні функції, які описуються в параметризованому класі, автоматично стають параметризованим функціями?

а) Так, автоматично стають параметризованим функціями.

б) Ні, не є автоматично параметризованим функціями.

в) При їх оголошенні як параметризрвані виникне помилка компіляції.

г) Їх не обов’язково оголошувати як параметризрвані.

ЧАСТИНА 2. ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ НА МОВІ C#

РОЗДІЛ 8. ВВЕДЕННЯ В C#

8.1 Мова C# і платформа .NET

На сьогоднішній момент мова програмування C# одна з найпотужніших і затребуваних мов в IT-галузі. На ній пишуться найрізноманітніші програми: від невеликих десктопних програм до великих веб-порталів і веб-сервісів, які обслуговують щодня мільйони користувачів.

У порівнянні з іншими мовами C# досить молода, але в той же час вона вже пройшов великий шлях. Перша версія мови вийшла разом з релізом Microsoft Visual Studio .NET в лютому 2002 року.

Поточна версія мови C#, що використовується у Visual Studio, залежить від версії самої Visual Studio та налаштувань проекту. Visual Studio 2022, наприклад, може підтримувати декілька версій C#, включаючи C# 10 та C# 11.

Мова C# є об'єктно-орієнтованої і в цьому плані багато перейняла у Java і C++. Наприклад, C# підтримує поліморфізм, успадкування, перевантаження операторів, статичну типізацію. Об'єктно-орієнтований підхід дозволяє вирішити завдання з побудови великих, але в той же час гнучких, масштабованих і розширюваних застосувань. C# продовжує активно розвиватися, і з кожною новою версією з'являється все більше цікавих можливостей, як, наприклад, лямбда-вирази, динамічне зв'язування, асинхронні методи тощо.

Незважаючи на те що C# є самодостатньою мовою програмування, у ній є особливий взаємозв'язок із середовищем виконання .NET.

Платформа .NET тривалий час розвивалася переважно як платформа для Windows під назвою .NET Framework. У першому і другоиу виданні цього посібника розглядалася саме ця платформа.

У 2019 році вийшла остання версія цієї платформи – .NET Framework 4.8. Вона більше не розвивається. З 2014 року компанія Microsoft стала розвивати альтернативну платформу – .NET Core, яка повинна була увібрати в себе всі можливості застарілого .NET Framework і додати нову функціональність. Потім Microsoft послідовно випустила ряд версій цієї платформи: .NET Core 1 (2016 рік), .NET Core 2 (2017 рік), .NET Core 3 (2019 рік), .NET 5 (2020 рік).

Починаючи з версії .NET 5 платформа стала називатися .NET (без використання "Core" в назві), що символізує об'єднання .NET Core, Mono і .NET Framework.

Поточною версією є платформа .NET 9.0, яка вийшла 12 листопада 2024 року. Платформа .NET 9.0 заснована на .NET Framework і відрізняється від неї модульністю, кросплатформенністю, можливістю застосування хмарних технологій.

Наявність взаємозв'язку з середовищем виконання .NET пояснюється двома причинами. По-перше, мова C# спочатку призначалася для створення коду, який повинен виконуватися в середовищі .NET. І по-друге, використовувані в C#

бібліотеки визначені в середовищі .NET. На практиці це означає, що C# і платформа (середовище) .NET Framework тісно пов'язані один з одним, хоча теоретично C# можна відокремити від середовища .NET. У зв'язку з цим дуже важливо мати хоча б саме загальне уявлення про середовище .NET і його значення для C#.

Про платформу .NET

Платформа (середовище) .NET забезпечує спільне використання різних мов програмування, а також безпеку, переносимість програм і загальну модель програмування для Windows.

Що ж стосується взаємозв'язку з C#, то середовище .NET визначає два дуже важливих елемента. Першим з них є *загальномовне середовище виконання* (Common Language Runtime – CLR). Це система, що управляє виконанням програм. CLR як складова частина середовища .NET підтримує багатомовне програмування, а також забезпечує переносимість та безпечне виконання програм.

Другим елементом середовища .NET є бібліотека класів. Ця бібліотека надає програмі доступ до середовища виконання.

Якщо програма обмежується засобами, обумовленими в бібліотеці класів .NET, то така програма може виконуватися всюди, де підтримується середовище .NET.

Принцип дії CLR

Середовище CLR управляє виконанням коду .NET. Діє воно за наступним принципом. Результатом компіляції програми на C# є не виконуваний код, а файл, що містить особливого роду псевдокод, так званий Microsoft Intermediate Language, MSIL (проміжна мова Microsoft). Псевдокод MSIL визначає набір інструкцій, що не залежать від конкретного процесора.

Призначення CLR – перетворити проміжний код у виконуваний код по ходу виконання програми. Отже, будь-яка програма, скомпільована в псевдокод MSIL, може бути виконана в будь-якому середовищі, де є реалізація CLR.

Саме таким чином частково досягається переносимість в середовищі .NET. Псевдокод MSIL перетвориться у виконуваний код за допомогою JIT-компілятора. Назва Just-in-time compilation (JIT) – компіляція «на льоту» відображає оперативний характер даного компілятора.

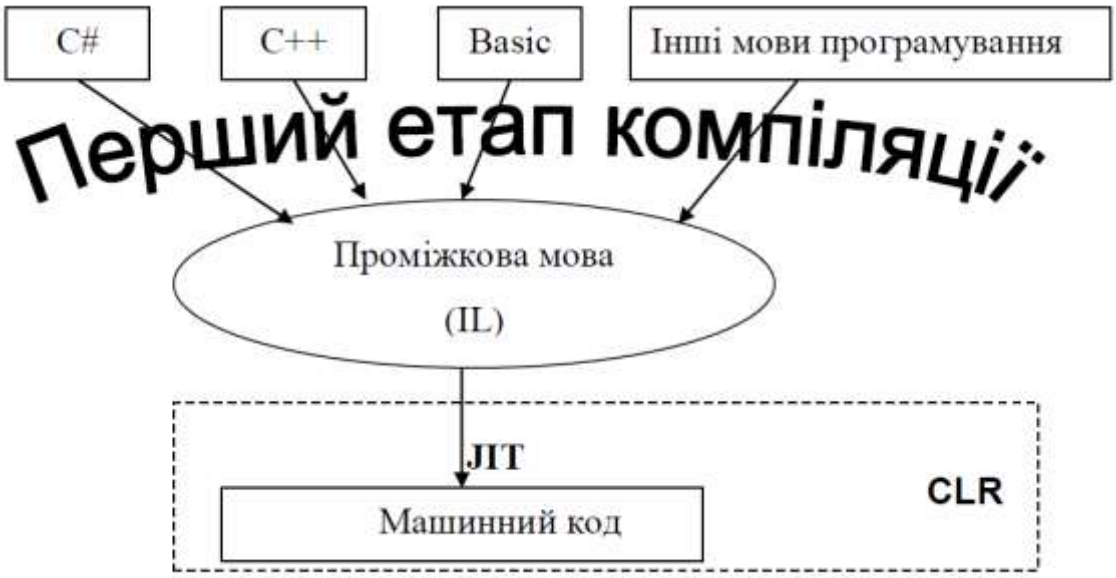


Рис. 8.1. Схема виконання програми у .NET

Процес перетворення зображено на рисунку 8.1. При виконанні програми середовище CLR активізує JIT-компілятор, який перетворює псевдокод MSIL (IL) во власний код системи на вимогу для кожної частини програми.

Таким чином, програма на C# фактично виконується як власний код, незважаючи на те, що спочатку вона скомпільована в псевдокод IL. Це означає, що така програма виконується так само швидко, як і в тому випадку, коли вона початково скомпільована во власний код, але в той же час вона набуває все переваги переносимості псевдокоду IL.

Крім псевдокоду IL, при компіляції програми на C# виходять також метадані, які служать для опису даних, які використовуються в програмі, а також забезпечують просте взаємодія одного коду з іншим. Метадані містяться в тому ж файлі, що і псевдокод IL.

Керований і некерований код

Як правило, при написанні програми на C# формується так званий керований код. Як пояснювалося вище, такий код виконується під управлінням середовища CLR, і тому на нього накладаються певні обмеження, хоча це і дає ряд переваг. Обмеження накладаються і задовольняться досить просто: компілятор повинен сформувати файл MSIL, призначений для виконання в середовищі CLR, використовуючи при цьому бібліотеку класів .NET, – і те й інше забезпечується засобами C#. До багатьох переваг керованого коду відносяться, зокрема, сучасні способи управління пам'яттю, можливість програмування на різних мовах, підвищення безпеки, підтримка управління версіями і чітка організація взаємодії програмних компонентів.

На відміну від керованого коду, некерований код не виконується в середовищі CLR.

Отже, до появи середовища .NET у всіх програмах для Windows застосовувався некерований код. Втім, керований і некерований коди можуть

взаємодіяти один з одним. Формування керованого коду в C# зовсім не означає, що на його можливість взаємодії з вже існуючими програмами накладаються якісь обмеження.

Загальна проміжна мова

Часто CIL називають псевдоасемблером, тому що вона визначає набір команд якогось процесора. Однак в даному випадку процесором є не шматок кремнію, а CLR. При написанні програм для .NET знати CIL потрібно не більш, ніж асемблер x86 для програмування під Windows. І все ж елементарні пізнання про CIL можуть бути корисні, якщо вам знадобиться дізнатися, чому якійсь метод FCL працює не так, як ви цього очікували. У вас немає вихідного коду FCL, але є CIL.

CIL містить приблизно 100 команд. Деякі з них – типові низькорівневі команди, аналогічні командам мікропроцесорів, наприклад, команди складання двох значень (ADD) або переходу, якщо два значення рівні (BEQ). Інші, більш високого рівня, рідко зустрічаються в апаратних наборах команд. Так, NEWOBJ створює екземпляр об'єкта, а THROW генерує виняток. Завдяки такому багатству набору команд CIL, код на мові високого рівня, такою як C# або Visual Basic .NET, часто породжує при компіляції дивно мале число команд.

CIL використовує стекову модель виконання. Якщо процесори x86 для обробки значень завантажують їх у регістри, то CLR поміщає їх в обчислювальний стек. Щоб скласти два числа, вони копіюються в стек, викликається ADD, і результат зчитується з стека. Копіювання значення з пам'яті в стек називається завантаженням (loading), а копіювання в зворотному напрямку – збереженням (storing).

У CIL є кілька команд завантаження і збереження. Наприклад, LDLOC завантажує в стек значення за деякою адресою в пам’яті, а STLOC копіює значення з стека в пам'ять, видаляючи його з стека.

Як приклад роботи CIL, розглянемо фрагмент програми C#, в якому оголошуються і ініціалізуються дві змінні, потім вони складаються, і сума записується в третю зміну:

```
int a = 3;
int b = 7;
int c = a + b;
```

Нижче наведено код CIL, згенерований компілятором Microsoft C# (з коментарями):

```
Idc.i4.3 // Завантажити в стек 32-розрядне число 3.
stloc.0 // Зберегти його в локальній змінній 0 (a).
Idc.i4.7 // Завантажити на стек 32-розрядний (i4) число 7,
stloc.1 // Зберегти його в локальній змінній 1 (b).
ldloc.0 // Завантажити в стек локальну змінну 0.
ldloc.1 // Завантажити в стек локальну змінну 1.
add // Скласти два числа і отримати суму в стеці.
stloc.2 // Зберегти суму в локальній змінній 2 (c).
```

Найбільш часто використовувані команди CIL та їх короткий опис наведені в таблиці 8.1:

Таблиця 8.1. Розповсюджені команди CIL.

Команда	**Опис**
BOX	Перетворює розмірний тип в контрольний.
CALL	Викликає метод; якщо метод віртуальний, то віртуальність ігнорується.
CALLVIRT	Викликає метод; якщо метод віртуальний, віртуальність враховується.
LDC	Завантажує в стек числову константу.
LDARGJA]	Завантажує в стек аргумент або його адресу [A].
LDELEM	Завантажує в стек елемент масиву,
LDLOCfA]	Завантажує в стек локальну змінну або її адресу [A],
LDSTR	Завантажує в стек строковий літерал.
NEWARK	Створює новий масив.
NEWOBJ	Створює новий об'єкт.
RET	Виконує повернення з методу.
STARG	Копіює в аргумент значення з стека,
THROW	Генерує виняток.
UNBOX	Перетворює контрольний тип в розмірний

Бібліотека класів .NET

С-програмісти, які пишуть для Windows, зазвичай використовують в своїх програмах виклики API (Application Programming Interface) Windows і DLL інших виробників. C++ – програмісти часто використовують бібліотеки класів, написані ними самими або стандартні бібліотеки типу MFC.

Працюючи з .NET, можна забути про всі ці застарілі API. Вам потрібно вивчити абсолютно новий API – бібліотеку класів .NET FCL (Framework Class Library), яка містить більше 7000 типів: класів, структур, інтерфейсів, перерахувань і делегатів.

Деякі класи FCL містять до 100 методів, властивостей та інших членів, так що вивчити повністю FCL досить важко. Щоб полегшити вивчення і використання FCL, Microsoft розділила цю бібліотеку на ієрархічні простори імен. Всього в FCL близько 100 таких просторів. У кожному містяться класи та інші типи, що мають деяке загальне призначення. Так, більша частина API Windows для управління вікнами інкапсульована в просторі імен System.Windows.Forms. Тут можна знайти класи, що представляють вікна, діалоги, меню та інші елементи, які зазвичай використовуються в застосуваннях з графічним інтерфейсом користувача. Простір імен System.Collections містить класи хеш-таблиць, масивів змінної розмірності та інших контейнерів, а System.IO – класи файлового введення-виведення.

FCL настільки величезна і всеосяжна, що для розробників немає потреби детально вивчати її повністю.

У таблиці 8.2 перераховані кілька просторів імен FCL і коротко описано їх вміст. Вираз «та ін.» позначає простори-нащадки. Наприклад, «System.Data та

ін.» – це System.Data, System.Data.Common, System.Data.OleDb, System.Data.SqlClient та System.Data.SqlTypes.

Таблиця 8.2. Деякі простори імен FCL.

Простір імен	**Вміст**
`System`	Фундаментальні типи даних і допоміжні класи
`System.Collections.Generic`	Класи узагальнених контейнерів
`SystemData та ін.`	Класи ADO.NET для доступу до даних
`System.Drawing`	Класи для виведення графіки (GDI +)
`System.IO`	Класи файлового і потокового введення-виведення
`System.Net`	Класи для роботи з мережевими протоколами, наприклад, з HTTP
`System.Reflection і ін.`	Класи для читання і запису метаданих
`System.Tbreading`	Класи для створення і управління потоками
`Syslem.Web`	Класи для підтримки HTTP
`System.Web.Services`	Класи для розробки Web-сервісів
`System.Web.Services.Protocols`	Класи для розробки клієнтів Web-сервісів
`System.Web.UI`	Основні класи, використовувані ASP.NET
`System.Web.UI.WebControls`	Серверні елементи управління ASP.NET
System.Windows.Forms	Класи для застосувань з графічним інтерфейсом користувача
System.Xml та ін.	Класи для читання і виведення даних в форматі XML

Перший, і найважливіший простір імен FCL, що використовується кожною програмою, – це System. Крім іншого, в ньому визначені фундаментальні типи даних, необхідні керованим застосуванням: байти, цілі, рядки тощо. Коли оголошується змінна типу int на C#, фактично створюється екземпляр System.Int32. Компілятор C# допускає скорочення int, тому що простіше написати:

```
int a=7;
```
ніж
```
Int32 a=7;
```

У просторі імен System знаходяться також багато типів винятків (наприклад, InvalidCastExceptiori), і такі корисні класи, такі як Math, що містить методи для виконання складних математичних операцій, Random, в якому реалізований генератор псевдовипадкових чисел і GC, що надає програмний інтерфейс прибиральника (збирача) сміття.

Фізично FCL являє собою набір DLL. Кожна DLL – це збірка, що завантажується CLR за запитом. Фундаментальні типи даних, такі як Int32, реалізовані в Mscorlib.dll, інші типи розкидані по різних DLL FCL. В документації на кожен тип вказана збірка, в якій він визначений. Це важливо,

оскільки ви зобов'язані вказати конкретну збірку, в якій реалізований клас FCL, якщо компілятор повідомляє про те, що цей клас є невизначеним типом.

8.2 Початок роботи. Visual Studio

Отже, створимо перше застосування на мові C#. Для створення застосувань на C# будемо використовувати безкоштовне і повнофункціональне середовище розробки – Visual Studio Community 2022, яке можна завантажити за наступною адресою: https://visualstudio.microsoft.com/ru/downloads/

При інсталяції Visual Studio на ваш комп'ютер будуть встановлені всі необхідні інструменти для розробки програм, в тому числі фреймворк .NET 9.0.

Після завершення установки створимо першу програму. Вона буде простенькою. Спочатку відкриємо Visual Studio і вгорі в рядку меню виберемо пункт File (Файл) -> New (Створити) -> Project (Проект). Перед нами відкриється діалогове вікно створення нового проекту: (Рис 8.2)

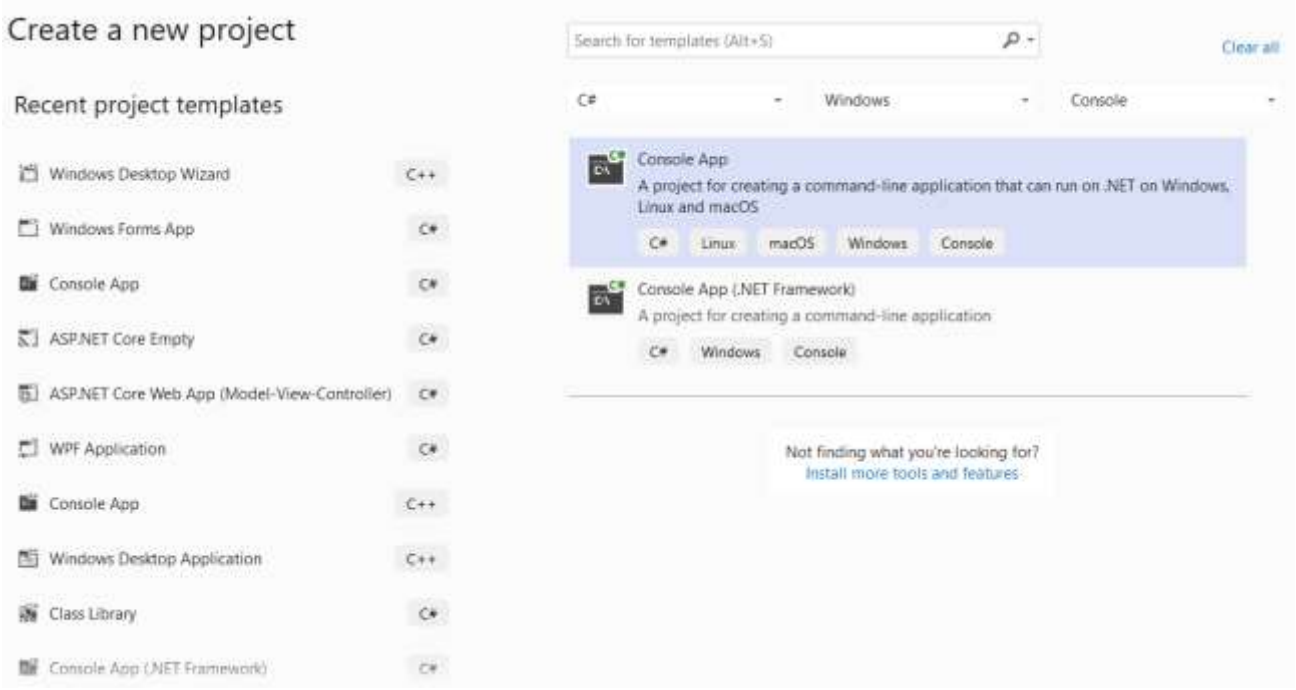


Рис. 8.2. Діалогове вікно створення нового проекту.

Виберемо пункт Console App, в поле Name дамо проекту назву ConsolApp1 і натиснемо ОК. Після цього Visual Studio створить і відкриє проект: (Рис. 8.3.)

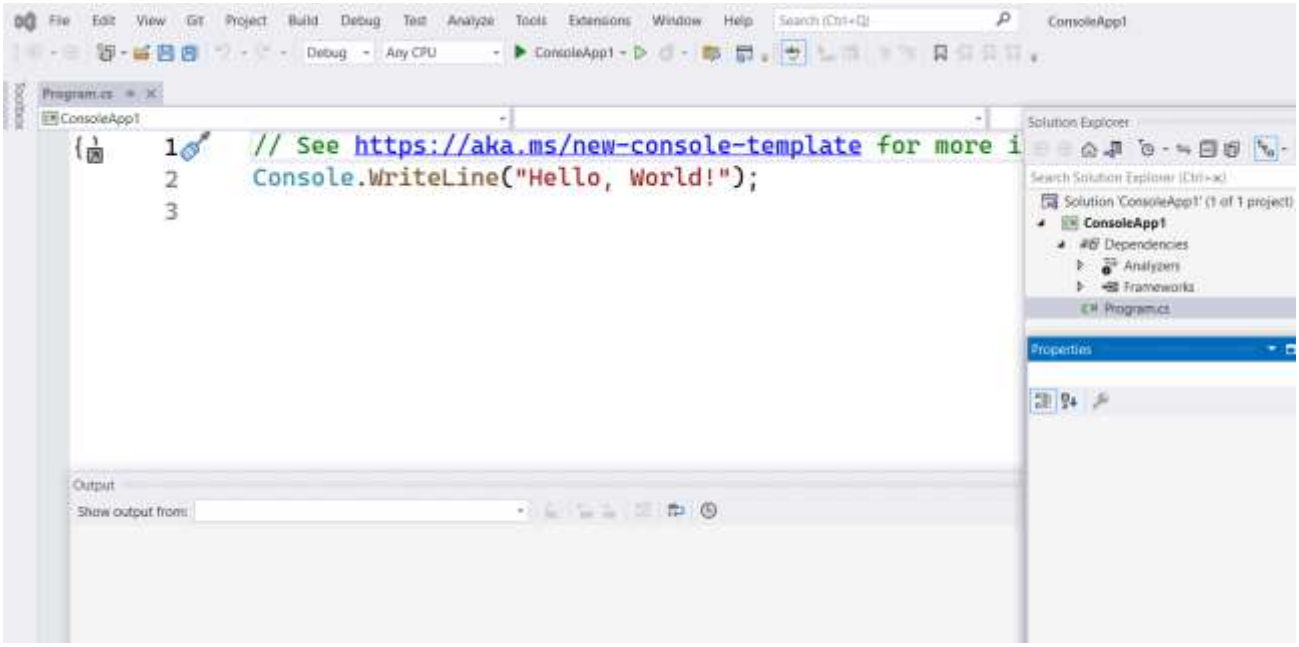


Рис. 8.3. Вигляд проекту Console App.

Якщо у діалоговому вікні створення нового проекту помітити пункт «Do not use top-level statement», з'являється більш звичне вікно проекту (рисунок 8.4)

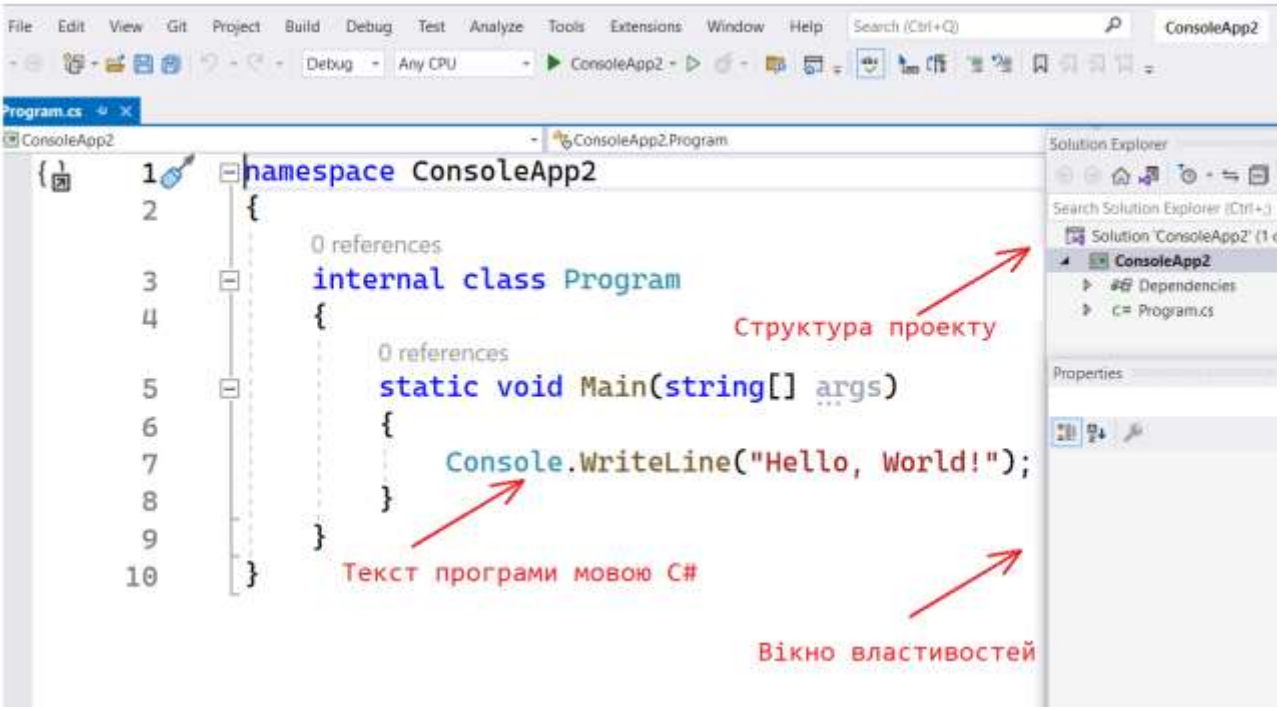


Рис. 8.4. Вигляд проекту Console App з опцією «Do not use top-level statement».

Файл коду програми Program.cs відкритий в центральному вікні. Розберемо, що весь цей код представляє:

```
namespace ConsoleApp2 // оголошення нового простору імен
{
     internal class Program // оголошення нового класу
     {
         static void Main(string[] args) // оголошення нового методу
         {
            Console.WriteLine("Hello, World!");
         } // Кінець оголошення нового методу
     } // Кінець оголошення нового класу
} // Кінець оголошення нового простору імен
```

Простори імен є організацію класів в загальні блоки. При створенні проєкту Console App під .NET 6.0 і вище, якщо увімкнений механізм ImplicitUsings (неявні директиви using), а він за замовчуванням увімкнений у нових шаблонах SDK-проєктів, за замовчуванням підключаються такі простори імен (namespaces):

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Net.Http;
using System.Threading;
using System.Threading.Tasks;
```

У .NET 6 і вище у типових проєктах Console App за замовчуванням увімкнено глобальні/неявні простори імен using.

Наприклад, using System, який містить фундаментальні і базові класи платформи .NET.

Простори імен, які автоматично доступні в Console App (через ImplicitUsings), фізично містяться в основних .NET-бібліотеках — зазвичай це збірки (бібліотеки) з *.dll. Наприклад, простір імен System фізично розташований у фізичній збірці (DLL) System.Private.CoreLib.dll.

C# має Сі-подібний синтаксис, кожен рядок завершується крапкою з комою, а кожен блок коду поміщається в фігурні дужки.

Далі починається наш простір імен, який буде створювати окрему збірку або програму, що виконується: спочатку йде ключове слово namespace, після якого назва простору імен. За замовчуванням Visual Studio дає йому назву проекту (у нас `ConsoleApp2`). Далі всередині фігурних дужок йде блок простору імен.

Простір імен може включати інші простори або класи. В даному випадку у нас за замовчуванням згенерований один клас – Program. Класи оголошуються схожим способом – спочатку йде ключове слово class, а потім назва класу, і далі блок самого класу в фігурних дужках.

Клас може містити різні змінні, методи, властивості, інші інструкції. В даному випадку у нас оголошений один метод Main. Зараз він містить команду

`Console.WriteLine("Hello, World!");` яка виводить на екран (консоль) текст «Hello, World!» . У програмі на C# метод Main є вхідною точкою програми, з нього починається все управління. Він обов'язково повинен бути присутнім в програмі.

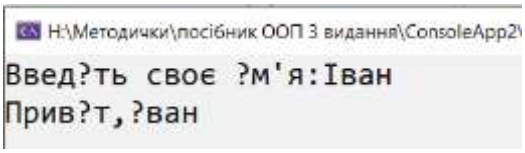
Слово static вказує, що метод Main – статичний, а слово void – що він не повертає ніякого значення.

Далі в дужках йдуть параметри методу string [] args – це масив args, який зберігає значення типу string, тобто рядки. В даному випадку вони нам поки не потрібні, але в реальній програмі це ті параметри, які передаються при запуску програми з консолі.

Тепер змінимо весь цей код на наступний:

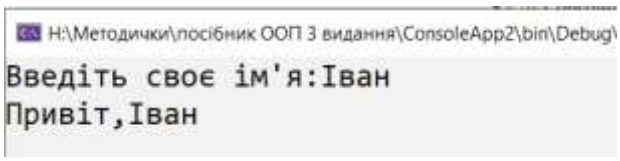
```
namespace ConsoleApp2
{
    internal class Program
    {
        static void Main(string[] args)
        {
          Console.Write("Введіть своє ім'я:");
          string name = Console.ReadLine(); // вводимо ім'я
          Console.WriteLine($"Привіт,{name}"); // виводимо на консоль
          Console.ReadKey();
      }
    }
}
```

Якщо запустити на виконання за допомогою клавіші F5 або з панелі інструментів, натиснувши на зелену стрілку, на консоль буде виведено:



Як бачите, не всі символи кирилиці виводятся коректно. Після додавання команди `Console.OutputEncoding = Encoding.UTF8;` всі символи кирилиці виводятся правильно.

```
        static void Main(string[] args)
        {
           Console.OutputEncoding = Encoding.UTF8;
           Console.Write("Введіть своє ім'я:");
           string name = Console.ReadLine(); // вводимо ім'я
           Console.WriteLine($"Привіт,{name}"); // виводимо на консоль
           Console.ReadKey();
      }
```



Отже, ми створили перше застосування. Ви його можете знайти в папці проекту в каталозі bin / Debug (воно буде називатися по імені проекту і мати розширення exe) і потім вже запускати без Visual Studio, а також переносити його на інші комп'ютери, де є .NET.

8.3 Базові поняття мови C#

C# має Сі-подібний синтаксис, тому ми не будемо в цьому посібнику розглядати такі елементи мови, як змінні, літерали, арифметичні операції, порозрядні операції, операції присвоювання, умовні конструкції, цикли. Якщо ви до цих пір не знайомі з цими синтаксичними конструкціями мови Сі, звертайтеся за допомогою до відповідних джерел. Ну, а ми зосередимося на на специфіці об'єктно-орієнтованої технології програмування мовою C#.

Типи даних

Як і в багатьох мовах програмування, в C# є своя система типів даних, яка використовується для створення змінних. Тип даних визначає внутрішнє представлення даних, множину значень, які може набувати об'єкт, і навіть допустимі дії, які можна застосовувати над об'єктом.

У мові C# є такі примітивні типи даних:

bool: зберігає значення true або false (логічні літерали). Представлений системним типом System.Boolean.

```
bool alive = true;
bool isDead = false;
```

byte: зберігає ціле число від 0 до 255 і займає 1 байт. Представлений системним типом System.Byte.

```
byte bit1 = 1;
byte bit2 = 102;
```

sbyte: зберігає ціле число від -128 до 127 і займає 1 байт. Представлений системним типом System.Sbyte.

```
sbyte bit1 = -101;
sbyte bit2 = 102;
```

short: зберігає ціле число від -32768 до 32767 і займає 2 байта. Представлений системним типом System.Int16.

```
short n1 = 1;
short n2 = 102;
```

ushort: зберігає ціле число від 0 до 65535 і займає 2 байта. Представлений системним типом System.UInt16.

```
ushort n1 = 1;
ushort n2 = 102;
```

int: зберігає ціле число від -2147483648 до 2147483647 і займає 4 байта. Представлений системним типом System.Int32. Всі цілочисельні літерали за замовчуванням представляють значення типу int:

```
int a = -10;
int b = 0b101; // бінарна форма b = 5
int c = 0xFF;  // шістнадцяткова форма c = 255
```

uint: зберігає ціле число від 0 до 4294967295 і займає 4 байта. Представлений системним типом System.UInt32.

```
uint a = 10;
uint b = 0b101; // бінарна форма b = 5
uint c = 0xFF;  // шістнадцяткова форма c = 255
```

long: зберігає ціле число від від –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807 і займає 8 байт Представлений системним типом System.Int64.

```
long a = -10;
long b = 0b101; // бінарна форма b = 5
long c = 0xFF;  // шістнадцяткова форма c = 255
```

ulong: зберігає ціле число від 0 до 18 446 744 073 709 551 615 і займає 8 байт. Представлений системним типом System.UInt64.

```
long a = 10;
long b = 0b101; // бінарна форма b = 5
long c = 0xFF;  // шістнадцяткова форма c = 255
```

float: зберігає число з плаваючою крапкою від -3.4 * 10^{38} до 3.4 * 10^{38} і займає 4 байта. Представлений системним типом System.Single.

double: зберігає число з плаваючою крапкою від -1.797 * 10^{324} до 1.797 * 10^{308} і займає 8 байтів. Представлений системним типом System.Double.

decimal: зберігає десяткове дробове число. Якщо вживається без коми, має значення від ± 1.0 * 10^{-28} до ± 7.9228 * 10^{28}, може зберігати 28 знаків після коми і займає 16 байт. Представлений системним типом System.Decimal.

char: зберігає одиночний символ в кодуванні Unicode і займає 2 байта. Представлений системним типом System.Char. Цьому типу відповідають символьні літерали:

```
char a = 'A';
char b = '\x5A';
char c = '\u0420';
```

string: зберігає набір символів Unicode. Представлений системним типом System.String. Цьому типу відповідають символьні літерали.

```
string hello = "Hello";
string word  = "world";
```

object: може зберігати значення будь-якого типу даних і займає 4 байта на 32-розрядної платформі і 8 байт на 64-розрядної платформі. Представлений системним типом System.Object, який є базовим для всіх інших типів і класів .NET.

```
object a = 22;
object b = 3.14;
object c = "hello code";
```

Наприклад, визначимо кілька змінних різних типів і виведемо їх значення на консоль:

```
using System;
namespace HelloApp
{
    class Program
    {
        static void Main(string[] args)
        {
            string name = "Вася";
            int age = 33;
            bool isEmployed = false;
            double weight = 78.65;
            Console.WriteLine($"Ім'я: {name}");
            Console.WriteLine($"Вік: {age}");
            Console.WriteLine($"Вага: {weight}");
            Console.WriteLine($"Працює: {isEmployed}");
            Console.ReadKey();
        }
    }
}
```

Для виведення даних на консоль тут застосовується інтерполяція: перед рядком ставиться знак $ і після цього ми можемо вводити в рядок в фігурних дужках значення змінних.

Консольний вивід програми:

Використання суфіксів

При присвоєнні значень треба мати на увазі наступний нюанс: всі дійсні літерали розглядаються як значення типу double. І щоб вказати, що дробове число представляє тип float або тип decimal, необхідно до літералу додавати суфікс: F / f - для float і M / m - для decimal.

```
float a = 3.14F;
float b = 30.6f;

decimal c = 1005.8M;
decimal d = 334.8m;
```

Подібним чином всі цілочисельні літерали розглядаються як значення типу int. Щоб явно вказати, що цілочисельний літерал являє значення типу uint, треба використовувати суфікс U / u, для типу long - суфікс L / l, а для типу ulong - суфікс UL / ul:

```
uint a = 10U;
long b = 20L;
ulong c = 30UL;
```

Використання системних типів

Вище при перерахуванні всіх базових типів даних для кожного згадувався системний тип. Тому що назва вбудованого типу по суті являє собою скорочене позначення системного типу. Наприклад, такі змінні будуть еквівалентні по типу:

```
int a = 4;
System.Int32 b = 4;
```

Неявна типізація

Раніше ми явно вказували тип змінних, наприклад, int x;. І компілятор при запуску вже знав, що x зберігає цілочисельне значення.

Однак ми можемо використовувати і модель неявній типізації:

```
var hello = "Hello, World";
var c = 20;
Console.WriteLine(c.GetType().ToString());
Console.WriteLine(hello.GetType().ToString());
```

Для неявної типізації замість назви типу даних використовується ключове слово var. Потім вже при компіляції компілятор сам виводить тип даних виходячи з присвоєного значення. В наведеному вище прикладі використовувався вираз

```
Console.WriteLine(c.GetType().ToString());,
```

який дозволяє дізнатися про тип змінної c. Через те, що за замовчуванням всі цілочисельні значення розглядаються як значення типу int, то в результаті змінна c матиме тип int або System.Int32.

Ці змінні подібні звичайним, однак вони мають деякі обмеження.

По-перше, ми не можемо спочатку оголосити неявно тіпізіруемую змінну, а потім ініціалізувати:

```
// цей код працює
int a;
a = 20;

// цей код не працює
var c;
c = 20;
```

По-друге, ми не можемо вказати в якості значення неявно тіпізіруемой змінної null:

```
// цей код не працює
var c = null;
```

Перетворення базових типів даних

При розгляді типів даних вказувалося, які значення може мати той чи інший тип і скільки байт пам'яті він може займати. Тепер застосуємо операцію складання до даних різних типів:

```
byte a = 4;
byte b = a + 70;  // помилка
```

Тут змінився тільки тип змінної, яка отримує результат складання – з int на byte. Однак при спробі скомпілювати програму ми отримаємо помилку на етапі компіляції. І якщо ми працюємо в Visual Studio, середовище підкреслить другий рядок червоною хвилястою лінією, вказуючи, що в ній помилка.

Справа в тому, що операція додавання (та й віднімання) повертає значення типу int, якщо в операції беруть участь цілочисельні типи даних з розрядністю менше або дорівнює int (тобто типи byte, short, int). Тому результатом операції a + 70 буде об'єкт, який має довжину в пам'яті 4 байта. Потім цей об'єкт ми намагаємося привласнити змінної b, яка має тип byte і в пам'яті займає 1 байт.

І щоб вийти з цієї ситуації, необхідно застосувати операцію перетворення типів:

```
byte a = 4;
byte b = (byte)(a + 70);
```

Операція перетворення типів передбачає зазначення в дужках того типу, до якого треба перетворити значення.

Звужувальні і розширювальні перетворення

Перетворення можуть бути **звужувальні** (**narrowing**) і **розширювальні** (**widening**). Розширювальні перетворення розширюють розмір об'єкта в пам'яті. наприклад:

```
byte a = 4;              // 0000100
ushort b = a;    // 0000000000000100
```

В даному випадку змінної типу ushort присвоюється значення типу byte. Тип byte займає 1 байт (8 біт), і значення змінної a в двійковому вигляді можна представити як: 00000100.

Значення типу ushort займає 2 байта (16 біт). І при присвоєнні змінної b значення змінної a розширюється до 2 байт: 0000000000000100.

Тобто значення, яке займає 8 біт, розширюється до 16 біт.

Звужувальні перетворення, навпаки, звужують значення до типу з меншою розрядністю.

```
byte a = 4;
byte b = (byte)(a + 70);
```

Тут змінної b, яка займає 8 біт, присвоюється значення ushort, яке займає 16 біт. Тобто з 0000000000000100 отримуємо 00000100. Таким чином, значення звужується з 16 біт (2 байт) до 8 біт (1 байт).

Явні і неявні перетворення

У випадку з розширюювальними перетвореннями компілятор за нас виконував всі перетворення даних, тобто перетворення були **неявними** (**implicit conversion**). Такі перетворення не викликають яких ось труднощів. Проте варто сказати пару слів про загальний механізм подібних перетворень.

Якщо відбувається перетворення від безнакового типу меншою розрядності до безнаковому типу більшої розрядності, то додаються додаткові біти, які мають значени 0. Це називається доповнення нулями або **zero extension**.

```
byte a = 4;              // 0000100
ushort b = a;    // 0000000000000100
```

Якщо відбувається перетворення до знакової типу, то бітове представлення доповнюється нулями, якщо число додатне, і одиницями, якщо число від'ємне. Останній розряд числа містить знаковий біт – 0 для додатних і 1 для від'ємних чисел.

Розглянемо перетворення додатного числа: (Рис. 8.5)

```
sbyte a = 4;             // 0000100
short b = a;     // 0000000000000100
```

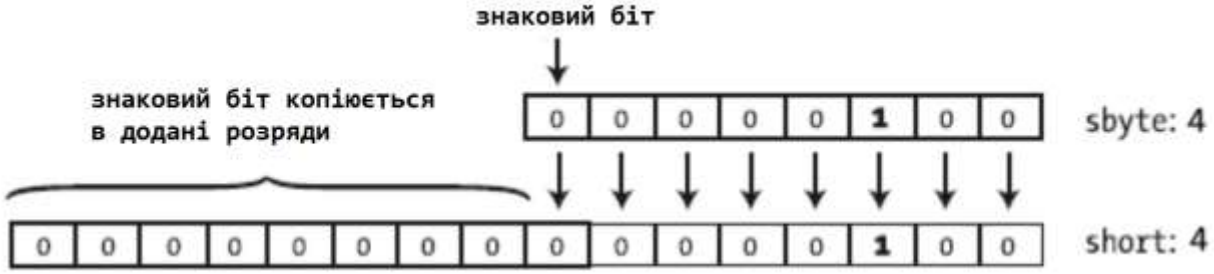


Рис. 8.5. Перетворення додатного числа.

Розглянемо перетворення від'ємного числа: (Рис. 8.6)

```
sbyte a = -4;            // 1111100
short b = a;    // 111111111111100
```

Рис. 8.6. Перетворення від'ємного числа

При **явних перетвореннях** (**explicit conversion**) ми самі повинні застосувати операцію перетворення (операція ()). Суть операції перетворення типів полягає в тому, що перед значенням вказується в дужках тип, до якого треба привести дане значення:

```
int a = 4;
int b = 6;
byte c = (byte)(a + b);
```

Все безпечні автоматичні перетворення представлені в таблиці 8.3:

Таблиця 8.3. Безпечні автоматичні перетворення

Тип	В які типи безпечно перетворюється
byte	short, ushort, int, uint, long, ulong, float, double, decimal
sbyte	short, int, long, float, double, decimal
short	int, long, float, double, decimal
ushort	int, uint, long, ulong, float, double, decimal
int	long, float, double, decimal
uint	long, ulong, float, double, decimal
long	float, double, decimal
ulong	float, double, decimal
float	double
char	ushort, int, uint, long, ulong, float, double, decimal

Розширювальні перетворення від типу з меншою розрядністю до типу з більшою розрядністю компілятор поводить неявно. Це можуть бути такі ланцюжки перетворень: (Рис. 8.7)

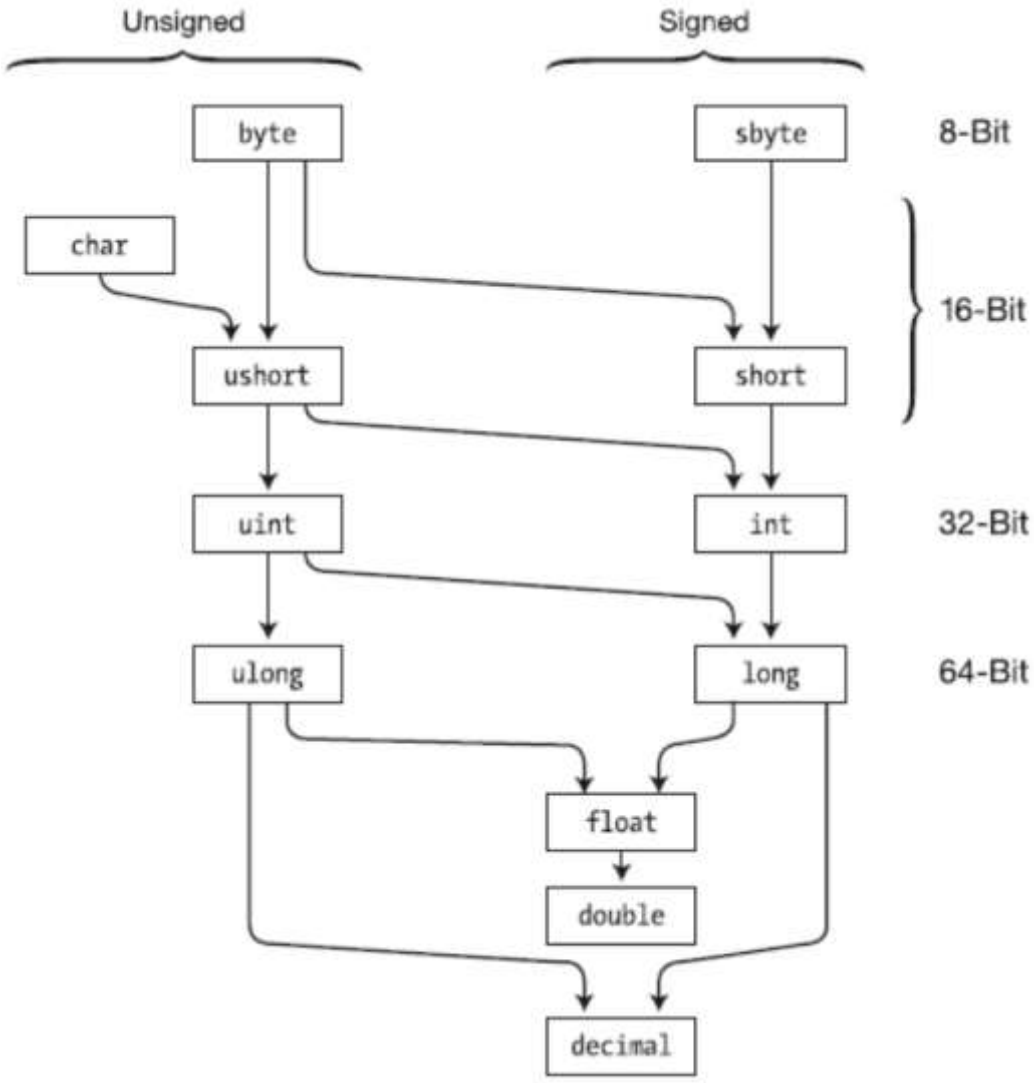


Рис. 8.7. Розширювальні перетворення від типу з меншою розрядністю до типу з більшою розрядністю

В інших випадках слід використовувати явні перетворення типів.

Також слід зазначити, що незважаючи на те, що і double, і decimal можуть зберігати дробові дані, а decimal має розрядність більше, ніж double, але все одно значення double потрібно явно приводити до типу decimal:

```
double a = 4.0;
decimal b = (decimal)a;
```

Втрата даних і ключове слово checked

Розглянемо ситуацію, яка виникне в наступному випадку:

```
int a = 33;
int b = 600;
byte c = (byte)(a + b);
```

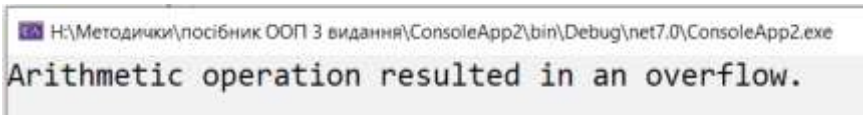
Результатом буде число 121, так число 633 не влучає у допустимий діапазон для типу byte, і старші біти будуть усікається. У підсумку вийде число 121. Тому при перетвореннях треба це враховувати. І ми в даному випадку можемо або взяти такі числа a і b, які в сумі дадуть число не більше 255, або ми можемо вибрати замість byte інший тип даних, наприклад, int.

Однак ситуації можуть бути різні. Ми можемо точно не знати, які значення будуть мати числа a і b. І щоб уникнути подібних ситуацій, в c# є ключове слово checked:

```
try
{
    int a = 33;
    int b = 600;
    byte c = checked((byte)(a + b));
    Console.WriteLine(c);
}
catch (OverflowException ex)
{
    Console.WriteLine(ex.Message);
}
```

При використанні ключового слова checked застосування ініціює виключення про переповнення. Тому для його обробки в даному випадку використовується конструкція try ... catch. У блок try ми включаємо дії, в яких може потенційно виникнути помилка, а в блоці catch обробляємо помилку.

Результат роботи програми:



Методи

По суті метод – це іменований блок коду, який виконує деякі дії.

Загальне визначення методів виглядає наступним чином:

[Модифікатори] тип_значення_що_повертається назва_методу ([параметри])
{
    // тіло методу
}

Модифікатори і параметри необов'язкові.

Наприклад, за замовчуванням консольна програма на мові C# має містити як мінімум один метод – метод Main, який є точкою входу в програму:

```
static void Main(string[] args)
{

}
```

Ключове слово static є модифікатором. Далі йде тип значення, що повертається. В даному випадку ключове слово void вказує на те, що метод нічого не повертає.

Визначимо ще пару методів:

```
class Program
{
   static void Main(string[] args)
    {
        SayHello();
        SayGoodbye();
    }

   static void SayHello()
    {
        Console.WriteLine("Hello");
    }
   static void SayGoodbye()
    {
        Console.WriteLine("GoodBye");
    }
}
```

Щоб використовувати методи SayHello і SayGoodbye в програмі, нам треба викликати їх в методі Main.

Перевагою методів є те, що їх можна повторно і багаторазово викликати в різних частинах програми.

Передача параметрів за посиланням і за значенням

Існує два способи передачі параметрів у метод в мові C#: за значенням і за посиланням.

Найбільш простий спосіб передачі параметрів представляє **передача за значенням**, по суті це звичайний спосіб передачі параметрів:

```
class Program
   {
       static void Main(string[] args)
       {
           Sum(10, 15); // параметри передаються за значенням
           Console.ReadKey();
       }
       static int Sum(int x, int y)
       {
           return x + y;
       }
   }
```

При передачі параметрів за посиланням перед параметрами використовується модифікатор **ref**:

```
class Program
{
    static void Main(string[] args)
    {
        int x = 10;
        int y = 15;
        Addition(ref x, y); // виклик методу
        Console.WriteLine(x); // 25

        Console.ReadLine();
    }
    // параметр x передається по посиланню
    static void Addition(ref int x, int y)
    {
        x += y;
    }
}
```

Зверніть увагу, що модифікатор ref вказується, як при оголошенні методу, так і при його виклику в методі Main.

У чому відмінність двох способів передачі параметрів? При передачі за значенням метод отримує не саму змінну, а її копію. А при передачі параметра за посиланням метод отримує адресу змінної в пам'яті. І, таким чином, якщо в методі змінюється значення параметра, переданого за посиланням, то також змінюється і значення змінної, яка передається на його місце.

Модифікатор out

Вище ми використовували вхідні параметри. Але параметри можуть бути також вихідними. Щоб зробити параметр вихідним, перед ним ставиться модифікатор out:

По суті, як і в випадку з ключовим словом ref, ключове слово out застосовується для для передачі аргументів за посиланням. Однак на відміну від ref для змінних, які передаються з ключовими словами out, не потрібно ініціалізація. І крім того метод, що викликається, повинен обов'язково надати їм значення.

```
class Program
{
    static void Main(string[] args)
    {
        int x = 10;
        int area;
        int perimetr;
        GetData(x, 15, out area, out perimetr);
        Console.WriteLine("Площа: " + area);
        Console.WriteLine("Периметр: " + perimetr);
        Console.ReadKey();
    }
```

```
    static void GetData(int x, int y, out int area, out int perim)
    {
        area = x * y;
        perim = (x + y) * 2;
    }
}
```

Резюме

1. Мова C# є об'єктно-орієнтованої і в цьому плані багато перейняла у Java і C++.
2. Незважаючи на те що C# є самодостатньою мовою програмування, у ній є особливий взаємозв'язок із середовищем виконання .NET Framework.
3. CLR як складова частина середовища .NET Framework підтримує багатомовне програмування, а також забезпечує переносимість та безпечне виконання програм.
4. Бібліотека класів надає програмі доступ до середовища виконання.
5. При написанні програми на C# формується керований код. Такий код виконується під управлінням середовища CLR.
6. У C# є своя система типів даних, яка використовується для створення змінних. Тип даних визначає внутрішнє представлення даних, множину значень, які може набувати об'єкт і допустимі дії, які можна застосовувати над об'єктом.
7. Тип **object** представлений системним типом System.Object, який є базовим для всіх інших типів і класів .NETі може зберігати значення будь-якого типу.
8. Для неявної типізації замість назви типу даних використовується ключове слово var.
9. Перетворення типів можуть бути звужувальні (narrowing) і розширювальні (widening).
10. Перетворення типів можуть бути неявними (implicit conversion) і явними (explicit conversion).
11. Існує два способи передачі параметрів в метод в мові C #: за значенням і за посиланням.
12. При передачі параметрів по посиланню перед параметрами використовується модифікатор ref і out.

Завдання до виконання

Завдання 1:

1. Скласти консольну програму на мові C# у Visual Studio 2019 - 2022 і запрограмувати оператори, які:
   - вводять з клавіатури значення дійсних змінних x, y, z;
   - розраховують вираз, зазначений в таблиці 8.2 відповідно варіанту;
   - виводять результат на екран.
2. Додати до складеної програми оператор циклу, що повторює введення даних і виконання розрахунків, доки не введено значення X=0.
3. Додати до складеної програми оператори, що виводять словами назву діапазону, в який попадає вхідне значення X та значення обчисленого виразу. Назви діапазонів для X: «менш за -10», «менш за -1», «більш за 1», «більш за 10». Назви діапазонів для результату: «негативне», «позитивне».

Варіанти завдань представлені в таблиці 8.4

Таблиця 8.4

№	Вид функции	№	Вид функции
1	$a = x + \dfrac{y^3}{z + z^2 + (z + y^2)}$	13	$a = \dfrac{x^2 + 4}{\sin^2 z + \dfrac{y}{2}}$
2	$a = (y - x) \cdot \dfrac{(y - z)(x - z)}{x + y}$	14	$a = \dfrac{x + z^2 + \dfrac{y}{4}}{z(y - 2) + \dfrac{1}{z^2 + 4}}$
3	$a = x + \dfrac{y}{z + \dfrac{1}{\ln(y)}}$	15	$a = \lvert\cos z + \cos y\rvert \cdot (1 + 2x^2)$
4	$a = x \cdot (arctg(z) + y^3)$	16	$a = x^2 + \sqrt{z^2 - y}$
5	$a = \sqrt{x^3 + y^2 + z}$	17	$a = \lg(\sqrt{y^3}) \cdot z + x$
6	$a = x^2 \cdot \sqrt[4]{z^2 + \lvert y\rvert}$	18	$a = 8 \cdot arctgy + \ln(\dfrac{z^2}{2}) + \lvert x\rvert$
7	$a = x + \dfrac{z^2}{3 + z \cdot \sin y}$	19	$a = \dfrac{\cos(y + z)}{\sqrt{z^2 + 12 - x^2}}$
8	$a = \ln(y + \sqrt{y + z^2}) \cdot z + y$	20	$a = \left(\sin\dfrac{7}{28} + \dfrac{y^2}{3 + \sqrt{z^3 + 16}}\right)\lvert x\rvert$

Завдання 2:

Створити програму, яка вирішує наступне завдання (відповідно варіанту):

1. Ресторан купує щодня масло т 1 кг по 8,50 грн/кг, сметану т 2 кг по 2,40 грн/кг, вершки т 3 кг по 4,10 грн/кг. Визначити суми, необхідні для закупівлі окремих продуктів і загальну суму.

2. Скільки секунд мають доба, тиждень, рік?
3. Ціни на два види товарів виросли на p відсотків. Вивести старі й нові ціни.

4. Швидкість світла становить 299792 км/с. Яку відстань проходить світло за годину, добу?

5. Ввести врожайність трьох сортів пшениці (36, 40, 44 т/га) і розміри трьох відповідних полів (у га). Скільки зібрали пшениці з кожного поля й зі всіх разом?

6. Радіус Місяця 1740 км. Обчислити його площу поверхні та та об'єм:
$S = 4\pi r^2$
$V = (4/3)\pi r^3$

Тести для закріплення матеріалу

1. Яким чином можна оголосити змінну строковим типом?

    а) var string;
    б) text stroka;
    б) stroka string;
    г) string stroka.

2. Що описується в методах?

    а) змінні;
    б) класи;
    в) поля;
    г) виконання дій.

3. Що буде виведено на екран після виконання програми?

```
static void ViewPerson(string Name, string Family)
        {
            Console.WriteLine($"Премію отримає: {Name} {Family}");
        }
          static void Main(string[] args)
        {
            ViewPerson("Ivan", "Ivanenko");
            Console.ReadKey();
        }
```

    а) помилка;
    б) рядок "Премію отримає:";
    в) рядок "Премію отримає: Name Family";
    г) рядок "Премію отримає: Ivan Ivanenko".

4. Вкажить, як правильно визначити неявно тіпізіруемую змінну?

    а) `int a;  a = 20;`
    б) `var a = 20;`
    в) `var a = null;`
    г) `var a;  a = 20;`

5. Яке перетворення типів здійснюється при виконанні наступного кода?

```
byte a = 66;
byte b = (byte)(a + 90);
```

а) Звужувальні неявне перетворення.
б) Звужувальні явне перетворення.
в) Розширювальне неявне перетворення.
г) Розширювальне явне перетворення.

6. Що буде буде після виконання програми?

```
try
  {
      int a = 33;
      int b = 600;
      byte c = checked((byte)(a + b));
      Console.WriteLine(c);
  }

  catch (OverflowException ex)
  {
      Console.WriteLine(ex.Message);
  }
```

а) явне перетворення зі втратою даних;
б) звужувальні явне перетворення;
в) повідомлення про виключну ситуацію у зв’язку з переповненням
г) розширювальне явне перетворення.

РОЗДІЛ 9. КЛАСИ І ОБ'ЄКТИ

9.1 Визначення класу

C# є повноцінною об'єктно-орієнтованою мовою. Це означає, що програму на C# можна представити у вигляді взаємопов'язаних взаємодіючих між собою об'єктів.

Описом об'єкта є клас, а об'єкт являє екземпляр цього класу. Можна ще провести наступну аналогію. У нас у всіх є деяке уявлення про людину, у якої є ім'я, вік, якісь інші характеристики. Тобто деякий шаблон - цей шаблон можна назвати класом. Конкретне втілення цього шаблону може відрізнятися, наприклад, одні люди мають одне ім'я, інші - інше ім'я. І реально існуюча людина (фактично екземпляр даного класу) буде представляти об'єкт цього класу.

За замовчуванням проект консольного застосування вже за замовчуванням містить один клас Program, з якого і починається виконання програми.

По суті клас представляє новий тип, який визначається користувачем. Клас визначається за допомогою ключового слова **class**:

```
class Person
{

}
```

Де визначається клас? Клас можна визначати всередині простору імен, поза простором імен, всередині іншого класу. Як правило, класи поміщаються в окремі файли. Але в даному випадку помістимо новий клас в файлі, де розташовується клас Program. Тобто файл Program.cs буде виглядати наступним чином

```
namespace HelloApp
{
   class Person
    {

    }

    internal class Program
    {
        static void Main(string[] args)
        {

        }
    }
}
```

Вся функціональність класу представлена його членами - полями (полями називаються змінні класу), властивостями, методами, ітераторами, подіями. Наприклад, визначимо в класі Person поля і метод:

```
namespace ConsoleApp
{
    class Person
    {
        public string name; // ім'я
        public int age; // вік

        public void GetInfo()
        {
            Console.WriteLine($"Ім'я: {name} Вік: {age}");
        }
    }
    internal class Program
    {
        static void Main(string[] args)
        {
            Person vasya; // це поки тільки ім'я об'єкта (посилання)

            Console.ReadKey();
        }
    }
}
```

В даному випадку клас Person представляє людину. Поле name зберігає ім'я, а поле age – вік людини. Метод GetInfo виводить всі дані на консоль. Щоб всі дані були доступні поза класом Person змінні і метод визначені з модифікатором public.

Оскільки клас являє собою новий тип, то в програмі ми можемо визначати змінні, які представляють даний тип. Так, тут в методі Main визначена змінна vasya, яка представляє клас Person. Але поки ця змінна не вказує ні на який об'єкт і за замовчуванням вона має значення null. Тому спочатку необхідно створити об'єкт класу Person.

9.2 Конструктори

Конструктор ініціалізує об'єкт при його створенні. Конструктор – це метод класу. У конструктора таке ж ім'я, як і у його класу. Конструктори ніколи не повертають значення, тому для них не вказується тип значення, що повертається.

Конструктор за замовчуванням

Якщо в класі не визначено жодного конструктора, то для цього класу автоматично створюється конструктор за замовчуванням. Такий конструктор не має парамтров і не має тіла.

Вище клас Person не має ніяких конструкторів. Тому для нього автоматично створюється конструктор за замовчуванням. І ми можемо використовувати цей конструктор. Зокрема, створимо один об'єкт класу Person:

```
static void Main(string[] args)
{
    Person vasya = new Person();
    vasya.GetInfo(); // Ім'я: Вік: 0
    vasya.name = "Вася";
    vasya.age = 34;
    vasya.GetInfo(); // Ім'я: Вася Вік: 34

    Console.ReadKey();
}
```

Для створення об'єкта Person використовується вираз new Person(). Оператор **new** виділяє пам'ять для об'єкта Person. І потім викликається конструктор за замовчуванням, який не приймає ніяких параметрів. У підсумку після виконання цього виразу в пам'яті буде виділено ділянку, де будуть зберігатися всі дані об'єкта Person. А змінна vasya отримає посилання на створений об'єкт.

Якщо конструктор НЕ ініціалізує значення змінних об'єкта, то вони отримують значення за замовчуванням. Для змінних числових типів це число 0, а для типу string і класів – це значення null (тобто фактично відсутність значення).

Після створення об'єкта ми можемо звернутися до змінних об'єкта Person через змінну vasya і встановити або отримати їх значення, наприклад, `vasya.name = "Вася";`.

Консольний вивід даної програми:

Хто будує конструктор умовчання?

Конструктор умовчання (конструктор з порожнім списком параметрів) будується транслятором за замовчуванням. Якщо клас не містить явних оголошень конструкторів, саме цей конструктор бере на себе роботу по перетворенню області пам'яті в об'єкт.

Створення конструкторів

Вище для ініціалізації об'єкта використовувався конструктор за замовчуванням.

Ми можемо самі визначити свої конструктори. Однак спроба оголосити **будь-який** варіант конструктора (з параметрами або без) призводить до того, що транслятор перестає займатися побудовою власних версій конструкторів. Відтепер в класі немає більше конструктора умовчання. Тепер все залежить від

відповідності оператора визначення об'єкта, що побудований нашим конструктором!

Одного разу взявшись за справу оголошення конструкторів, розробник класу повинен брати на себе відповідальність за створення **всіх** без винятку варіантів конструкторів, які можуть знадобитися для вирішення поставленого завдання. Такі правила.

```
class Person
{
    public string name;
    public int age;
    // 1 конструктор
    public Person() { name = "Вася"; age = 18; }
    // 2 конструктор
    public Person(string n) { name = n; age = 18; }
    // 3 конструктор
    public Person(string n, int a) { name = n; age = a; }

    public void GetInfo()
    {
        Console.WriteLine($"Ім'я: {name} Вік: {age}");
    }
}
```

Тепер в класі визначено три конструктора, кожен з яких приймає різну кількість параметрів і встановлює значення полів класу. Використовуємо ці конструктори:

```
    static void Main(string[] args)
    {
        // виклик 1-го конструктора без параметрів
        Person vasya = new Person();
        // виклик 2-го конструктора з одним параметром
        Person tom = new Person("Томас");
        // виклик 3-го конструктора з двома параметрами
        Person  sam = new Person("Самюель", 25);

        vasya.GetInfo(); // Ім'я: Вася Вік: 18
        tom.GetInfo(); // Ім'я: Томас Вік: 18
        sam.GetInfo(); // Ім'я: Самюель Вік: 25

        Console.ReadKey();
    }
```

Консольний вивід даної програми:

Якщо в класі визначено кілька конструкторів, то при створенні об'єкта необхідно використовувати один з цих конструкторів.

Ключове слово this

Ключове слово this являє собою посилання на поточний екземпляр класу. В яких ситуаціях воно нам може стати в нагоді? В наведеному вище прикладі визначено три конструктора. Всі три конструктора виконують однотипні дії - встановлюють значення полів name і age. Але цих повторюваних дій могло бути більше. І ми можемо не дублювати функціональність конструкторів, а просто звертатися з одного конструктора до іншого через ключове слово this, передаючи потрібні значення для параметрів:

```
class Person
{
    public string name;
    public int age;

    public Person() : this("Вася")
    {
    }
    public Person(string name) : this(name, 18)
    {
    }
    public Person(string name, int age)
    {
        this.name = name;
        this.age = age;
    }
    public void GetInfo()
    {
        Console.WriteLine($"Ім'я: {name} Вік: {age}");
    }
}
```

В даному випадку перший конструктор викликає другий, а другий конструктор викликає третій. За кількістю і типом параметрів компілятор дізнається, який саме конструктор викликається. Наприклад, у другому конструкторі:

```
    public Person(string name) : this(name, 18)
    {
    }
```

йде звернення до третього конструктору, якому передаються два значення. Причому з початку буде виконуватися саме третій конструктор, і тільки потім код другого конструктора.

Також варто відзначити, що в третьому конструкторі параметри називаються також, як і поля класу:

```
public Person(string name, int age)
{
    this.name = name;
    this.age = age;
}
```

і щоб розмежувати параметри і поля класу, до полів класу звернення йде через ключове слово this.

Так, в вираженні this.name = name; перша частина this.name означає, що name - це поле поточного класу, а не назву параметра name. Якби у нас параметри і поля називалися по-різному, то використовувати слово this було б необов'язково. Також через ключове слово this можна звертатися до будь-якого полю або методу.

Ініціалізатори об'єктів

Для ініціалізації об'єктів класів можна застосовувати ініціалізатори. Ініціалізатори представляють передачу в фігурних дужках значень доступним полям і властивостями об'єкта:

```
Person vasya = new Person { name = "Вася", age = 31 };
vasya.GetInfo();          // Ім'я: Вася  Вік: 31
```

За допомогою ініціалізатора об'єктів можна присвоювати значення всім доступним полям і властивостям об'єкта в момент створення без явного виклику конструктора.

При використанні ініціалізаторів слід враховувати наступні моменти:

- За допомогою ініціалізаторів ми можемо встановити значення тільки доступних з зовнішнього коду полів і властивостей об'єкта. Наприклад, в прикладі вище поля name і age мають модифікатор доступу public, тому вони доступні з будь-якої частини програми.

- Ініціалізатор виконується після конструктора, тому якщо і в конструкторі, і в ініціалізаторі встановлюються значення одних і тих же полів і властивостей, то значення, що встановлюються в конструкторі, замінюються значеннями з ініціалізатора.

9.3 Структури

Поряд з класами структури представляють ще один спосіб створення власних типів даних в C#. Більш того багато примітивних типів, наприклад, char, int, long, float, double, по суті є структурами.

Наприклад, визначимо структуру, яка представляє людину:

```
struct Human
{
    public string name;
    public int age;

    public void DisplayInfo()
    {
        Console.WriteLine($"Name: {name}  Age: {age}");
    }
}
```

Як і класи, структури можуть зберігати стан у вигляді змінних і визначати поведінку у вигляді методів. Так, в даному випадку визначено дві змінні - name і age для зберігання відповідно імені і віку людини і метод DisplayInfo для виведення інформації про людину.

Використовуємо цю структуру в програмі:

```
namespace ConsoleApp2
{
    struct Human
    {
        public string name;
        public int age;
        public void DisplayInfo()
        {
            Console.WriteLine($"Name: {name}  Age: {age}");
        }
    }

    internal class Program
    {
        static void Main(string[] args)
        {
            Human tom;
            tom.name = "Томас";
            tom.age = 3;
            tom.DisplayInfo();

            Console.ReadKey();
        }
    }
}
```

В даному випадку створюється об'єкт tom. У нього встановлюються значення глобальних змінних, і потім виводиться інформація про нього.

```
H:\Методички\посібник ООП 3 видання\ConsoleApp2\bin\[
Name: Томас  Age: 3
```

Конструктори структури

Як і клас, структура може визначати констуктор. Але на відміну від класу нам не обов'язково викликати конструктор для створення об'єкта структури:

```
Human tom;
```

Однак якщо ми таким чином створюємо об'єкт структури, то обов'язково треба проініціалізувати все поля (глобальні змінні) структури перед отриманням їх значень або перед викликом методів структури. Тобто, наприклад, в наступному випадку ми отримаємо помилку, тому що звернення до полів і методів відбувається до присвоєння їм початкових значень:

```
Human tom;
int x = tom.age;  // error
tom.DisplayInfo();// error
```

Також ми можемо використовувати для створення структури конструктор за замовчуванням, при виклику якого полям структури буде присвоєно значення за замовчуванням (наприклад, для числових типів це число 0):

```
Human tom = new Human();
tom.DisplayInfo();  // Name:   Age: 0
```

Також ми можемо визначити свої конструктори. Наприклад, змінимо структуру Human:

```
namespace ConsoleApp2
{
    struct Human
    {
        public string name;
        public int age;
        public Human(string name, int age)
        {
            this.name = name;
            this.age = age;
        }
        public void DisplayInfo()
        {
            Console.WriteLine($"Name: {name}  Age: {age}");
        }
    }

    internal class Program
    {
        static void Main(string[] args)
        {
            Human tom = new Human("Tom", 34);
            tom.DisplayInfo();
```

```
            User bob = new Human();
            bob.DisplayInfo();

            Console.ReadKey();
        }
    }
}
```

Важливо враховувати, що якщо ми визначаємо конструктор в структурі, то він повинен ініціювати все поля структури, як в даному випадку встановлюються значення для змінних name і age.

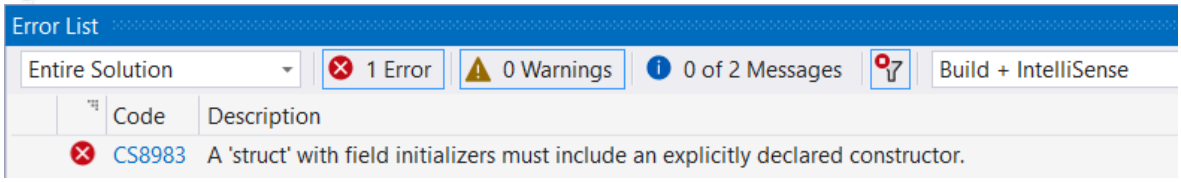
У структури конструктори без параметрів не існують! Для них оголошуються тільки параметризрвані конструктори.

Також, як і для класу, можна використовувати ініціалізатор для створення структури:

```
Human person = new Human { name = "Вася", age = 31 };
```

Але на відміну від класу не можна ініціалізувати поля структури безпосередньо при їх оголошенні, наприклад, наступним чином:

```
struct Human
{
    public string name="Вася";
    public int age=23;
    0 references
    public void DisplayInfo()
    {
        Console.WriteLine($"Name: {name}  Age: {age}");
    }
}
```



9.4 Типи значень і посилальні типи

Раніше ми розглядали такі елементарні типи даних: int, byte, double, string, object і інші. Також є складні типи: структури, класи. Всі ці типи даних можна розділити на типи значень (value types) і посилальні типи (reference types). Важливо розуміти між ними відмінності.

Типи значень:

- Цілочисельні типи (byte, sbyte, char, short, ushort, int, uint, long, ulong);
- Типи з плаваючою комою (float, double);
- Тип decimal;
- Тип bool;

- перерахування enum;
- Структури (struct).

Посилальні типи:

- Тип object;
- Тип string;
- Класи (class);
- Інтерфейси (interface);
- Делегати (delegate);

На рисунку 9.1 зображена система типів .NET.

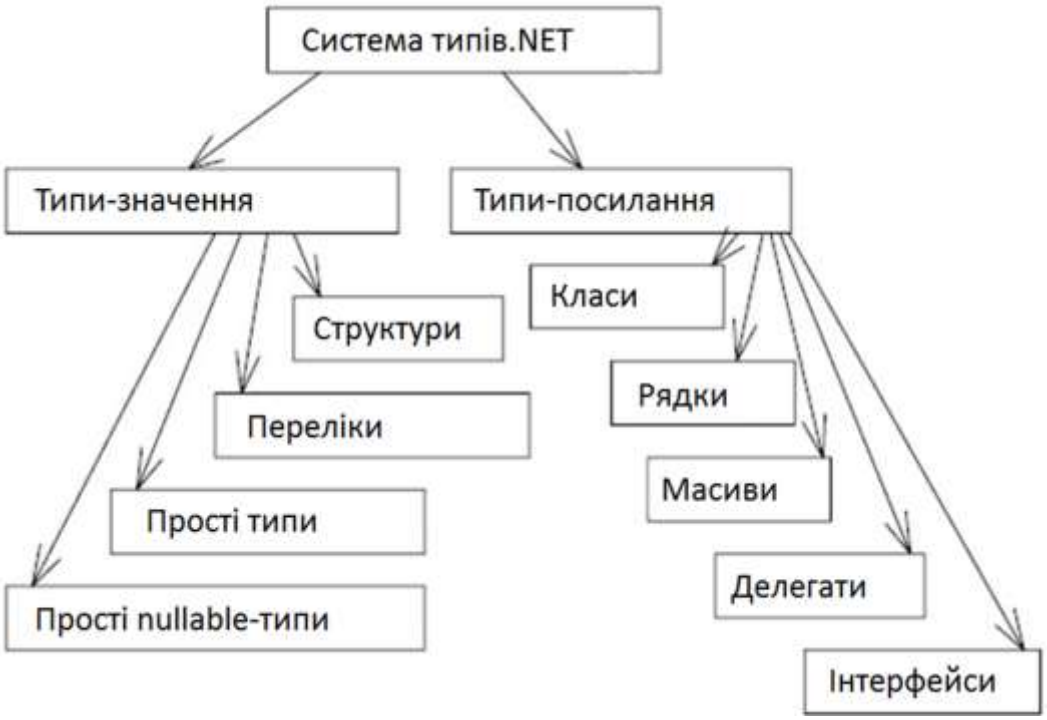


Рис. 9.1. Система типів .NET.

У чому ж між ними відмінності? Для цього треба зрозуміти організацію пам'яті в .NET. Тут пам'ять ділиться на два типи: стек і купа (heap). Параметри і змінні методу, які представляють типи значень, розміщують своє значення в стеку. Стек являє собою структуру даних, яка росте знизу вгору: кожен новий елемент, поміщаються поверх попереднього. Час життя змінних таких типів обмежена їх контекстом. Фізично стек - це деяка область пам'яті в адресному просторі.

Коли програма тільки запускається на виконання, в кінці блоку пам'яті, зарезервованого для стека встановлюється покажчик стека. При приміщенні даних у стек покажчик заново таким чином, що знову вказує на нове вільне місце. При виклику кожного окремого методу в стеці буде виділятися область пам'яті або фрейм стека, де будуть зберігатися значення його параметрів і змінних.

Наприклад:

```
class Program
{
    static void Main(string[] args)
    {
        Calculate(5);
        Console.ReadKey();
    }

    static void Calculate(int t)
    {
        int x = 6;
        int y = 7;
        int z = y + t;
    }
}
```

При запуску такої програми в стеці (рис. 9.2) будуть визначатися два фрейми – для методу Main (бо він викликається при запуску програми) і для методу Calculate:

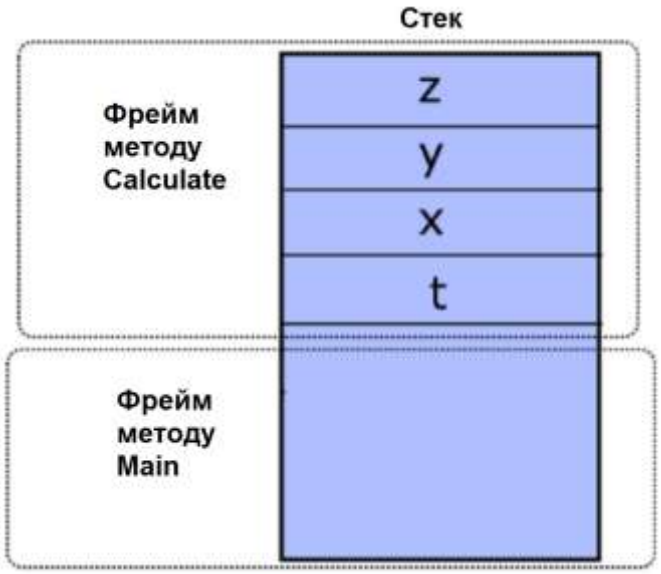


Рис. 9.2. Фрейми стеку.

При виклику цього методу Calculate в його фрейм в стеці будуть поміщатися значення t, x, y і z. Вони визначаються в контексті даного методу. Коли метод відпрацює, всі ці змінні знищуються, і пам'ять у стеці очищується.

Причому якщо параметр або змінна методу представляє тип значень, то в стеці буде зберігатися непосредсвенно значення цього параметра або змінної. Наприклад, в даному випадку змінні і параметр методу Calculate становлять тип-значення – тип int, тому в стеці будуть зберігатися їх числові значення.

Посилальні типи зберігаються в купі (heap), яку можна уявити як невпорядкований набір різнорідних об'єктів. Фізично це інша частина пам'яті, яка доступна процесу.

При створенні об'єкта посилального типу в стеці міститься посилання на адресу в купі. Коли об'єкт посилального типу перестає використовуватися, то посилання зі стека видаляється і пам'ять очищається. Після цього в справу вступає автоматичний збирач сміття: він бачить, що на об'єкт в купі немає більше посилань, видаляє цей об'єкт і очищає пам'ять.

Так, зокрема, якщо ми змінимо метод Calculate наступним чином:

```
static void Calculate(int t)
{
    object x = 6;
    int y = 7;
    int z = y + t;
}
```

то тепер значення змінної x буде зберігатися в купі, тому що вона являє собою посилальний тип object, а в стеку буде зберігатися посилання на об'єкт в купі. (Рис. 9.3.)

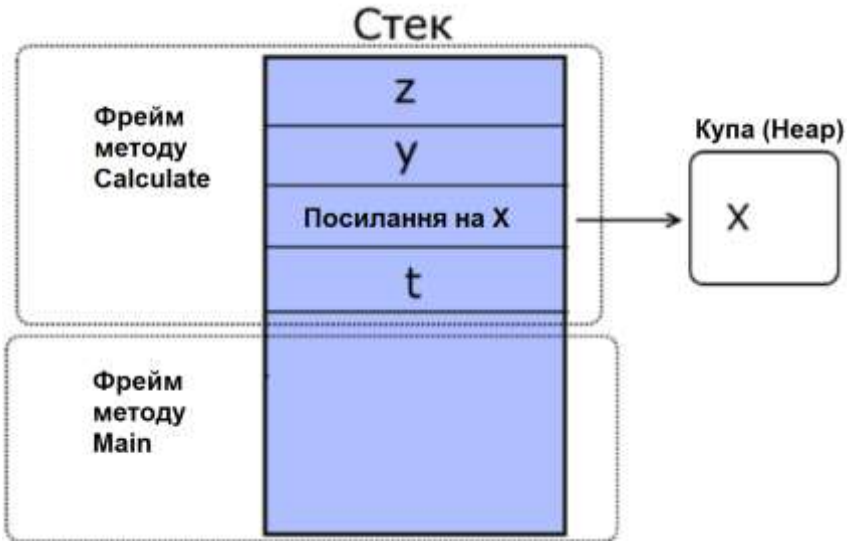


Рис. 9.3. Фрейми стеку. Значення змінної x зберегається в купі.

Складові типи

Тепер розглянемо ситуацію, коли тип значень і посилальний тип представляють складові типи – структуру і клас:

```
class Program
{
    static void Main(string[] args)
    {
        State state1 = new State(); // State - структура, її дані
                                    //розміщені в стеці
        Country country1 = new Country(); // Country - клас, в стек
        //міститься посилання на адресу в купі
        // а в купі розташовуються всі дані об'єкта country1
    }
}
```

```
struct State
{
    public int x;
    public int y;
    public Country country;
}
class Country
{
    public int x;
    public int y;
}
```

Тут в методі Main в стеці виділяється пам'ять для об'єкта state1. Далі в стеці створюється посилання для об'єкта country1 (Country country1), а за допомогою виклику конструктора з ключовим словом new виділяється місце в купі (new Country ()). Посилання в стеці для об'єкта country1 представлятиме адресу на місце в купі, за якою розміщений даний об'єкт. (Рис. 9.4.)

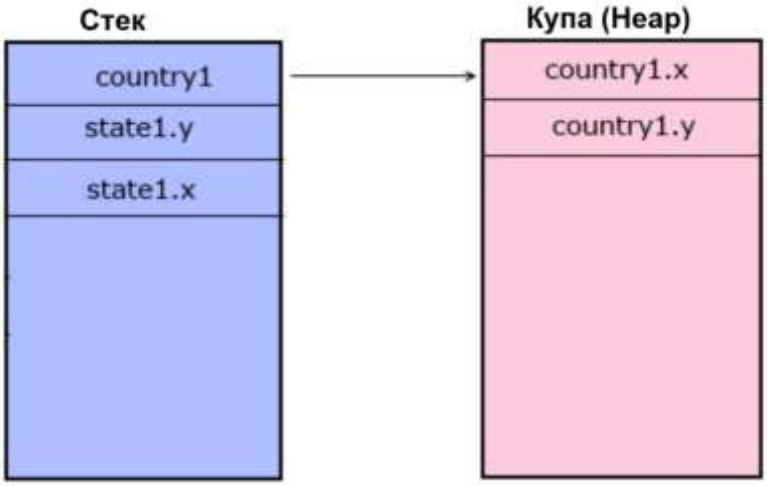


Рис.9.4. Посилання в стеці для об'єкта country1 представлятиме адресу в купі.

Таким чином, в стеці виявляться всі поля структури state1 і посилання на об'єкт country1 в купі.

Однак в структурі State також визначена змінна посилального типу Country. Де вона буде зберігати своє значення, якщо вона визначена в типі значень?

```
static void Main(string[] args)
{
    State state1 = new State();
    state1.country = new Country();
    Country country1 = new Country();
}
```

Значення змінної state1.country також буде зберігатися в купі, тому що ця змінна представляє посилальний тип (Рис. 9.5.)

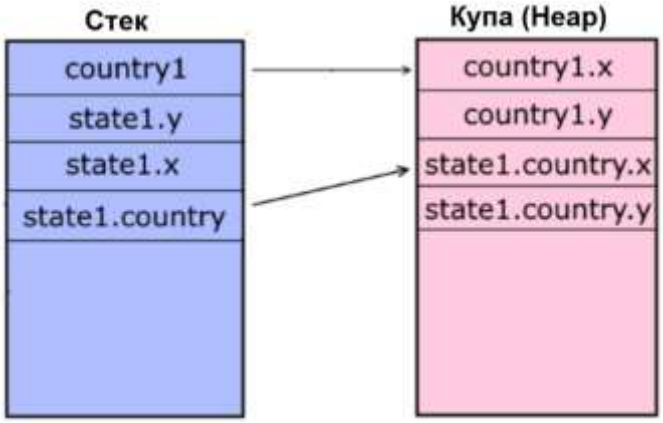


Рис. 9.5. Значення змінної state1.country також буде зберігатися в купі.

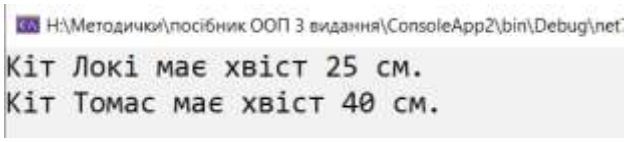
Копіювання значень

Тип даних треба враховувати при копіюванні значень. При присвоєнні даних об'єкту типу значень він отримує копію даних. При присвоєнні даних об'єкту посилального типу він одержує не копію об'єкта, а посилання на цей об'єкт в купі.

Наступний приклад демонструє відмінності між класами (посилальний тип) і структурами (тип значень).

```
struct Cat
{
    public string Name { get; set; }
    public int Tail { get; set; }
    public void Print()
    {
        Console.WriteLine($"Кіт {Name} має хвіст {Tail} см.");
    }
}

class Program
{
    static void Main(string[] args)
    {

        Cat a = new Cat { Name = "Локі", Tail = 25 };
        Cat b = a;
        b.Name = "Томас";
        b.Tail = 40;
        a.Print();
        b.Print();
        Console.ReadKey();
    }
}
```

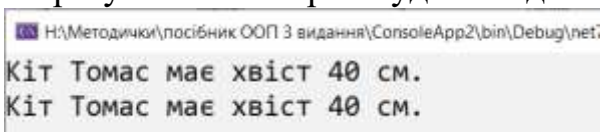
В результаті на екран буде виведено:



А тепер, увага! В попередньому коді ми поміняли тільки ключове слово **struct** на ключове слово **class**:

```
class Cat
{
    public string Name { get; set; }
    public int Tail { get; set; }
    public void Print()
    {
        Console.WriteLine($"Кіт {Name} має хвіст {Tail} см.");
    }
}

class Program
{
    static void Main(string[] args)
    {

        Cat a = new Cat { Name = "Локі", Tail = 25 };
        Cat b = a;
        b.Name = "Томас";
        b.Tail = 40;
        a.Print();
        b.Print();
        Console.ReadKey();
    }
}
```

І в результаті на екран буде виведено:



Чому ми отримали таку істотну відмінність, спробуйте пояснити самі.

Значення null

Одна з відмінностей посилальних типів від типів значень полягає в тому, що змінні посилальних типів можуть приймати значення null. Наприклад:

```
object o = null;
string s = null;
```

Якщо змінним посилального типу не присвоюється значення, то їм дається значення за замовчуванням – значення null. Фактично воно говорить про відсутність значення як такого.

Але типи значень, наприклад, int, decimal, double і т.д. не можуть приймати значення null.

Оператор ??

Оператор ?? називається оператором null-об'єднання. Він застосовується для установки значень за замовчуванням для типів, які допускають значення null. Оператор ?? повертає лівий операнд, якщо цей операнд НЕ дорівнює null. Інакше повертається правий операнд. При цьому лівий операнд повинен приймати null. Подивимося на прикладі:

```
object x = null;
object y = x ?? 100; // дорівнює 100, тому що x дорівнює null

object z = 200;
object t = z ?? 44; // дорівнює 200, тому що z НЕ дорівнює null
```

Але ми не можемо написати наступним чином:

```
int x = 44;
int y = x ?? 100;
```

Тут змінна x представляє тип значення int і не може приймати значення null, тому в якості лівого операнда в операції ?? вона використовуватися не може.

Оператор умовного null

Іноді при роботі з об'єктами, які приймають значення null, ми можемо зіткнутися з помилкою: ми намагаємося звернутися до об'єкта, а цей об'єкт дорівнює null. Наприклад, нехай у нас є наступна система класів:

```
class User
{
    public Phone Phone { get; set; }
}

class Phone
{
    public Company Company { get; set; }
}

class Company
{
    public string Name { get; set; }
}
```

Об'єкт User містить посилання на об'єкт Phone, а об'єкт Phone містить посилання на об'єкт Company, тому теоретично ми можемо отримати з об'єкта User назву компанії, наприклад, так:

```
User user = new User();
Console.WriteLine(user.Phone.Company.Name);
```

В даному випадку властивість Phone не визначена, буде за замовчуванням мати значення null. Тому ми зіткнемося з виключенням NullReferenceException. Щоб уникнути цієї помилки ми могли б використовувати умовну конструкцію для перевірки на null:

```
User user = new User();

if (user != null)
{
    if (user.Phone != null)
    {
        if (user.Phone.Company != null)
        {
            string companyName = user.Phone.Company.Name;
            Console.WriteLine(companyName);
        }
    }
}
```

Виходить багатоповерхова конструкція і щоб її спростити, в C# є оператор умовного null (Null-Conditional Operator):

```
string companyName = user?.Phone?.Company?.Name;
```

Вираз?. і представляє оператор умовного null. Тут послідовно перевіряється чи дорівнює об'єкт user і вкладені об'єкти значенням null. Якщо ж на якомусь етапі один з об'єктів виявиться рівним null, то змінна companyName матиме значення за замовчуванням, тобто null.

9.5 Модифікатори доступу

Всі члени класу – поля, методи, властивості – всі вони мають модифікатори доступу. Модифікатори доступу дозволяють задати допустиму область видимості для членів класу. Тобто контекст, в якому можна використовувати цю змінну або метод. Ми вже з ними стикалися, коли оголошували поля класу Person відкритими (тобто з модифікатором public).

У C# застосовуються такі модифікатори доступу:

public: відкритий, загальнодоступний клас або член класу. Такий член класу доступний з будь-якого місця в коді, а також з інших програм і збірок.

private: закритий клас або член класу. Представляє повну протилежність модифікатору public. Такий закритий клас або член класу доступний тільки з коду в тому ж класі або контексті.

protected: такий член класу доступний з будь-якого місця в поточному класі або в похідних класах. При цьому похідні класи можуть розташовуватися в інших збірках.

internal: клас і члени класу з подібним модифікатором доступні з будь-якого місця коду в тій же збірці, проте він недоступний для інших програм і збірок (як у випадку з модифікатором public).

protected internal: поєднує функціонал двох модифікаторів. Класи і члени класу з таким модифікатором доступні з поточної збірки і з похідних класів.

private protected: такий член класу доступний з будь-якого місця в поточному класі або в похідних класах, які визначені в тій же збірці.

Оголошення полів класу без модифікатора доступу рівнозначно їх оголошенню з модифікатором private. Класи, оголошені без модифікатора, за замовчуванням мають доступ internal.

Всі класи і структури, визначені безпосередньо в просторах імен і не є вкладеними в інші класи, можуть мати тільки модифікатори public або internal.

Створимо наступний клас State:

```
public class State
{
    int a; // все одно, що private int a;
    private int b; // поле доступно тільки з поточного класу
    protected int c; // доступно з поточного класу і похідних класів
    internal int d; // доступно в будь-якому місці програми
    protected internal int e; // доступно в будь-якому місці програми
                              //і з класів-спадкоємців
    public int f; // доступно в будь-якому місці програми, а також для
                  //інших програм і збірок
    protected private int g; // доступно з поточного класу і похідних
                             //класів, які визначені в тому ж проекті

    private void Display_f()
    {
        Console.WriteLine($"Змінна f = {f}");
    }

    public void Display_a()
    {
        Console.WriteLine($"Змінна a = {a}");
    }

    internal void Display_b()
    {
        Console.WriteLine($"Змінна b = {b}");
    }

    protected void Display_e()
    {
        Console.WriteLine($"Змінна e = {e}");
    }
}
```

Клас State оголошений з модифікатором public, буде доступний з будь-якого місця програми, а також з інших програм і збірок. Клас State має п'ять полів для кожного рівня доступу. Плюс одна змінна без модифікатора, яка є закритою за замовчуванням.

Також є чотири методи, які будуть виводити значення полів класу на екран. Зверніть увагу, що тому що всі модифікатори дозволяють використовувати члени класу всередині даного класу, то і всі змінні класу, в тому числі закриті, у нас доступні всім його методам, тому що всі знаходяться в контексті класу State.

Тепер подивимося, як ми зможемо використовувати змінні нашого класу в програмі (тобто в методі Main класу Program):

Завдяки такій системі модифікаторів доступу можна приховувати деякі моменти реалізації класу від інших частин програми. Таке приховування називається інкапсуляцією.

9.6 Властивості та інкапсуляція

Крім звичайних методів в мові C # передбачені спеціальні методи доступу, які називають властивостями. Властивості забезпечують простий доступ до полів класу, дозволяють дізнатися про їх значення або виконати їх установку.

Стандартне опис властивості має наступний синтаксис:
[Модіфікатор_доступа] тип_що_ повертається назва
{ // код властивості }

Наприклад:

```
class Person
{
    private string name;

    public string Name
    {
        get { return name;   }

        set { name = value;  }
    }
}
```

Тут у нас є закрите поле name і є загальнодоступна властивість Name. Хоча вони мають практично однакову назву за винятком регістра, але це не більше ніж стиль, назви у них можуть бути довільні і не обов'язково повинні збігатися.

За допомогою цієї властивості ми можемо управляти доступом до змінної name. Стандартне визначення властивості містить блоки **get** і **set**. У блоці get ми повертаємо значення поля, а в блоці set встановлюємо. Параметр value являє передане значення.

Ми можемо використовувати цю властивість наступним чином:

```
    static void Main(string[] args)
    {
        Person p = new Person();
        // Встановлюємо властивість - спрацьовує блок Set
        // значення "Вася" і є передане в властивість value
        p.Name = "Вася";

        // Отримуємо значення властивості і присвоюємо його змінної –
        //спрацьовує блок Get
        string personName = p.Name;

        Console.ReadKey();
    }
```

Можливо, може виникнути питання, навіщо потрібні властивості, якщо ми можемо в даній ситуації обходитися звичайними полями класу? Але властивості дозволяють вкласти додаткову логіку, яка може бути необхідна, наприклад, при присвоєнні змінної класу будь-якого значення.

Блоки set і get не обов'язково одночасно повинні бути присутніми в властивості. Якщо властивість визначають тільки блок get, то така властивість доступно тільки для читання - ми можемо отримати його значення, але не встановити. І, навпаки, якщо властивість має тільки блок set, тоді це властивість доступно тільки для запису – можна тільки встановити значення, але не можна отримати.

Ми можемо застосовувати модифікатори доступу не тільки до всієї властивості, але і до окремих блоків – або get, або set:

```
class Person
  {
      private string name;

      public string Name
      {
          get { return name; }
          private set { name = value; }
      }
      public Person(string name)
      {
          Name = name;
      }
  }
```

Тепер закритий блок set ми зможемо використовувати тільки в даному класі – в його методах, властивостях, конструкторі, але ніяк не в іншому класі.

При використанні модифікаторів у властивостях слід враховувати ряд обмежень:

– Модифікатор для блоку set або get можна встановити, якщо властивість має обидва блоки (і set, і get).

– Тільки один блок set або get може мати модифікатор доступу, але не обидва відразу.

– Модифікатор доступу блоку set або get повинен бути більш обмежуючим, аніж модифікатор доступу властивості. Наприклад, якщо властивість має модифікатор public, то блок set / get може мати тільки модифікатори protected internal, internal, protected, private.

Інкапсуляція

Через властивості встановлюється доступ до приватних змінних класу. Подібне приховування стану класу від втручання ззовні представляє механізм інкапсуляції, який представляє одне з базових принципів об'єктно-орієнтованого програмування. Застосування модифікаторів доступу типу private захищає змінну від зовнішнього доступу. Для управління доступом в багатьох мовах програмування використовуються спеціальні методи, геттери і сеттери. У C# їх роль, як правило, виконують властивості.

Наприклад, є деякий клас Account, в якому визначено поле sum, що представляє суму:

```
class Account
  {
      public int sum;
  }
```

Оскільки змінна sum є відкритою, то в будь-якому місці програми ми можемо отримати до неї доступ і змінити її, в тому числі встановити будь-яке неприпустиме значення, наприклад, негативне. Навряд чи подібна поведінка є бажаною. Тому застосовується інкапсуляція для обмеження доступу до змінної sum і приховування її всередині класу:

```
class Account
{
    private int sum;
    public int Sum
    {
        get { return sum; }
        set
        {
            if (value > 0)
            {
                sum = value;
            }
        }
    }
}
```

Автоматичні властивості

Властивості керують доступом до полів класу. Однак, якщо у нас з десяток і більше полів, то визначати кожне поле і писати для нього однотипну властивість було б втомливо. Тому у фреймворк .NET були додані автоматичні властивості. Вони мають скорочене оголошення:

```
class Person
{
    public string Name { get; set; }
    public int Age { get; set; }

    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }
}
```

Насправді тут також створюються поля для властивостей, тільки їх створює не програміст в коді, а компілятор автоматично генерує при компіляції.

У чому перевага автовластивостей, якщо по суті вони просто звертаються до автоматично створюваної змінної, чому б прямо не звернутися до змінної без автовластивостей? Справа в тому, що в будь-який момент часу при необхідності ми можемо розгорнути автовластивість у звичайну властивість, додати до неї якусь певну логіку.

Варто враховувати, що не можна створити автоматичну властивість тільки для запису, як у випадку зі стандартними властивостями.

Автовластивостям можна привласнити значення за замовчуванням (ініціалізація автовластивостей):

```
class Person
{
    public string Name { get; set; } = "Вася";
    public int Age { get; set; } = 23;
}

class Program
{
    static void Main(string[] args)
    {
        Person person = new Person();
        Console.WriteLine(person.Name); // Вася
        Console.WriteLine(person.Age);  // 23

        Console.ReadKey();
    }
}
```

І якщо ми не вкажемо для об'єкта Person значення властивостей Name і Age, то будуть діяти значення за замовчуванням.

```
class Person
    {
        public string Name { private set; get; }
        public Person(string n)
        {
            Name = n;
        }
    }
```

Ми можемо прибрати блок set і зробити автосвойство доступним тільки для читання. В цьому випадку для зберігання значення цієї властивості для нього неявно буде створюватися поле з модифікатором readonly, тому слід враховувати, що подібні get-властивості можна встановити або з конструктора класу, як в прикладі вище, або при ініціалізації властивості:

```
class Person
    {
        public string Name { get; } = "Вася";
    }
```

9.7 Індексатори

Індексатори дозволяють індексувати об'єкти і звертатися до даних за індексом. Фактично за допомогою індексаторів ми можемо працювати з об'єктами як з масивами. За формою вони нагадують властивості зі стандартними блоками get і set, які повертають і привласнюють значення.

Формальне визначення індекатора:

```
повертаємий_тип this[Тип параметр1, ...]
    {
        get { ... }
        set { ... }
    }
```

На відміну від властивостей індексатор не має назви. Замість його вказується ключове слово this, після якого в квадратних дужках йдуть параметри. Індексатор повинен мати як мінімум один параметр.

Подивимося на прикладі. Припустимо, у нас є клас Person, який представляє людину, і клас People, який представляє групу людей. Використовуємо індексатори для визначення класу People:

```
class Person
{
    public string Name { get; set; }
}
```

```
class People
{
    Person[] data;
    public People()
    {
        data = new Person[5];
    }
    // індексатор
    public Person this[int index]
    {
        get
        {
            return data[index];
        }
        set
        {
            data[index] = value;
        }
    }
}
```

Після цього ми можемо працювати з об'єктом People як з набором об'єктів Person:

```
class Program
{
    static void Main(string[] args)
    {
        People people = new People();
        people[0] = new Person { Name = "Вася" };
        people[1] = new Person { Name = "Томас" };

        Console.ReadKey();
    }
}
```

Індексатор отримує набір індексів у вигляді параметрів. Однак індекси необязаткельно повинні представляти тип int. Наприклад, ми можемо розглядати об'єкт як сховище властивостей і передавати ім'я атрибута об'єкта у вигляді рядка:

```
class User
{
    string name;
    string email;
    string phone;
    public string this[string propname]
    {
        get
        {
            switch (propname)
            {
```

```
                case "name": return "Пан/Пані " + name;
                case "email": return email;
                case "phone": return phone;
                default: return null;
            }
        }
        set
        {
            switch (propname)
            {
                case "name":
                    name = value;
                    break;
                case "email":
                    email = value;
                    break;
                case "phone":
                    phone = value;
                    break;
            }
        }
    }
}
class Program
{
    static void Main(string[] args)
    {
        User vasya = new User();
        vasya["name"] = "Вася";
        vasya["email"] = "vasisualiy@gmail.com";
        Console.WriteLine(vasya["name"]); // Пан/Пані Вася
        Console.ReadKey();
    }
}
```

```
H:\Методички\посібник ООП 3 видання\ConsoleApp2\
Пан/Пані Вася
```

Застосування декількох параметрів

Також індексатор може приймати кілька параметрів. Припустимо, у нас є клас, в якому сховище визначено у вигляді двомірного масиву або матриці:

```
class Matrix
{
    private int[,] numbers = new int[,]
    {   { 1, 2, 4 },
        { 2, 3, 6 },
        { 3, 4, 8 }
    };
```

```
    public int this[int i, int j]
    {
        get
        {
            return numbers[i, j];
        }
        set
        {
            numbers[i, j] = value;
        }
    }
}
```

Тепер для визначення індексатора використовуються два індекси - i та j. І в програмі ми вже повинні звертатися до об'єкта, використовуючи два індекса:

```
Matrix matrix = new Matrix();
Console.WriteLine(matrix[0, 0]);
matrix[0, 0] = 111;
Console.WriteLine(matrix[0, 0]);
```

9.8 Перевантаження операторів

Операція – це метод, який приймає параметри і повертає значення.
Наприклад, нехай у нас є наступний клас Counter:

```
class Counter
{
    public int Value { get; set; }
}
```

Цей клас представляє певний лічильник, значення якого зберігається у властивості Value.

І припустимо, у нас є два об'єкти класу Counter - два лічильника, які ми хочемо порівнювати або складати на підставі їх властивості Value, використовуючи стандартні операції порівняння і складання:

```
Counter c1 = new Counter { Value = 23 };
Counter c2 = new Counter { Value = 45 };
bool result = c1 > c2;
Counter c3 = c1 + c2;
```

Але на даний момент ні операція порівняння, ні операція складання для об'єктів Counter не доступні. Ці операції можуть використовуватися для ряду примітивних типів. Наприклад, за замовчуванням ми можемо складати числові значення, але як складати об'єкти комплексних типів – класів і структур компілятор не знає. І для цього нам треба виконати перевантаження потрібних нам операторів.

Перевантаження операторів полягає у визначенні в класі, для об'єктів якого ми хочемо визначити оператор, спеціального методу:

```
public static повертаємий_тип operator оператор (параметри)
{
}
```

Цей метод повинен мати модифікатори public static, тому що оператор, що перевантажується, буде використовуватися для всіх об'єктів даного класу. Далі йде назва типу, що повертається – це той тип, об'єкти якого ми хочемо отримати. Наприклад, в результаті складання двох об'єктів Counter ми очікуємо отримати новий об'єкт Counter. А в результаті порівняння двох ми хочемо отримати об'єкт типу bool, який вказує істинний умовний вираз або помилковий.

Потім замість назви методу йде ключове слово operator і власне сам оператор. І далі в дужках перераховуються параметри. Бінарні оператори приймають два параметра, унарні – один параметр. І в будь-якому випадку один з параметрів повинен становити той тип – клас або структуру, в якому визначається оператор.

Наприклад, перевантажимо ряд операторів для класу Counter:

```
class Counter
{
    public int Value { get; set; }

    public static Counter operator +(Counter c1, Counter c2)
    {
        return new Counter { Value = c1.Value + c2.Value };
    }
    public static bool operator >(Counter c1, Counter c2)
    {
        return c1.Value > c2.Value;
    }
    public static bool operator <(Counter c1, Counter c2)
    {
        return c1.Value < c2.Value;
    }
}
```

Оскільки все перевантажені оператори – бінарні, тобто проводяться над двома об'єктами, то для кожного перевантаження передбачено по два параметра.

Якщо ми хочемо скласти два об'єкти класу Counter, то оператор приймає два об'єкти цього класу. І тому що ми хочемо в результаті складання отримати новий об'єкт Counter, то даний клас також використовується в якості повертається типу. Всі дії цього оператора зводяться до створення, нового об'єкта, властивість Value якого об'єднує значення властивості Value обох параметрів.

Також перевантажені дві операції порівняння. Якщо ми перенавантажуємо одну з цих операцій порівняння, то ми також повинні перевантажити другу з цих

операцій. Самі оператори порівняння порівнюють значення властивостей Value і в залежності від результату порівняння повертають або true, або false.

Тепер використовуємо перевантажені оператори в програмі:

```
static void Main(string[] args)
{
    Counter c1 = new Counter { Value = 23 };
    Counter c2 = new Counter { Value = 45 };
    bool result = c1 > c2;
    Console.WriteLine(result); // false

    Counter c3 = c1 + c2;
    Console.WriteLine(c3.Value);  // 23 + 45 = 68

    Console.ReadKey();
}
```

Слід враховувати, що при перевантаженні не повинні змінюватися ті об'єкти, які передаються в оператор через параметри. Наприклад, ми можемо визначити для класу Counter оператор інкремента:

```
public static Counter operator ++(Counter c1)
{
    c1.Value += 1; // так не вірно!!!
    return c1;
}
```

Оскільки оператор унарний, він приймає тільки один параметр – об'єкт того класу, в якому даний оператор визначений. Але це **невірне** визначення інкремента, тому що оператор **не повинен міняти значення** своїх параметрів.

І більш коректне перевантаження оператора інкремента буде виглядати так:

```
public static Counter operator ++(Counter c1)
{
    return new Counter { Value = c1.Value + 1 };
}
```

Тобто повертається новий об'єкт, який містить у властивості Value інкрементоване значення.

При цьому нам не треба визначати окремо оператори для префіксного і для постфіксного інкремента (а також декремента), тому що одна реалізація буде працювати в обох випадках.

Наприклад, використовуємо операцію префіксного і посфіксного інкремента:

```
Counter counter = new Counter() { Value = 10 };
Console.WriteLine(counter.Value);      // 10
Console.WriteLine((++counter).Value);  // 11
Console.WriteLine(counter.Value);      // 11
Console.WriteLine((counter++).Value);  // 11
Console.WriteLine(counter.Value);      // 12
```

Також варто відзначити, що ми можемо перевизначити оператори true і false. Наприклад, визначимо їх в класі Counter:

```
public static bool operator true(Counter c1)
{
    return c1.Value != 0;
}
public static bool operator false(Counter c1)
{
    return c1.Value == 0;
}
```

Ці оператори перевантажуються, коли ми хочемо використовувати об'єкт типу в якості умови. Наприклад:

```
Counter counter = new Counter() { Value = 0 };
if (counter)
    Console.WriteLine(true);
else
    Console.WriteLine(false);
```

При перевантаженні операторів треба враховувати, що не всі оператори можна перевантажити. Зокрема, ми можемо перевантажити наступні оператори:

унарні оператори +, –, !, ~, ++, –
бінарні оператори +, –, *, /, %
операції порівняння ==, ! =, <, >, <=, > =
логічні оператори &&, ||

Не можна перевантажувати операції: =, ., ?:, ->, new, is, sizeof, typeof.

Оператори присвоювання +=, –=, *=, /=, %= не можуть бути перевантажені, але, наприклад, += оцінюється за допомогою +, який може бути перевантажений.

```
public static Counter operator +(Counter c1, Counter c2)
{
    return new Counter { Value = c1.Value + c2.Value };
}

    Counter counter1 = new Counter() { Value = 5 };
    Counter counter2 = new Counter() { Value = 6 };
    counter1 += counter2;
    Console.WriteLine(counter1.Value); // 11
```

Якщо ми перевизначаємо одну з операцій порівняння, то ми також повинні перевизначити другу з цих операцій. Самі оператори в залежності від результату порівняння повертають або true, або false.

Якщо в класі перевантажуються операції == і !=, необхідно перевизначити методи **GetHashCode** і **Equals**.

При перевантаженні операторів також слід пам'ятати, що ми не можемо змінити пріоритет оператора або його асоціативність, ми не можемо створити новий оператор або змінити логіку операторів в типах, який є за замовчуванням в .NET.

9.9 Статичні члени і модифікатор static

Крім звичайних полів, методів, властивостей клас може мати статичні поля, методи, властивості. Статичні поля, методи, властивості відносяться до всього класу і для звернення до подібних членам класу необов'язково створювати екземпляр класу. наприклад:

```
class Account
{
    public static decimal bonus = 100;
    public decimal totalSum;
    public Account(decimal sum)
    {
        totalSum = sum + bonus;
    }
}
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine(Account.bonus);      // 100
        Account.bonus += 200;

        Account account1 = new Account(150);
        Console.WriteLine(account1.totalSum);   // 450


        Account account2 = new Account(1000);
        Console.WriteLine(account2.totalSum);   // 1300

        Console.ReadKey();
    }
}
```

На рівні пам'яті для статичних полів буде створюватися ділянку в пам'яті, який буде спільним для всіх об'єктів класу.

В даному випадку клас Account має два поля: bonus і totalSum. Поле bonus є статичним, тому воно зберігає стан класу в цілому, а не окремого об'єкта. І тому ми можемо звертатися до цього поля по імені класу:

```
        Console.WriteLine(Account.bonus);
        Account.bonus += 200;
```

При цьому пам'ять для статичних змінних виділяється навіть в тому випадку, якщо не створено жодного об'єкта цього класу.

Статичні властивості і методи

Подібним чином ми можемо створювати і використовувати статичні методи і властивості:

```
class Account
{
    public Account(decimal sum, decimal rate)
    {
        if (sum < MinSum) throw new Exception("Неприпустима сума!");
        Sum = sum; Rate = rate;
    }
    private static decimal minSum = 100; // мінімальна допустима сума
    public static decimal MinSum
    {
        get { return minSum; }
        set { if (value > 0) minSum = value; }
    }

    public decimal Sum { get; private set; } // сума на рахунку
    public decimal Rate { get; private set; }   // процентна ставка

    // підрахунок суми на рахунку через певний період
    public static decimal GetSum(
        decimal sum,
        decimal rate,
        int period)
    {
        decimal result = sum;
        for (int i = 1; i <= period; i++)
            result = result + result * rate / 100;
        return result;
    }
}
```

Те ж саме з методом GetSum – він обчислює суму на рахунку через певний період за певною відсотковою ставкою для певної початкової суми. Виклик і результат цього методу не залежить від конкретного об'єкта або його стану.

Таким чином, змінні і властивості, які зберігають стан, загальні для всіх об'єктів класу, слід визначати як статичні. І також методи, які визначають загальну для всіх об'єктів поведінку, також слід оголошувати як статичні.

Статичні члени класу є загальними для всіх об'єктів цього класу, тому до них треба звертатися по імені класу:

```
Account.MinSum = 560;
decimal result = Account.GetSum(1000, 10, 5);
```

Статичний конструктор

Крім звичайних конструкторів у класу також можуть бути статичні конструктори. Статичні конструктори мають такі відмінні риси:

1. Статичні конструктори не повинні мати модифікатор доступу і не приймають параметрів
2. Як і в статичних методах, в статичних конструкторах не можна використовувати ключове слово this для посилання на поточний об'єкт класу і можна звертатися тільки до статичних членів класу
3. Статичні конструктори не можна викликати в програмі вручну. Вони виконуються автоматично при найпершому створенні об'єкта даного класу або при першому зверненні до його статичним членам (якщо такі є)
4. Статичні конструктори зазвичай використовуються для ініціалізації статичних даних, або ж виконують дії, які потрібно виконати тільки один раз.

Визначимо статичний конструктор:

```
class User
{
    static User()
    {
        Console.WriteLine("Створено перший користувач");
    }
}
class Program
{
    static void Main(string[] args)
    {
        User user1 = new User(); //працює статичний конструктор
        User user2 = new User();
        Console.ReadKey();
    }
}
```

Статичні класи

Статичні класи оголошуються з модифікатором static і можуть містити тільки статичні поля, властивості і методи. Наприклад, якби клас Account мав би тільки статичні змінні, властивості і методи, то його можна було б оголосити як статичний:

```
static class Account
{
    private static decimal minSum = 100; // мінімальна допустима сума
    public static decimal MinSum
    {
        get { return minSum; }
        set { if (value > 0) minSum = value; }
    }
```

```
        // підрахунок суми на рахунку через певний період
        public static decimal GetSum(
            decimal sum,
            decimal rate,
            int period)
        {
            decimal result = sum;
            for (int i = 1; i <= period; i++)
                result = result + result * rate / 100;
            return result;
        }
    }
```

У C# показовим прикладом статичного класу є клас Math, який застосовується для різних математичних операцій.

Резюме

1. Клас представляє новий тип, який визначається користувачем. Клас визначається за допомогою ключового слова class:
2. Описом об'єкта є клас, а об'єкт являє екземпляр цього класу.
3. Клас можна визначати всередині простору імен, поза простором імен, всередині іншого класу.
4. Конструктор ініціалізує об'єкт при його створенні. Конструктор – це метод класу. У конструктора таке ж ім'я, як і у його класу. Конструктори ніколи не повертають значення, тому для них не вказується тип значення, що повертається.
5. Якщо в класі не визначено жодного конструктора, то для цього класу автоматично створюється конструктор за замовчуванням.
6. Через властивості встановлюється доступ до приватних змінних класу. Подібне приховування стану класу від втручання ззовні представляє механізм інкапсуляції.
7. Індексатори дозволяють індексувати об'єкти і звертатися до даних за індексом.
8. Перевантаження операторів полягає у визначенні в класі, для об'єктів якого ми хочемо визначити оператор, спеціального методу, який повинен мати модифікатори public static. Далі йде назва типу, що повертається.
9. Якщо ми перевизначаємо одну з операцій порівняння, то ми також повинні перевизначити другу з цих операцій. Самі оператори в залежності від результату порівняння повертають або true, або false.
10. Якщо в класі перевантажуються операції == і !=, необхідно перевизначити методи GetHashCode і Equals.
11. Статичні поля, методи, властивості відносяться до всього класу і для звернення до подібних членам класу необов'язково створювати екземпляр класу.

12. Крім звичайних конструкторів у класу також можуть бути статичні конструктори.

Завдання до виконання

1. Створити клас з прихованими полями, конструкторами з параметрами та без параметрів, методами, властивостями і перевантаженими операціями відповідно варіанту.
2. Розроблена програма повинна містити перевірку усіх розроблених елементів класу.

Варіанти завдань:

Варіант 1. Створити клас «парне число», такий, що:

a) його екземпляр може містити тільки позитивне парне число в діапазоні 2..200.

b) його конструктор без параметра створює екземпляр зі значенням 2, а конструктор з параметром при створенні екземпляра округлює значення параметра до найближчого парного числа в зазначеному діапазоні.

c) його методи дозволяють отримувати і присвоювати значення, виконуючи при цьому відповідні перетворення.

d) його функція print () виводить на екран значення екземпляра, укладену в подвійні лапки.

e) його операція «+» перевантажена: для двох парних чисел обчислює їх полусумму і повертає парне число; при операції зі звичайним числом обчислює суму і повертає ціле число.

Скласти програму, яка перевіряє виконання перерахованих функцій.

Варіант 2. Створити клас «непарне число», такий, що:

a) його екземпляр може містити тільки позитивне непарне число в діапазоні 1..201.

b) його конструктор без параметра створює екземпляр зі значенням 1, а конструктор з параметром при створенні екземпляра округлює значення параметра до найближчого непарного числа в зазначеному діапазоні.

c) його методи дозволяють отримувати і присвоювати значення, виконуючи при цьому відповідні перетворення.

d) функція print () виводить на екран значення екземпляра, укладену в одинарні лапки.

Операція «–» перевантажена: для двох непарних чисел обчислює їх різницю і повертає непарне число, при операції зі звичайним числом обчислює різницю і повертає дійсне число.

Скласти програму, яка перевіряє виконання перерахованих функцій.

Варіант 3. Створити клас «бухгалтерське число», такий, що:

a) його екземпляр може містити грошове вираження в діапазоні від 0.00грн до +1 000 000.00грн.

b) його конструктор без параметра створює екземпляр зі значенням 0, а конструктор з параметром при створенні екземпляра округлює значення параметра до копійок в зазначеному діапазоні. При перевищенні діапазону заповниться 0.00, та встановлює прапорець «переповнення»

c) його методи дозволяють отримувати і присвоювати значення, виконуючи при цьому відповідні перетворення.

d) функція print () виводить на екран значення екземпляра з пробілами перед кожною тріадою цифр і круглими дужками навколо від'ємного значення, замість знаку «-».

e) операція «+» перевантажена для бухгалтерських і звичайних чисел, обчислює суму, виводячи попередження в разі виходу за діапазон.
Скласти програму, яка перевіряє виконання перерахованих функцій.

Варіант 4. Створити базовий клас «комплексне число», такий, що:

a) його екземпляр містить дійсну і уявну частину ¬- змінні з плаваючою точкою.

b) його конструктор без параметра створює екземпляр зі значенням 0.0 + i0.0, а конструктор з параметрами створює екземпляр з відповідною дійсної та уявної частиною.

c) його методи дозволяють отримувати і присвоювати значення, виконуючи при цьому відповідні перетворення.

d) функція print () виводить на екран значення екземпляра у вигляді (0.00 ± i0.00).

e) операція «+» перевантажена: для двох комплексних чисел обчислює суму, складаючи окремо дійсні частини і окремо - уявні частини; при додаванні з дійсним числом збільшується тільки дійсна частина.
Скласти програму, яка перевіряє виконання перерахованих функцій.

Варіант 5. Створити клас «вектор», такий, що:

a) його екземпляр містить три координати - змінні з плаваючою точкою.

b) його конструктор без параметра створює екземпляр зі значенням 0.0 0.0 0.0, а конструктор з параметрами створює екземпляр з відповідними значеннями координат.

c) його методи дозволяють отримувати і присвоювати значення, виконуючи при цьому відповідні перетворення.

d) функція print () виводить на екран значення екземпляра в <0.0 0.0 0.0>

e) операція «+» перевантажена: для векторів обчислює суму, складаючи попарно значення кожної координати, а при додаванні зі скаляром збільшує кожну координату на величину скаляра.
Скласти програму, яка перевіряє виконання перерахованих функцій.

Варіант 6. Створити клас «точка», такий, що:

a) його екземпляр містить дві координати - позитивні цілі числа.

b) його конструктор без параметра створює екземпляр зі значенням 0 0, а конструктор з параметрами створює екземпляр з відповідними значеннями координат.

c) його методи дозволяють отримувати і присвоювати значення, виконуючи при цьому відповідні перетворення.

d) функція print () виводить на екран значення координат в форматі <0: 0>.

e) його операція «+» перевантажена: для двох точок створює нову точку, розташовану на середині відрізка, що з'єднує ці дві точки; при додаванні з числом змінює значення обох своїх координат на це число і повертає свою адресу.

Скласти програму, яка перевіряє виконання перерахованих функцій.

Варіант 7. Створити базовий клас «квадрат», такий, що:

a) його екземпляр містить розмір сторони квадрата.

b) його конструктор без параметра створює екземпляр зі значенням 0, а конструктор з параметрами створює екземпляр з відповідним значенням боку.

c) його методи дозволяють отримувати і привласнювати значення сторони і площі, виконуючи при цьому відповідні перетворення.

d) функція print_sqr () виводить на екран значення його площі.

e) операція «+» перевантажена: для двох квадратів створює новий квадрат сумарної площі, а при додаванні з числом змінює свою площу на цю величину і повертає нове значення площі.Скласти програму, яка перевіряє виконання перерахованих функцій.

Тести для закріплення матеріалу

1. Є наступний клас:

```
class Person
{
    public string name = "Леопольд";
    public int age;

    public Person(string name, int age)
    {
        this.name = name;
        this.age = age;
    }
}
```

Яке значення поле name буде мати при виконання наступного коду:

```
Person person = new Person("Вася", 34) { name = "Томас", age = 7 };
```

а) "Вася";
б) "Томас";
в) "Леопольд";
г) помилка компіляції.

2. Вам треба визначити в класі змінну, яка повинна бути доступна з будь-якого місця в поточному проекті. Який модифікатор (або модифікатори, якщо їх декілька) ви будете використовувати?
   а) public;
   б) protected;
   в) static;
   г) internal.

3. Що буде виведено на консоль в результаті виконання наступної програми?

```
class Person
{
    private int age = 15;

    public int Age
    {
        get { return age; }
        set { }
    }
}
class Program
{
    static void Main(string[] args)
    {
        Person tom = new Person();
        tom.Age = 3;
        Console.WriteLine(tom.Age);

        Console.ReadKey();
    }
}
```

   а) 15
   б) 3
   в) 0
   г) Програма не скомпілюється

4. У класі Counter визначено оператор складання, який дозволяє складати об'єкт Counter з об'єктом типу int наступним чином:

```
Counter counter = new Counter { Number = 45 };
int x = counter + 6;
Console.WriteLine(x);  // 51
```

Який саме оператор складання визначено в класі Counter?

а)
```
public int operator +(Counter counter, int val)
 {
     return counter.Number + val;
 }
```
б)
```
public static int operator +(Counter counter, int val)
 {
     return counter.Number + val;
 }
```
в)
```
public static int operator +(int val, Counter counter)
 {
     return counter.Number + val;
 }
```
г)
```
public static Counter operator +(Counter counter, int val)
 {
     return new Counter { Number = counter.Number + val };
 }
```

РОЗДІЛ 10. СПАДКУВАННЯ І ПОЛІМОРФІЗМ

10.1 Спадкування

Спадкування (inheritance) є одним із принципів ООП. Базовий клас задає загальні ознаки і загальну поведінку для класів-спадкоємців.

Загальні властивості і методи унаслідуються від базового класу. До них додаються і визначаються **нові** властивості і методи.

Таким чином, перш за все спадкування реалізує механізми розширення базового класу.

Нехай у нас є наступний клас Person, який описує окрему людину:

```
class Person
{
    private string _name;

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }
    public void Display()
    {
        Console.WriteLine(Name);
    }
}
```

Але раптом нам знадобився клас, що описує співробітника підприємства – клас Employee. Оскільки цей клас буде реалізовувати той же функціонал, що і клас Person, тому що співробітник – це також і людина, то було б раціонально зробити клас Employee похідним (або спадкоємцем, або підкласом) від класу Person, який, в свою чергу, називається базовим класом або батьком (або суперкласом):

```
class Employee : Person
{

}
```

Після двокрапки ми вказуємо базовий клас для даного класу. Для класу Employee базовим є Person, і тому клас Employee успадковує всі ті ж властивості, методи, поля, які є в класі Person. Інша справа, що скористатися в похідному класі можна не всім спадком. Об'єкт-представник похідного класу **не** може отримати доступ до **private** членів базового класу. Використовуємо ще один модифікатор доступу – **protected**. Цей модифікатор забезпечує відкритий доступ до членів базового класу, але тільки для похідного класу.

В C# структури не підтримують спадкування. Тому модифікатор доступу protected в оголошенні членів структур **не застосовується**.

Конструктор базового класу **не успадковується**!

Спадкування реалізує ставлення is-a (є), об'єкт класу Employee також є об'єктом класу Person:

```
static void Main(string[] args)
{
    Person p = new Person { Name = "Вася" };
    p.Display();
    p = new Employee { Name = "Томас" };
    p.Display();
    Console.ReadKey();
}
```

І оскільки об'єкт Employee є також і об'єктом Person, то ми можемо так визначити змінну:

```
Person p = new Employee();
```

За замовчуванням всі класи успадковуються від базового класу Object, навіть якщо ми явно не встановлюємо успадкування. Тому вище певні класи Person і Employee крім своїх власних методів, також матимуть і методи класу Object: ToString(), Equals(), GetHashCode() і GetType().

Всі класи за замовчуванням можуть успадковуватися. Однак тут є ряд обмежень:

В C# не підтримується множинне спадкування, клас може успадковуватися тільки від одного класу.

При створенні похідного класу треба враховувати тип доступу до базового класу – тип доступу до похідному класу повинен бути таким же, як і у базового класу, або більш суворим. Тобто, якщо базовий клас у нас має тип доступу internal, то похідний клас може мати тип доступу internal або private, але не public.

Якщо клас оголошений з модифікатором **sealed**, то від цього класу не можна створювати похідні класи. Наприклад, наступний клас Cat не допускає створення спадкоємців, у котів це буває після певної операці…

```
sealed class Cat
{
}
```

Не можна успадкувати клас від **статичного** класу.

Ключове слово base

Тепер додамо в наші класи конструктори:

```
class Person
{
    public string Name { get; set; }

    public Person(string name)
    {
        Name = name;
    }

    public void Display()
    {
        Console.WriteLine(Name);
    }
}

class Employee : Person
{
    public string Company { get; set; }

    public Employee(string name, string company)
        : base(name)
    {
        Company = company;
    }
}
```

Клас Person має конструктор, який встановлює властивість Name. Оскільки клас Employee успадковує і встановлює ту ж властивість Name, то логічно було б не писати по сто разів код установки, а якось викликати відповідний код класу Person. До того ж властивостей, які треба встановити в конструкторі базового класу, і параметрів може бути набагато більше.

За допомогою ключового слова base ми можемо звернутися до базового класу. У нашому випадку в конструкторі класу Employee нам треба встановити ім'я і компанію. Але ім'я ми передаємо на установку в конструктор базового класу, тобто в конструктор класу Person, за допомогою виразу `base(name)`.

```
static void Main(string[] args)
{
    Person p = new Person("Вася");
    p.Display();
    Employee emp = new Employee("Томас", "Microsoft");
    emp.Display();
    Console.ReadKey();
}
```

Конструктори в похідних класах

Конструктори не передаються похідному класу при наслідуванні. І якщо в базовому класі не визначений конструктор за замовчуванням без параметрів, а тільки конструктори з параметрами (як у випадку з базовим класом Person), то в

похідному класі ми обов'язково повинні викликати один з цих конструкторів через ключове слово base. Наприклад, з класу Employee приберемо визначення конструктора:

```
class Employee : Person
{
    public string Company { get; set; }
}
```

В даному випадку ми отримаємо помилку, через те, що клас Employee не відповідає класу Person, а саме не викликає конструктор базового класу.

```
Person p = new Person("Вася");
p.Display();
Employee emp = new Employee("Томас", "Microsoft");
emp.Display();
```

Навіть якби ми додали якийсь конструктор, який би встановлював все ті ж властивості, то ми все одно б отримали помилку:

```
class Employee : Person
{
    public string Company { get; set; }
    public Employee(string name, string company)
    {
        Name = name;
        Company = company;
    }
}
```

Тобто в класі Employee через ключове слово base треба явним чином викликати конструктор класу Person:

```
public Employee(string name, string company)
    : base(name)
{
    Company = company;
}
```

Або в якості альтернативи ми могли б визначити в базовому класі конструктор без параметрів:

```
class Person
{
    // решта коду класу
    // конструктор за замовчуванням
    public Person()
    {
        Name = "Вася";
        Console.WriteLine("Виклик конструктора без параметрів");
    }
}
```

Тоді в будь-якому конструкторі похідного класу, де немає звернення до конструктора базового класу, все одно неявно викликався б цей конструктор за замовчуванням. Наприклад, конструктор

```
public Employee(string company)
{
    Company = company;
}
```

фактично був би еквівалентний наступному конструктору:

```
public Employee(string company)
    : base()
{
    Company = company;
}
```

Порядок виклику конструкторів

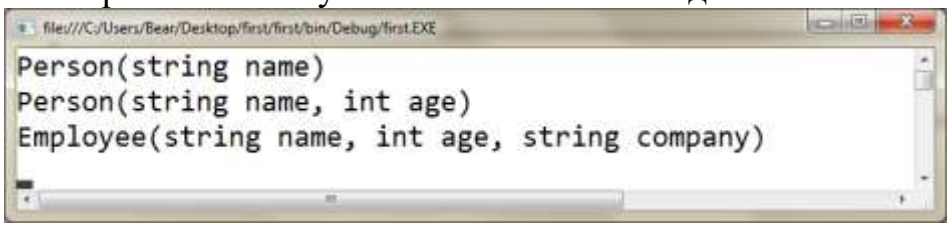
При виклику конструктора класу спочатку відпрацьовують конструктори базових класів і тільки потім конструктори похідних. Наприклад, візьмемо такі класи:

```
class Person
{
    string name;
    int age;

    public Person(string name)
    {
        this.name = name;
        Console.WriteLine("Person(string name)");
    }
    public Person(string name, int age)
        : this(name)
    {
        this.age = age;
        Console.WriteLine("Person(string name, int age)");
    }
}
class Employee : Person
{
    string company;

    public Employee(string name, int age, string company)
        : base(name, age)
    {
        this.company = company;
 Console.WriteLine("Employee(string name, int age, string company)");
    }
}
```

При створенні об'єкта Employee:

```
Employee vasya = new Employee("Вася", 22, "Microsoft");
```

Ми отримаємо наступний консольний вивід:



У підсумку ми отримуємо наступний ланцюжок виконань:

1. Спочатку викликається конструктор Employee(string name, int age, string company). Він делегує виконання конструктору Person(string name, int age).
2. Викликається конструктор Person(string name, int age), який сам поки не виконується і передає виконання конструктору Person (string name).
3. Викликається конструктор Person(string name), який передає виконання конструктору класу System.Object, бо це базовий за замовчуванням клас для Person.
4. Виконується конструктор System.Object.Object(), потім виконання повертається конструктору Person(string name).
5. Виконується тіло конструктора Person(string name), потім виконання повертається конструктору Person(string name, int age).
6. Виконується тіло конструктора Person(string name, int age), потім виконання повертається конструктору Employee(string name, int age, string company).
7. Виконується тіло конструктора Employee (string name, int age, string company). В результаті створюється об'єкт Employee.

10.2 Перетворення типів

В розділі 8.3 ми говорили про перетворення об'єктів простих типів. Тепер розглянемо перетворення об'єктів класів. Припустимо, у нас є наступна ієрархія класів:

```
class Person
{
    public string Name { get; set; }
    public Person(string name)
    {
        Name = name;
    }
    public void Display()
    {
        Console.WriteLine("Person " + Name);
    }
}
```

```
class Employee : Person
{
    public string Company { get; set; }
    public Employee(string name, string company) : base(name)
    {
        Company = company;
    }
}

class Client : Person
{
    public string Bank { get; set; }
    public Client(string name, string bank) : base(name)
    {
        Bank = bank;
    }
}
```

У цій ієрархії класів ми можемо простежити наступний ланцюг спадкування: Object (всі класи неявно успадковуються від типу Object) -> Person -> Employee | Client. (Рис. 10.1)

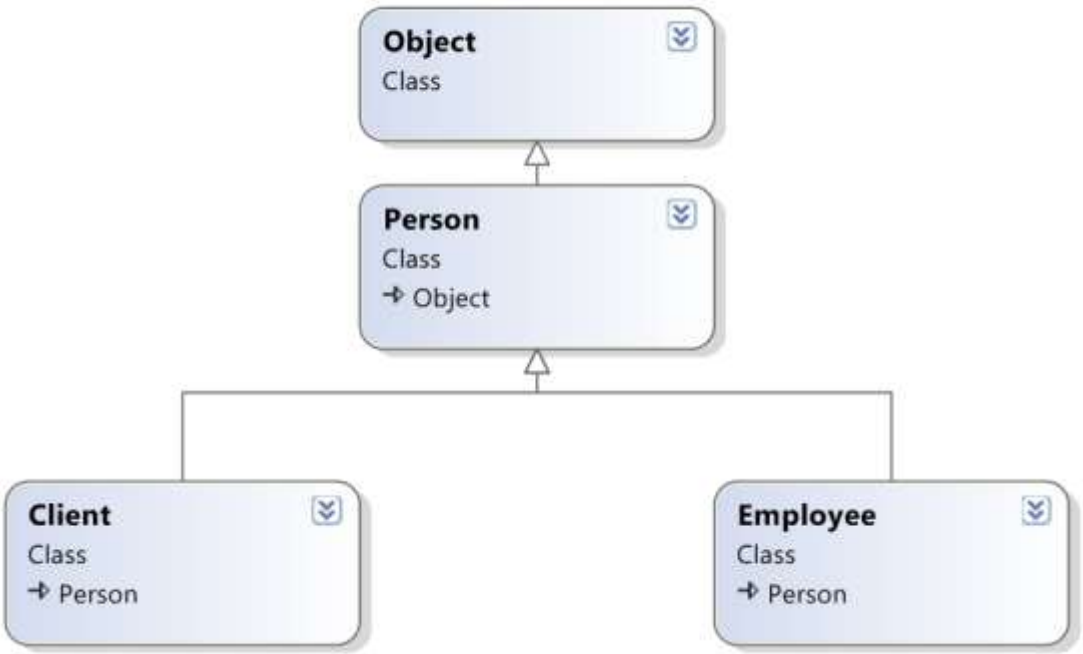


Рис. 10.1. Ієрархія класів.

У цій ієрархії класів базові типи знаходяться вгорі, а похідні типи – внизу.

Висхідні перетворення. Upcasting

Об'єкти похідного типу (який знаходиться внизу ієрархії) в той же час представляють і базовий тип. Наприклад, об'єкт Employee в той же час є і об'єктом класу Person. Що в принципі природно, тому що кожен співробітник

(Employee) є людиною (Person). І ми можемо написати, наприклад, наступним чином:

```
class Program
{
   static void Main(string[] args)
   {
      Employee employee = new Employee("Вася", "Microsoft");
      Person person = employee; // перетворення від Employee до Person

      Console.WriteLine(person.Name);
      Console.ReadKey();
   }
}
```

В даному випадку змінної person, яка представляє тип Person, присвоюється посилання на об'єкт Employee. Але щоб зберегти посилання на об'єкт одного класу в змінну іншого класу, необхідно виконати перетворення типів – в даному випадку від типу Employee до типу Person. І тому що Employee успадковується від класу Person, то автоматично виконується неявне висхідне перетворення – перетворення до типу, що знаходиться вгорі ієрархії класів, тобто до базового класу.

В результаті змінні employee і person вказуватимуть на один і той же об'єкт в пам'яті, але змінної person буде доступна тільки та частина, яка представляє функціонал типу Person. (Рис. 10.2)

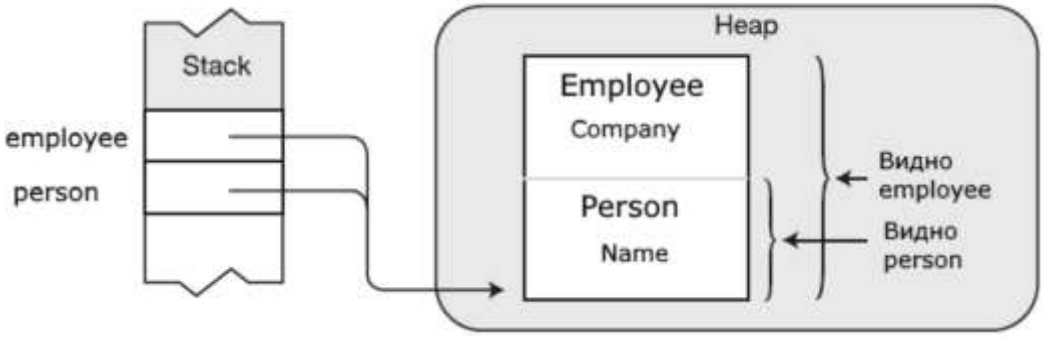


Рис. 10.2. Неявне висхідне перетворення.

Низхідні перетворення. Downcasting

Але крім висхідних перетворень від похідного до базового типу є низхідні перетворення або downcasting – від базового типу до похідного.

Може виникнути питання, чи можна звернутися до функціонала типу Employee через змінну типу Person. Але автоматично такі перетворення не проходять, адже не кожна людина (об'єкт Person) є співробітником підприємства (об'єктом Employee). І для низхідного перетворення необхідно застосувати явне перетворення, вказавши в дужках тип, до якого потрібно виконати перетворення:

```
Employee employee = new Employee("Вася", "Microsoft");
Person person = employee; // перетворення від Employee до Person
```

```
// Employee employee2 = person; // так можна, потрібно явне перетворення
Employee employee2 = (Employee)person; // перетворення від Person до Employee
```

Розглянемо деякі приклади перетворень:

```
// Об'єкт Employee також представляє тип object
object obj = new Employee("Bill", "Microsoft");
//щоб звернутися до можливостей типу Employee,
//приводимо об'єкт до типу Employee
Employee emp = (Employee)obj;

// об'єкт Client також представляє тип Person
Person person = new Client("Sam", "ContosoBank");
// перетворення від типу Person до Client
Client client = (Client)person;
```

У першому випадку змінної obj привласнено посилання на об'єкт Employee, тому ми можемо перетворити об'єкт obj до будь-якого типу який розташовується в ієрархії класів між типом object і Employee.

Якщо нам треба звернутися до якихось окремих властивостей або методів об'єкта, то нам необов'язково привласнювати перетворений об'єкт змінної:

```
// Об'єкт Employee також представляє тип object
object obj = new Employee("Bill", "Microsoft");
// перетворення до типу Person для виклику методу Display
((Person)obj).Display();
// або так
// ((Employee)obj).Display();

// перетворення до типу Employee, щоб отримати властивість Company
        string comp = ((Employee)obj).Company;
```

У той же час необхідно бути обережними при подібних перетвореннях. Наприклад, що буде в наступному випадку:

```
// Об'єкт Employee також представляє тип object
object obj = new Employee("Bill", "Microsoft");

// перетворення до типу Client, щоб отримати властивість Bank
string bank = ((Client)obj).Bank;
```

В даному випадку ми отримаємо помилку, тому що змінна obj зберігає посилання на об'єкт Employee. Даний об'єкт є також об'єктом типів object і Person, тому ми можемо перетворити його до цих типів. Але до типу Client ми перетворити не можемо.

Інший приклад:

```
Employee emp = new Person("Вася");   // ! Помилка

Person person = new Person("Томас");
Employee emp2 = (Employee)person;  // ! Помилка
```

В даному випадку ми намагаємося перетворити об'єкт типу Person до типу Employee, а об'єкт Person не є об'єктом Employee.

Існує ряд способів, щоб уникнути подібних помилок перетворення.

Способи перетворень

По-перше, можна використовувати ключове слово as. За допомогою нього програма намагається перетворити вираз до певного типу, при цьому не викидає виключення. У разі невдалого перетворення вираз буде містити значення null:

```
Person person = new Person("Вася");
Employee emp = person as Employee;
if (emp == null)
{
    Console.WriteLine("Перетворення пройшло невдало");
}


else
{
    Console.WriteLine(emp.Company);
}
```

Другий спосіб полягає в відловлюванні виключення InvalidCastException, яке виникне в результаті перетворення:

```
Person person = new Person("Вася");
try
{
    Employee emp = (Employee)person;
    Console.WriteLine(emp.Company);
}
catch (InvalidCastException ex)
{
    Console.WriteLine(ex.Message);
}
```

Третій спосіб полягає в перевірці допустимості перетворення за допомогою ключового слова is:

```
Person person = new Person("Вася");
if (person is Employee)
{
```

```
        Employee emp = (Employee)person;
        Console.WriteLine(emp.Company);
    }
    else
    {
        Console.WriteLine("Перетворення не припустимо");
    }
```

Вираз person is Employee перевіряє, чи є змінна person об'єктом типу Employee. Але тому що в даному випадку явно не є, то така перевірка поверне значення false, і перетворення не спрацює.

Перевантаження операцій перетворення типів

Раніше ми розглядали явні і неявні перетворення примітивних типів. наприклад:

```
int x = 50;
byte y = (byte)x; // явне перетворення від int до byte
int z = y; // неявне перетворення від byte до int
```

Якщо для елементарних типів проблема перетворення вирішується, то для класів, що знов оголошуються, алгоритми перетворення повинні реалізовуватися розробниками цих класів.

Логіка побудови явних і неявних перетворювачів досить проста. Програміст самостійно приймає рішення щодо того:

– яким повинен бути алгоритм перетворення;

– чи буде цей алгоритм виконуватися неявно або необхідно буде явно вказувати на відповідне перетворення.

Визначати логіку перетворення одних типів в інші можна за допомогою перевантаження операторів.

Для цього в класі визначається метод такої форми:

```
public static implicit|explicit operator Тип_в_який_треба_перетворити
(тип param)
     {
       // логіка перетворення
     }
```

Після модифікаторів public static йде ключове слово **explicit** (якщо перетворення явне, тобто потрібна операція приведення типів) або **implicit** (якщо перетворення неявне). Потім йде ключове слово **operator** і далі повертається тип, в який треба перетворити об'єкт. У дужках як параметр передається об'єкт, який треба перетворити.

Нижче розглядається приклад, що містить оголошення класів Point2D і Point3D (двовимірна і тривимірна точка). У класах передбачені алгоритми перетворення значень від одного типу до іншого, які активізуються при виконанні операцій присвоєння:

```
// Оголошення класів.
 class Point3D
 {
     public int x, y, z;
     public Point3D()
     {
         x = 0;
         y = 0;
         z = 0;
     }
     public Point3D(int xKey, int yKey, int zKey)
     {
         x = xKey;
         y = yKey;
         z = zKey;
     }

     // Операторная функція, в якій реалізується алгоритм перетворення
     // значення типу Point2D в значення типу Point3D. це перетворення
     // здійснюється неявно.

     public static implicit operator Point3D(Point2D p2d)
     {
         Point3D p3d = new Point3D();
         p3d.x = p2d.x;
         p3d.y = p2d.y;
         p3d.z = 0;
         return p3d;
     }
 }


 class Point2D
 {
     public int x, y;

     public Point2D()
     {
         x = 0;
         y = 0;
     }

     public Point2D(int xKey, int yKey)
     {
         x = xKey;
         y = yKey;
     }

// Операторная функція, в якій реалізується алгоритм перетворення
// значення типу Point3D в значення типу Point2D. це перетворення
// здійснюється явно.
```

```
        public static explicit operator Point2D(Point3D p3d)
        {
            Point2D p2d = new Point2D();
            p2d.x = p3d.x;
            p2d.y = p3d.y;
            return p2d;
        }
    }


// Клас Program. Тут все відбувається.
    class Program
    {
        static void Main(string[] args)
        {
            Point2D p2d = new Point2D(125, 125);
            Point3D p3d; // Зараз це тільки посилання!
            // Цією посиланням присвоюється значення в результаті
            // неявні перетворення значення типу Point2D до типу Point3D
            p3d = p2d;

            // Змінили значення полів об'єкта.
            p3d.x = p3d.x * 2;
            p3d.y = p3d.y * 2;
            p3d.z = 125; // Головне - з'явилася нова інформація,
            // яка буде втрачена в разі присвоєння значення типу Point3D
            // значенням типу Point2D.
            p2d = (Point2D)p3d;
        }
    }
```

10.3 Віртуальні методи і властивості

При наслідуванні нерідко виникає необхідність змінити в класі-спадкоємці функціонал методу, який був успадкований від базового класу. В цьому випадку клас-спадкоємець може перевизначати методи і властивості базового класу.

Ті методи і властивості, які ми хочемо зробити доступними для перевизначення, в базовому класі позначається модифікатором **virtual**. Такі методи і властивості називають віртуальними.

А щоб перевизначити метод в класі-спадкоємці, цей метод визначається з модифікатором **override**. Перевизначений метод в класі-спадкоємці повинен мати той же набір параметрів, що і віртуальний метод в базовому класі.

Наприклад, розглянемо наступні класи:

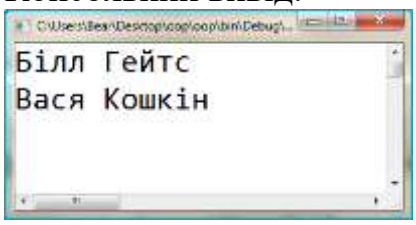
```
class Person
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public Person(string firstName, string lastName)
        {
            FirstName = firstName;
            LastName = lastName;
        }
        public virtual void Display()
        {
            Console.WriteLine($"{FirstName} {LastName}");
        }
    }

    class Employee : Person
    {
        public string Company { get; set; }
        public Employee(string firstName, string lastName, string company)
            : base(firstName, lastName)
        {
            Company = company;
        }
    }
```

Тут клас Person представляє людину. Клас Employee успадковується від Person і представляє співробітника підприємства. Цей клас крім успадкованих властивостей має ще одну властивість – Company.

Щоб зробити метод Display доступним для перевизначення, цей метод визначений з модифікатором virtual. Тому ми можемо перевизначити цей метод, але можемо і не перевизначати. Припустимо, нас влаштовує реалізація методу з базового класу. В цьому випадку об'єкти Employee використовуватимуть реалізацію методу Display з класу Person:

```
      static void Main(string[] args)

        {
            Person p1 = new Person("Білл", "Гейтс");
            p1.Display(); // виклик методу Display з класу Person

            Employee p2 = new Employee("Вася", "Кошкін", "Microsoft");
            p2.Display(); // виклик методу Display з класу Person

            Console.ReadKey();
        }
```

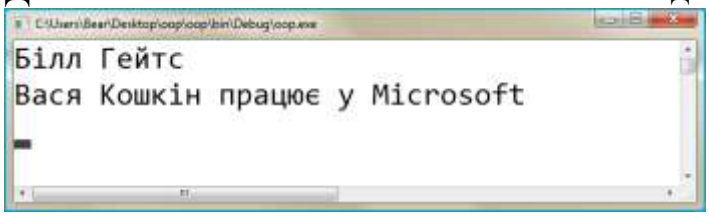
Консольний вивід:



Але також можемо перевизначити віртуальний метод. Для цього в класі-спадкоємці визначається метод з модифікатором **override**, який має таке ж ім'я і набір параметрів:

```
class Employee : Person
{
    public string Company { get; set; }
    public Employee(string firstName, string lastName, string company)
        : base(firstName, lastName)
    {
        Company = company;
    }

    public override void Display()
    {
        Console.WriteLine
        ($"{FirstName} {LastName} працює у {Company}");
    }
}
```

Для тих же самих об'єктів косольний вивід такий:



Віртуальність успадковується. Віртуальні методи базового класу визначають інтерфейс всієї ієрархії, тобто в будь-якому похідному класі, який не є прямим спадкоємцем від базового класу, можна перевизначити віртуальні методи. Наприклад, ми можемо визначити клас Manager, який буде похідним від Employee, і в ньому також перевизначити віртуальний метод Display класу Employee.

При перевизначенні віртуальних методів слід враховувати ряд обмежень:

Віртуальний і перевизначений методи повинні мати один і той же модифікатор доступу. Тобто є якщо віртуальний метод визначений за допомогою модифікатора public, то і перевизначений метод також повинен мати модифікатор public.

Не можна перевизначити або оголосити віртуальним статичний метод.

Перевизначення властивостей

Так саме, як методи, можна перевизначати властивості:

```
class Credit
{
    public virtual decimal Sum { get; set; }
}
class LongCredit : Credit
{
    private decimal sum;
    public override decimal Sum
    {
        get
        {
            return sum;
        }
        set
        {
            if (value > 1000)
            {
                sum = value;
            }
        }
    }
}
```

Ключове слово base

Крім конструкторів, ми можемо звернутися за допомогою ключового слова base до інших членів базового класу. У нашому випадку виклик base.Display (); буде зверненням до методу Display () в класі Person:

```
class Employee : Person
{
    public string Company { get; set; }

    public Employee(string lastName, string firstName, string company)
            : base(firstName, lastName)
    {
        Company = company;
    }

    public override void Display()
    {
        base.Display();
        Console.WriteLine($"працює у {Company}");
    }
}
```

Заборона перевизначення методів

Також можна заборонити перевизначення методів і властивостей. В цьому випадку їх треба оголошувати з модифікатором **sealed**:

```
class Employee : Person
{
    public string Company { get; set; }

    public Employee(string lastName, string firstName, string company)
            : base(firstName, lastName)
    {
        Company = company;
    }
    public override sealed void Display()
    {
        base.Display();
        Console.WriteLine($"працює у {Company}");
    }
}
```

При створенні методів з модифікатором sealed треба враховувати, що sealed застосовується в парі з override, тобто тільки в переовизначаємих методах.

І в цьому випадку ми не зможемо перевизначити метод Display в класі, успадкованому від Employee.

Приховування методів

Іншим способом змінити функціональність методу, успадкованого від базового класу, є приховування (shadowing / hiding).

Фактично приховуванням є визначення в класі-спадкоємці методу або властивості, які мають таке саме ім'я, повертаємий тип і набір параметрів, як відповідний метод або властивість базового класу. Для приховування членів класу застосовується ключове слово new. Наприклад:

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public Person(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }

    public void Display()
    {
        Console.WriteLine($"{FirstName} {LastName}");
    }
}

class Employee : Person
{
    public string Company { get; set; }
    public Employee(string firstName, string lastName, string company)
            : base(firstName, lastName)
```

```
    {
        Company = company;
    }
    public new void Display()
    {
      Console.WriteLine($"{FirstName} {LastName} працює у {Company}");
    }
}
```

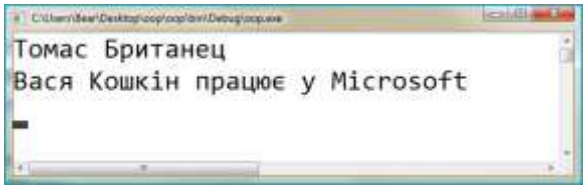
Метод Display виводить інформацію про компанію разом з ім'ям і прізвищем на консоль. Для цього визначається метод Display з ключовим словом new, який приховує реалізацію даного методу з базового класу.

В яких ситуаціях можна використовувати приховування? Наприклад, в прикладі вище метод Display в базовому класі не є віртуальним, ми не можемо його перевизначити, але, припустимо, нас не влаштовує його реалізація для похідного класу, тому ми можемо скористатися приховуванням, щоб визначити потрібний нам функціонал.

Використовуємо ці класи в програмі в методі Main:

```
class Program
{
    static void Main(string[] args)
    {
        Person tom = new Person("Томас", "Британец");
        tom.Display();      // Томас Британец

        Employee vasya = new Employee("Вася", "Кошкін", "Microsoft");
        vasya.Display();      // Вася Кошкін працює у Microsoft

        Console.ReadKey();
    }
}
```

Консольний вивід такий:



Подібним обазом ми можемо організувати приховування властивостей:

```
class Person
 {
     protected string name;
     public string Name
     {
         get { return name; }
         set { name = value; }
     }
```

```
}
class Employee : Person
{
    public new string Name
    {
        get { return "Employee " + base.Name; }
        set { name = value; }
    }
}
```

Різниця перевизначення і приховування методів

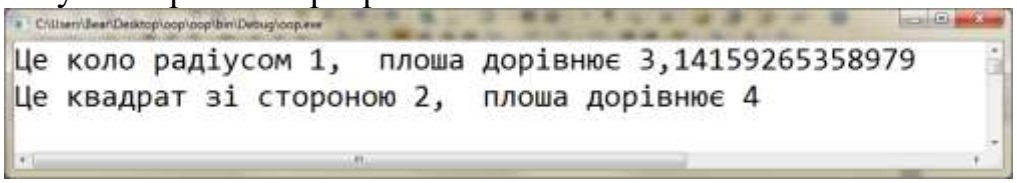
Раніше було розглянуто два способи зміни функціональності методів, успадкованих від базового класу - приховування і перевизначення. У чому різниця між двома цими способами?

Перевизначення

Візьмемо приклад з перевизначенням методів:

```
class Shape
{
   public double Area { get; set; }
   public virtual void GetInfo()
    {
        Console.WriteLine($"Це просто якась фігура");
    }
}
class Circle : Shape
{
    public double R { get; set; }
    public Circle(double R)
    {
        this.R = R;
        Area = Math.PI * R * R;
    }
    public override void GetInfo()
    {
        Console.WriteLine($"Це коло радіусом {R},  площа {Area}");
    }
}
class Square : Shape
{
    public double a { get; set; }
    public Square(double a)
    {
        this.a = a;
        Area = a*a;
    }
    public override void GetInfo()
    {
      Console.WriteLine($"Це квадрат зі стороною {a}, площа  {Area}");
    }
```

```
}
```
Використовуємо ці класи в програмі в методі Main:
```
class Program
{
    static void Main(string[] args)
    {
        Shape s = new Circle(1);
        s.GetInfo(); // виклик поліморфного методу
        s = new Square(2);
        s.GetInfo(); // виклик поліморфного методу

        Console.ReadKey();
    }
}
```
Результат роботи програми:



Перевизначення методу служить підставою для втілення одного з найефективніших в C# принципів: динамічної диспетчеризації методів, яка представляє собою механізм дозволу виклику під час виконання, а не компіляції.

Значення динамічної диспетчеризації методів полягає в тому, що саме завдяки їй в C # реалізується **динамічний поліморфізм**.

Для роботи з віртуальними методами компілятор формує таблицю віртуальних методів (Virtual Method Table або VMT). У неї записується адреси віртуальних методів. Для кожного класу створюється своя таблиця.

Реалізується так званий механізм пізнього зв'язування.

Коли створюється об'єкт класу, то компілятор передає в конструктор об'єкта спеціальний код, який пов'язує об'єкт і таблицю VMT.

А при виклику віртуального методу з об'єкта береться адреса його таблиці VMT. Потім з VMT витягується адреса методу і йому передається управління. Тобто процес вибору реалізації методу проводиться під час виконання програми. Власне так і виконується віртуальний метод. Слід враховувати, що тому що середовищі виконання спочатку необхідно отримати з таблиці VMT адресу потрібного методу, то це трохи уповільнює виконання програми.

Приховування

Тепер візьмемо ті ж класи, але замість перевизначення використовуємо приховування:

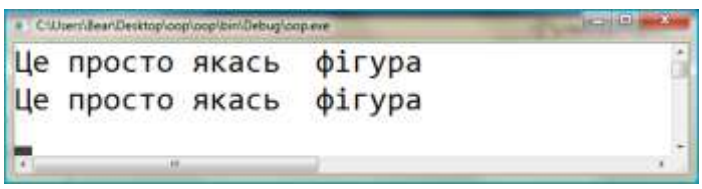
```
class Shape
{
   public double Area { get; set; }
   public virtual void GetInfo()
    {
```

```
            Console.WriteLine($"Це просто якась  фігура");
        }
    }
    class Circle : Shape
    {
        public double R { get; set; }
        public Circle(double R)
        {
            this.R = R;
            Area = Math.PI * R * R;
        }
        public new void GetInfo()
        {
            Console.WriteLine($"Це коло радіусом {R},  площа {Area}");
        }
    }
    class Square : Shape
    {
        public double a { get; set; }
        public Square(double a)
        {
            this.a = a;
            Area = a*a;
        }
        public new void GetInfo()
        {
         Console.WriteLine($"Це квадрат зі стороною {a},  площа  {Area}");
        }
    }
```

І подивимося, що буде в наступному випадку:

```
        static void Main(string[] args)
        {
            Shape s = new Circle(1);
            s.GetInfo(); // метод вже не поліморфний!
            s = new Square(2);
            s.GetInfo(); // метод вже не поліморфний!
            Console.ReadKey();
        }
```

Метод Main точно такий самий, як і в попередньому прикладі, але результат зовсім інший:



Змінна s представляє тип Shape, але зберігає посилання на об'єкт Circle і Square. Однак при виклику методу GetInfo буде виконуватися та версія методу, яка визначена саме в класі Shape, а не в класах Circle і Square. Чому? Класи Circle і Square ніяк не переважають метод GetInfo, успадкований від базового класу, а фактично визначає новий метод. Тому при виклику s.GetInfo(); викликається метод GetInfo з класу Shape.

Що дає перевизначення методів

Завдяки перевизначенню методів в C# підтримується **динамічний поліморфізм**.

В об'єктно-орієнтованому програмуванні поліморфізм грає дуже важливу роль, тому що він дозволяє визначити в загальному класі методи, які стають загальними для всіх похідних від нього класів, а в похідних класах – визначити конкретну реалізацію деяких або ж усіх цих методів.

Перевизначення методів – це ще один спосіб втілити в C# головний принцип поліморфізму: **один інтерфейс – багато методів**.

Вдале застосування поліморфізму частково залежить від правильного розуміння тієї особливості, що базові та похідні класи утворюють ієрархію, яка просувається від меншої до більшої спеціалізації. При належному застосуванні базовий клас надає всі необхідні елементи, які можуть використовуватися в похідному класі безпосередньо. А за допомогою віртуальних методів в базовому класі визначаються ті методи, які можуть бути самостійно реалізовані в похідному класі. Таким чином, поєднуючи спадкування з віртуальними методами, можна визначити в базовому класі загальну форму методів, які будуть використовуватися у всіх його похідних класах.

10.4 Абстрактні класи та члени класів

Іноді потрібно створити базовий клас, в якому визначається лише сама загальна форма для всіх його похідних класів, а наповнення її деталями надається кожному з цих класів. У такому класі визначається лише характер методів, які повинні бути конкретно реалізовані в похідних класах, а не в самому базовому класі. Подібна ситуація виникає, наприклад, у зв'язку з неможливістю отримати змістовну реалізацію методу в базовому класі. Саме така ситуація була продемонстрована в варіанті класу Shape з попереднього прикладу, де метод GetInfo() був просто визначений як заповнювач. Такий метод не обчислює і не виводить площа двовимірного об'єкта будь-якого типу.

У методу GetInfo() відсутнє змістовне визначення в базовому класі. У попередньому прикладі, він просто виводить якесь повідомлення. Такий спосіб може стати в нагоді при налагодженні, але на практиці він зазвичай не застосовується. Адже в базовому класі можуть бути оголошені методи, які повинні бути перевизначені в похідному класі, щоб цей клас став змістовним.

У подібних випадках потрібен якийсь спосіб, який гарантує, що в похідному класі дійсно будуть перевизначені всі необхідні методи. І такий спосіб в C# є. Він полягає у використанні абстрактного методу.

Абстрактний метод створюється за допомогою модифікатора типу **abstract**. У абстрактного метода відсутнє тіло, і тому він не реалізується в базовому класі. Це означає, що він **повинен бути** перевизначений у похідному класі, оскільки його варіант з базового класу просто непридатний для використання. Абстрактний метод автоматично стає віртуальним і не вимагає вказівки модифікатора virtual. Насправді спільне використання модифікаторів virtual і abstract є помилкою.

Для визначення абстрактного методу служить наведена нижче загальна форма:

```
abstract тип ім'я (список_параметрів);
```

Як ви бачите, у абстрактного метода відсутнє тіло. Абстрактними можуть бути також індексатори і властивості.

Клас, що містить один або більше абстрактних методів, повинен бути також оголошений як абстрактний, і для цього перед його оголошенням class вказується модифікатор abstract. А оскільки реалізація абстрактного класу не визначається повністю, то у нього не може бути об'єктів. Отже, спроба створити об'єкт абстрактного класу за допомогою оператора new призведе до помилки під час компіляції.

Коли похідний клас успадковує абстрактний клас, в ньому **повинні** бути реалізовані всі абстрактні методи базового класу. В іншому випадку похідний клас повинен бути також визначений як abstract. Таким чином, атрибут abstract успадковується до тих пір, поки не буде досягнута повна реалізація класу.

Використовуючи абстрактний клас, ми можемо вдосконалити клас Shape, що розглядався у попередньому прикладі. Для невизначеної двомірної фігури поняття площі не має ніякого сенсу, тому в наведеному нижче варіанті класу Shape метод GetInfo() і сам клас Shape оголошуються як abstract. Це означає, що у всіх класах, похідних від класу Shape, повинен бути перевизначений метод GetInfo().

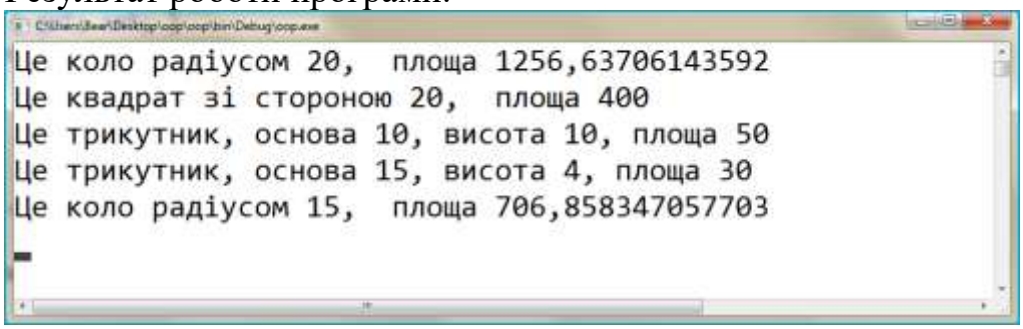
```
abstract class Shape // абстрактний клас
 {
    public double Area { get; set; }
    public abstract void GetInfo(); // абстрактний метод
 }
 class Circle : Shape
 {
     public double R { get; set; }
     public Circle(double R)
     {
         this.R = R;
         Area = Math.PI * R * R;
     }
     public override void GetInfo() // зобов'язані перевизначити!
     {
```

```
            Console.WriteLine($"Це коло радіусом {R},  площа {Area}");
        }
    }
    class Square : Shape
    {
        public double a { get; set; }
        public Square(double a)
        {
            this.a = a;
            Area = a*a;
        }
        public override void GetInfo() // зобов'язані перевизначити!
        {
         Console.WriteLine($"Це квадрат зі стороною {a},  площа {Area}");
        }
    }
    class Triangle : Shape
    {
        public double a { get; set; } // основа
        public double h { get; set; } // висота
        public Triangle(double a, double h)
        {
            this.a = a;
            this.h = h;
            Area = a * h / 2;
        }
        public override void GetInfo() // зобов'язані перевизначити!
        {
Console.WriteLine($"Це трикутник, основа {a}, висота {h}, площа {Area}");
        }
    }
```

Використовуємо ці класи в програмі в методі Main:

```
  class Program
    {
        static void Main(string[] args)
        {
            Shape []shape = new Shape[5]; // масив посилань!
            shape[0] = new Circle(20);
            shape[1] = new Square(20);
            shape[2] = new Triangle(10, 10);
            shape[3] = new Triangle(15, 4);
            shape[4] = new Circle(15);
            foreach (Shape s in shape)
                s.GetInfo(); // виклик поліморфного методу!

            Console.ReadKey();
        }
    }
```

Результат роботи програми:



Зверніть також увагу на те, що в класі Shape як і раніше є властивість Area, яка не оголошується з модифікатором abstract. В абстрактні класи цілком допускається включати методи і властивості, які можуть бути використані в своєму початковому вигляді в похідному класі. А обов'язковому перевизначенню в похідних класах підлягають тільки ті методи і властивості, які оголошені як abstract.

10.5 Клас object і його методи

У C# передбачений спеціальний клас object, який неявно вважається базовим класом для всіх інших класів і типів, включаючи і типи значень. Іншими словами, всі інші типи є похідними від object. Це, зокрема, означає, що змінна посилального типу object може посилатися на об'єкт будь-якого іншого типу. Крім того, змінна типу object може посилатися на будь-який масив, оскільки в C# масиви реалізуються як об'єкти. Формально ім'я object вважається в C# ще одним позначенням класу System.Object, що входить в бібліотеку класів для середовища .NET Framework.

У класі object визначаються методи, що наведені в таблиці. 10.1

Методи класу object

Таблиця 10.1.

Метод	Призначення
`public virtual bool Equals(object ob)`	Визначає, чи є викликаючий об'єкт таким же, як і об'єкт, доступний за посиланням ob
`public static bool Equals(object objA, object objB)`	Визначає, чи є об'єкт, доступний за посиланням objA, таким же, як і об'єкт, доступний за посиланням objB
`protected Finalize()`	Виконує завершальні дії перед "збиранням сміття". У C# метод Finalize () доступний за допомогою деструктора
`public virtual int GetHashCode()`	Повертає хеш-код, пов'язаний з викликаючим об'єктом
`public Type GetType()`	Отримує тип об'єкта під час виконання програми

Метод	Призначення
`protected object MemberwiseClone()`	Виконує неповне копіювання об'єкта, тобто копіюються тільки члени, але не об'єкти, на які посилаються ці члени
`public static bool ReferenceEquals(obj objA, object objB)`	Визначає, чи робляться посилання objA і objB на один і той же об'єкт
`public virtual string ToString()`	Повертає рядок, що описує об'єкт

Метод GetHashCode() повертає хеш-код, пов'язаний з викликаючим об'єктом. Цей хеш-код можна потім використовувати в будь-якому алгоритмі, де хешування застосовується як засіб доступу до збережених об'єктів. Слід, однак, мати на увазі, що стандартна реалізація методу GetHashCode() не придатна на всі випадки застосування.

Як згадувалося в розділі 9.8, якщо перевантажується оператор ==, то зазвичай доводиться перевизначати методи Equals(object) і GetHashCode(), оскільки найчастіше потрібно, щоб метод Equals(object) і оператор == функціонували однаково.

Коли ж переопределяется метод Equals(object), то слід перевизначити і метод GetHashCode(), щоб обидва методи виявилися сумісними.

Метод ToString () повертає символьний рядок, що містить опис того об'єкта, для якого він викликається. Крім того, метод ToString() автоматично викликається при виведенні вмісту об'єкта за допомогою методу WriteLine(). Цей метод переопределяется в багатьох класах, що дозволяє пристосовувати опис до конкретних типів об'єктів, що створюються в цих класах. Нижче наведено приклад застосування цього методу.

```
class Person
{
    public string Name { get; set; }
    public override string ToString()
    {
        if (String.IsNullOrEmpty(Name))
            return base.ToString();
        return Name;
    }
}
```

Упаковка та розпакування

Як пояснювалося вище, всі типи в C#, включаючи і прості типи значень, є похідними від класу object. Отже, посиланням типу object можна скористатися для звернення до будь-якого іншого типу, в тому числі і до типів значень.

Коли посилання на об'єкт класу object використовується для звернення до типу значення, то такий процес називається упаковкою (boxing). Упаковка призводить до того, що значення простого типу зберігається в екземплярі об'єкта, тобто "Упаковується" в об'єкті, який потім використовується як і будь-

який інший об'єкт. Але в будь-якому випадку упаковка відбувається автоматично. Для цього досить привласнити значення змінної посилального типу object, а про решту подбає компілятор C#.

Розпакування являє собою процес вилучення упакованого значення з об'єкта. Це робиться за допомогою явного приведення типу посилання на об'єкт класу об'єкт до відповідного типу значення. Спроба розпакувати об'єкт в інший тип може привести до помилки під час виконання.

При упаковці загальномовне середовище CLR обгортає значення в об'єкт типу System.Object і зберігає його в керованій купі (heap). Розпакування (unboxing), навпаки, передбачає перетворення об'єкта типу object до значимого типу. Упаковка і розпакування ведуть до зниження продуктивності, тому що системі треба здійснити необхідні перетворення. Однак в переважній більшості застосувань маємо виграш в продуктивності від зберігання простих типів даних в стеці, а не в купі, де треба прибирати сміття, значно переважує ці втрати.

```
class Program
{
    static void Main(string[] args)
    {
        int x;
        object obj;
        x = 10;
        obj = x; // упакувати значення змінної x в об'єкт
        int y = (int)obj; // розпакувати значення з об'єкта,
        //доступного за  посиланням obj, в змінну типу int
        Console.WriteLine(y);
        Console.ReadKey();
    }
}
```

10.6 Узагальнення

С виходом версії 2.0 фреймворк .NET став підтримувати узагальнені типи (generics), а також створення узагальнених методів. Щоб розібратися в особливості даного явища, спочатку подивимося на проблему, яка могла виникнути до появи узагальнених типів. Подивимося на прикладі. Припустимо, ми визначаємо клас для подання банківського рахунку. Наприклад, він міг би виглядати наступним чином:

```
class Account
    {
        public int Id { get; set; }
        public int Sum { get; set; }
    }
```

Клас Account визначає дві властивості: Id – унікальний ідентифікатор і Sum – суму на рахунку.

Тут ідентифікатор заданий як числове значення, тобто банківські рахунки будуть мати значення 1, 2, 3, 4 і так далі. Однак також для ідентифікатора використовуються і строкові значення. І у числових, і у строкових значень є свої плюси і мінуси. І на момент написання класу ми можемо точно не знати, що краще вибрати для зберігання ідентифікатора – рядки або числа. Або, можливо, цей клас буде використовуватися іншими розробниками, які можуть мати свою думку з даної проблеми.

І на перший погляд, щоб вийти з подібної ситуації, ми можемо визначити властивість Id як властивість типу object. Тому що тип object є універсальним типом, від якого успадковується всі типи, відповідно у властивостях подібного типу ми можемо зберегти і рядки, і числа:

```
class Account
{
    public object Id { get; set; }
    public int Sum { get; set; }
}
```

Потім можна було цей клас використовувати для створення банківських рахунків в програмі:

```
Account account1 = new Account { Sum = 5000 };
Account account2 = new Account { Sum = 4000 };
account1.Id = 2;
account2.Id = "4356";
int id1 = (int)account1.Id;
string id2 = (string)account2.Id;
Console.WriteLine(id1);
Console.WriteLine(id2);
```

Все ніби чудово працює, але таке рішення є не дуже оптимальним. Справа в тому що в цьому випадку ми стикаємося з такими явищами як упаковка (boxing) і розпакування (unboxing).

Так, при присвоєнні властивості Id значення типу int, відбувається упаковка цього значення в тип Object:

```
account1.Id = 2; // упаковка  значення int в тип object
```

Щоб отримати дані в змінну int, необхідно виконати розпакування:

```
int id1 = (int)account1.Id;  // Розпакування в тип int
```

Упаковка (boxing) передбачає перетворення об'єкта значимого типу (наприклад, типу int) до типу object. Як було сказано вище, упаковка і розпакування ведуть до зниження продуктивності, тому що системі треба здійснити необхідні перетворення.

Крім того, існує інша проблема – проблема безпеки типів. Так, ми отримаємо помилку під час виконання програми, якщо напишемо в такий спосіб:

```
Account account2 = new Account { Sum = 4000 };
account2.Id = "4356";
int id2 = (int)account2.Id;     // виняток InvalidCastException
```

Ми можемо не знати, який саме об'єкт являє Id, і при спробі отримати число в даному випадку ми зіткнемося з виключенням InvalidCastException.

Узагальнені типи дозволяють вказати конкретний тип, який буде використовуватися. Тому визначимо клас Account як узагальнений:

```
class Account<T>
    {
        public T Id { get; set; }
        public int Sum { get; set; }
    }
```

Кутові дужки в описі class Account <T> вказують, що клас є узагальненим, а тип T, укладений в кутові дужки, буде використовуватися цим класом. Необов'язково використовувати саме букву T, це може бути і будь-яка інша буква або набір символів. Причому зараз нам невідомо, що це буде за тип, це може бути будь-який тип. Тому параметр T в кутових дужках ще називається універсальним параметром, тому що замість нього можна підставити будь-який тип.

Наприклад, замість параметра T можна використовувати об'єкт int, тобто число, яке представляє номер рахунку. Це також може бути об'єкт string, або будь-який інший клас або структура:

```
Account<int> account1 = new Account<int> { Sum = 5000 };
Account<string> account2 = new Account<string> { Sum = 4000 };
account1.Id = 2; // упаковка не потрібна
account2.Id = "4356";
int id1 = account1.Id; // розпакування не потрібне
string id2 = account2.Id;
Console.WriteLine(id1);
Console.WriteLine(id2);
```

Оскільки клас Account є узагальненим, то при визначенні змінної після назви типу в кутових дужках необхідно вказати той тип, який буде використовуватися замість універсального параметра T. В даному випадку об'єкти Account Типізуються типами int і string:

```
Account<int> account1 = new Account<int> { Sum = 5000 };
Account<string> account2 = new Account<string> { Sum = 4000 };
```

Тому у першого об'єкта account1 властивість Id матиме тип int, а у об'єкта account2 – тип string.

При спробі привласнити значення властивості Id змінної іншого типу ми отримаємо помилку компіляції:

```
Account<string> account2 = new Account<string> { Sum = 4000 };
account2.Id = "4356";
int id1 = account2.Id;  // ошибка компіляції
```

Значення за замовчуванням

Іноді виникає необхідність привласнити змінним універсальних параметрів деяке початкове значення, в тому числі і null. Але безпосередньо ми його привласнити не можемо:

```
T id = null;
```

У цьому випадку нам треба використовувати оператор default (T). Він привласнює посилальним типам як значення null, а типам значень – значення 0:

```
class Account<T>
{
    T id = default(T);
}
```

Статичні поля узагальнених класів

При типізації узагальненого класу певним типом буде створюватися свій набір статичних членів. Наприклад, в класі Account визначено наступне статичне поле:

```
class Account<T>
{
    public static T session;

    public T Id { get; set; }
    public int Sum { get; set; }
}
```

Тепер типизуємо клас двома типами int і string:

```
Account<int> account1 = new Account<int> { Sum = 5000 };
Account<int>.session = 5436;

Account<string> account2 = new Account<string> { Sum = 4000 };
Account<string>.session = "45245";
Console.WriteLine(Account<int>.session);      // 5436
Console.WriteLine(Account<string>.session);   // 45245
```

В результаті для Account <string> і для Account <int> буде створена своя змінна session.

Використання декількох універсальних параметрів

Узагальнення можуть використовувати кілька універсальних параметрів одночасно, які можуть представляти різні типи:

```
class Transaction<U, V>
{
    public U FromAccount { get; set; } // з якого рахунку переклад
```

```
        public U ToAccount { get; set; }   // на який рахунок переклад
        public V Code { get; set; }        // код операції
        public int Sum { get; set; }       // сума переказу
    }
```

Тут клас Transaction використовує два універсальних параметра. Застосуємо даний клас:

```
Account<int> acc1 = new Account<int> { Id = 1857, Sum = 4500 };
Account<int> acc2 = new Account<int> { Id = 3453, Sum = 5000 };

Transaction<Account<int>, string> transaction1 = new
Transaction<Account<int>, string>
            {
                FromAccount = acc1,
                ToAccount = acc2,
                Code = "45478758",
                Sum = 900
            };
```

Тут об'єкт Transaction типизується типами Account <int> і string. Тобто в якості універсального параметра U використовується клас Account <int>, а для параметра V - тип string. При цьому, як можна помітити, клас, яким Типізуються Transaction, сам є узагальненим.

Узагальнені методи

Крім узагальнених класів можна також створювати узагальнені методи, які точно також будуть використовувати універсальні параметри. Наприклад:

```
    class Cat
{
    public string Name { get; set; }
}
class Program
{
    static void Main(string[] args)
    {

        int x = 7;
        int y = 25;
        Swap<int>(ref x, ref y);
        Console.WriteLine($"x={x}    y={y}");   // x=25   y=7
        string s1 = "hello";
        string s2 = "bye";
        Swap<string>(ref s1, ref s2);
        Console.WriteLine($"s1={s1}    s2={s2}"); // s1=bye   s2=hello
        Cat c1 = new Cat();
        c1.Name = "Вася";
        Cat c2 = new Cat();
        c2.Name = "Лелик";
        Swap<Cat>(ref c1, ref c2);
        Console.WriteLine($"cat1={c1.Name}    Cat2={c2.Name}");
```

```
        Console.ReadKey();
    }
    public static void Swap<T>(ref T x, ref T y)
    {
        T temp = x;
        x = y;
        y = temp;
    }
```

Тут визначено узагальнений метод Swap, який приймає параметри по посиланню і змінює їх значення. При цьому в даному випадку не важливо, який тип представляють ці параметри.

У методі Main викликаємо метод Swap, тіпізіруя його певним типом і передаємо йому деякі значення.

Резюме

1. Спадкування реалізує механізми розширення базового класу. Базовий клас задає загальні ознаки і загальну поведінку для класів-спадкоємців. Загальні властивості і методи успадковуються від базового класу. До них додаються і визначаються нові властивості і методи.
2. В C# структури не підтримують спадкування.
3. В C# не підтримується множинне спадкування, клас може успадковуватися тільки від одного класу.
4. Якщо клас оголошений з модифікатором sealed, то від цього класу не можна створювати похідні класи.
5. При виклику конструктора класу спочатку відпрацьовують конструктори базових класів і тільки потім конструктори похідних.
6. Щоб зробити метод базового класу доступним для перевизначення, цей метод визначається з модифікатором virtual. Ми можемо перевизначити цей метод, але можемо і не перевизначати.
7. Для того, щоб перевизначити віртуальний метод, в класі-спадкоємці визначається метод з модифікатором override, який має таке ж саме ім'я і набір параметрів.
8. Перевизначення методу реалізує однин з найефективніших в ООП принципів: динамічний поліморфізм.
9. Абстрактний метод створюється за допомогою модифікатора типу abstract. У абстрактного методу відсутнє тіло, і тому він не реалізується в базовому класі.
10. Абстрактний метод повинен бути перевизначений у похідному класі. В іншому випадку похідний клас повинен бути також визначений як abstract.
11. Коли посилання на об'єкт класу object використовується для звернення до типу значення, то такий процес називається упаковкою (boxing).
12. Розпаковка являє собою процес вилучення упакованого значення з об'єкта. Це робиться за допомогою явного приведення типу посилання на об'єкт класу об'єкт до відповідного типу значення.

13. Узагальнені типи дозволяють вказати конкретний тип, який буде використовуватися. Кутові дужки в описі class Account <T> вказують, що клас є узагальненим, а тип T, укладений в кутові дужки, буде використовуватися цим класом.

14. Крім узагальнених класів можна також створювати узагальнені методи, які точно також будуть використовувати універсальні параметри.

Завдання до виконання

1. Описати базовий клас (можливо, абстрактний), у якому за допомогою віртуальних або абстрактних методів та властивостей задається інтерфейс для похідних класів відповідно варіанту.
2. Показати на прикладі одного з методів, присутніх у кожному класі, використання поліморфізму.

Варіанти завдань:

1. Квіткарка. Визначити ієрархію квітів. Створити кілька об'єктів-квітів. Зібрати букет з визначенням його вартості.
2. Новорічний подарунок. Визначити ієрархію цукерок та інших солодощів. Створити кілька об'єктів-цукерок. Зібрати дитячий подарунок з визначенням його ваги.
3. Електрик. Визначити ієрархію електроприладів. Включити деякі в розетку. Порахувати споживану потужність
4. Дієта. Визначити ієрархію овочів. Зібрати в салат. Порахувати калорійність.
5. Меломан. Визначити ієрархію музичних композицій. Записати на диск збірку. Порахувати тривалість.
6. Камені. Визначити ієрархію дорогоцінних і напівкоштовних каменів. Відібрати камені для намиста. Порахувати загальну вагу (в каратах) і вартість.
7. Зброя. Визначити ієрархію зброї (холодної та вогнепальної). Озброїти військовий підрозділ. Порахувати вартість.
8. Транспорт. Визначити ієрархію пасажирського транспорту. Створити набір маршрутів для переміщення і точки А в точку Б. Порахувати загальну вартість проїзду.
9. Авіакомпанія. Визначити ієрархію літаків. Створити авіакомпанію. Порахувати загальну місткість (в пасажирах).
10. Автомобілі. Визначити ієрархію легкових автомобілів. Створити автопарк організації. Порахувати вартість автопарку.
11. Розробити систему класів для представлення шахової фігури. Базовий клас – фігура, від цього класу спадкуються кінь, слон, король, ферзь. Повинні бути визначені: поточне положення фігури на дошці, симуляція наступного ходу (усі можливі ходи з поточного положення) та власне сам хід (зміна поточного положення фігури). Вивести позицію на шахівниці.

12. Базовий клас – Тіло, похідні – Куля, Паралелепіпед, Тетраедр. Обов’язкові методи – обчислення площі поверхні та об’єму. Архімед кинув у повну до країв ванну золоту кулю, тетраєдр і паралелепіпед. Який об’єм води витече?

Тести для закріплення матеріалу

1. Що з перерахованого є прикладом поліморфізму?
   а) наявність в класі декількох деструкторів;
   б) використання віртуальних функцій;
   в) створення декількох об'єктів класу;
   г) всі відповіді невірні.

2. Яке поле дозволяє тільки об'єктам класів-спадкоємців звертатися до поля?
   а) public
   б) protected
   в) private
   г) new

3. Просте спадкування передбачає, що
   а) клас має одного батька;
   б) клас не має батьківських класів;
   в) батьківський клас пов'язаний з нащадком;
   г) інше.

4. Який синтаксис використовується для вказівки класу батька в C #?
   а) class ChildClass : ParentClass
   б) class ChildClass : base ParentClass
   в) class ChildClass : parent ParentClass
   г) new ChildClass : ParentClass

5. Віртуальний метод (virtual) – це
   А) метод, який не має коду в реалізації;
   Б) метод, оголошений в базовому класі і який може бути перевизначений в похідному;
   В) статичний метод абстрактного класу;
   Г) інше.
6. Абстрактний метод (abstract) – це
   А) метод, який не має реалізацію, тільки опис методу, який має бути реалізований в похідних класах;
   Б) метод, який не має реалізацію, тільки опис методу, який може бути реалізований в похідних класах;
   В) статичний метод абстрактного класу;
   Г) інше.

7. Якщо в базовому класі метод оголошений як віртуальний, то в похідному класі метод з таким же ім'ям і набором параметрів буде
    а) звичайним методом;
    б) константним методом;
    в) статичним методом;
    г) віртуальним методом

РОЗДІЛ 11. ІНТЕРФЕЙСИ

11.1 Введення в інтерфейси

Іноді в об'єктно-орієнтованому програмуванні корисно визначити, що саме повинен робити клас, але не як він повинен це робити. Прикладом тому може служити згадуваний раніше абстрактний метод.

В абстрактному методі визначаються повертаємий тип і сигнатура методу, але не надається його реалізація.

А в похідному класі повинна бути забезпечена своя власна реалізація кожного абстрактного методу, визначеного в його базовому класі. Таким чином, абстрактний метод визначає інтерфейс, але не реалізацію методу. Звичайно, абстрактні класи і методи приносять відому користь, але покладений в їх основу принцип може бути розвинений далі. У C# передбачено поділ інтерфейсу класу і його реалізації за допомогою ключового слова interface.

З точки зору синтаксису інтерфейси подібні абстрактним класам. Але в інтерфейсі у жодного з методів не повинно бути тіла. Це означає, що в інтерфейсі взагалі не надається ніякої реалізації. У ньому вказується тільки, що саме слід робити, але не як це робити. Як тільки інтерфейс буде визначено, він може бути реалізований в будь-якій кількості класів. Крім того, в одному класі може бути реалізована будь-яка кількість інтерфейсів.

Для визначення інтерфейсу використовується ключове слово interface. Як правило, назви інтерфейсів в C# починаються з великої літери I, наприклад, IComparable, IEnumerable (так звана угорська нотація), однак це не обов'язкова вимога, а більше стиль програмування. Наприклад, інтерфейс IMovable:

```
interface IMovable
{
    void Move();
}
```

В даному випадку інтерфейс визначає метод Move, який буде представляти деяке пересування. Він не приймає ніяких параметрів і нічого не повертає.

В оголошенні методів інтерфейсу використовуються тільки їх повертаємий тип і сигнатура. Вони, по суті, є абстрактними методами.

Ще один момент в оголошенні інтерфейсу: всі його члени – методи і властивості не мають модифікаторів доступу, але фактично за замовчуванням доступ **public**, тому що мета інтерфейсу – визначення функціоналу для реалізації його класом. Тому весь функціонал повинен бути відкритий для реалізації.

В цілому інтерфейси можуть визначати наступні сутності:

– методи;

– властивості;

– індексатори;

– події.

Однак інтерфейси не можуть визначати статичні члени, змінні, константи.

Потім якийсь клас або структура можуть застосувати даний інтерфейс:

```
interface IMovable
{
    void Move();
}

// застосування інтерфейсу в класі
class Person : IMovable
{
    public void Move()
    {
        Console.WriteLine("Людина йде");
    }
}
// застосування інтерфейсу в структурі
struct Car : IMovable
{
    public void Move()
    {
        Console.WriteLine("Машина їде");
    }
}
```

Клас або структура повинні реалізувати всі методи і властивості застосовуваних інтерфейсів. При цьому оскільки всі методи і властивості інтерфейсу є публічними, при реалізації цих методів і властивостей в класі до них можна застосовувати тільки модифікатор public.

Якщо клас або структура не реалізують будь-які властивості або методи інтерфейсу, то ми зіткнемося з помилкою на етапі компіляції.

Однак також можна і не реалізувати методи, зробивши їх абстрактними, переклавши право їх реалізації на похідні класи:

```
interface IMovable
{
    void Move();
}
abstract class Person : IMovable
{
    public abstract void Move();
}
class Driver : Person
{
    public override void Move()
    {
        Console.WriteLine("Водій веде машину");
    }
}
```

11.2 Множинна реалізація інтерфейсів

Інтерфейси мають ще одну важливу функцію: в C# не підтримується множинне спадкування, тобто ми можемо успадкувати клас тільки від одного класу, на відміну, скажімо, від мови C++, де множинне спадкування можна використовувати. Інтерфейси дозволяють частково обійти це обмеження, оскільки в C# клас може реалізувати відразу кілька інтерфейсів. Всі реалізовані інтерфейси вказуються через кому:

```
class myClass: myInterface1, myInterface2, myInterface3, ...
{

}
```

Розглянемо на прикладі:

```
interface IAccount
{
    int CurrentSum { get; } // Поточна сума на рахунку
    void Put(int sum);      // Покласти гроші на рахунок
    void Withdraw(int sum); // Взяти з рахунку
}
interface IClient
{
    string Name { get; set; }
}
class Client : IAccount, IClient
{
    int _sum; // Змінна для зберігання суми
    public string Name { get; set; }
    public Client(string name, int sum)
    {
        Name = name;
        _sum = sum;
    }

    public int CurrentSum { get { return _sum; } }

    public void Put(int sum) { _sum += sum; }

    public void Withdraw(int sum)
    {
        if (_sum >= sum)
        {
            _sum -= sum;
        }
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Client client = new Client("Вася", 200);
        client.Put(30);
        Console.WriteLine(client.CurrentSum); // 230
        client.Withdraw(100);
        Console.WriteLine(client.CurrentSum); // 130
        Console.ReadKey();
    }
}
```

В даному випадку визначені два інтерфейси. Інтерфейс IAccount визначає властивість CurrentSum для поточної суми грошей на рахунку і два метода Put і Withdraw для додавання грошей на рахунок і вилучення грошей. Інтерфейс IClient визначає властивість для зберігання імені клієнта.

Зверніть увагу, що властивості CurrentSum і Name в інтерфейсах схожі на автосвойства, але це не автосвойства. При реалізації ми можемо розгорнути їх у повноцінні властивості, або ж зробити автосвойствами.

Клас Client реалізує обидва інтерфейси і потім застосовується в програмі.

11.3 Інтерфейси в перетвореннях типів

Все сказане щодо перетворення типів характерно і для інтерфейсів. Оскільки клас Client реалізує інтерфейс IAccount, то змінна типу IAccount може зберігати посилання на об'єкт типу Client:

```
static void Main(string[] args)
{
  // Всі об'єкти Client є об'єктами IAccount
  IAccount account = new Client("Вася", 200);
  account.Put(200);
  Console.WriteLine(account.CurrentSum); // 400
  // Не всі об'єкти IAccount є об'єктами Client, необхідно явне приведення
  Client client = (Client)account;
  // Інтерфейс IAccount не має властивості Name, необхідно явне приведення
  string clientName = ((Client)account).Name;
  Console.ReadKey();
 }
```

Перетворення від класу до його інтерфейсу, як і перетворення від похідного типу до базового, виконується автоматично. Тому що будь-який об'єкт Client реалізує інтерфейс IAccount.

Зворотне перетворення – від інтерфейсу до реалізуючого його класу буде аналогічно перетворенню від базового класу до похідного. Не кожен об'єкт IAccount є об'єктом Client, тому для подібного перетворення необхідна операція приведення типів. І якщо ми хочемо звернутися до методів класу Client, які не

визначені в інтерфейсі IAccount, але є частиною класу Client, то нам треба явним чином виконати перетворення типів:

```
string clientName = ((Client)account).Name;
```

11.4 Спадкування інтерфейсів

Інтерфейси, як і класи, можуть успадковуватися:

```
interface IAction
{
    void Move();
}
interface IRunAction : IAction
{
    void Run();
}
class BaseAction : IRunAction
{
    public void Move()
    {
        Console.WriteLine("Move");
    }
    public void Run()
    {
        Console.WriteLine("Run");
    }
}
```

При застосуванні цього інтерфейсу клас BaseAction повинен буде реалізувати як методи і властивості інтерфейсу IRunAction, так і методи і властивості базового інтерфейсу IAction.

Однак на відміну від класів ми не можемо застосовувати до інтерфейсів модифікатор sealed (а також до методів інтерфейсів), щоб заборонити спадкування інтерфейсів.

Також ми не можемо застосовувати до інтерфейсів модифікатор abstract, оскільки інтерфейс фактично вже надає абстрактний функціонал, який повинен бути реалізований в класі.

Однак методи інтерфейсів можуть використовувати ключове слово new для приховування методів з базового інтерфейсу.

При наслідуванні інтерфейсів, як і при наслідуванні класів, слід враховувати, що похідний інтерфейс повинен мати той же рівень доступу або більш суворий, ніж базовий інтерфейс. наприклад:

```
public interface IAction
{
    void Move();
}
internal interface IRunAction : IAction
{
    void Run();
```

```
}
```

Але не навпаки. Наприклад, в наступному випадку ми отримаємо помилку, і програма НЕ скомпілюється, тому що похідний інтерфейс має менш суворий рівень доступу, ніж базовий:

```
internal interface IAction
{
    void Move();
}
public interface IRunAction : IAction  // помилка
// IRunAction может быть только internal
{
    void Run();
}
```

11.5 Явна та неявна реалізація

В прикладах застосування інтерфейсів, яки були розглянути вище, використовувалась так звана неявна реалізація інтерфейсу – кожен член інтерфейсу реалізується з сигнатурою, що відповідає визначенню в інтерфейсі.

При явній реалізації вказується назва методу або властивості разом з назвою інтерфейсу, при цьому ми не можемо використовувати модифікатор **public**, тобто методи є закритими:

```
interface IAction
{
    void Move();
}
class BaseAction : IAction
{
    void IAction.Move()
    {
        Console.WriteLine("Move in Base Class");
    }
}
```

Слід враховувати, що при явній реалізації інтерфейсу його методи і властивості не є частиною інтерфейсу класу. Тому безпосередньо через об'єкт класу ми до них звернутися не зможемо:

```
    static void Main(string[] args)
    {
        BaseAction action = new BaseAction();
  ((IAction)action).Move();   // необхідно приведення до типу IAction

        // або так
        IAction action2 = new BaseAction();
        action2.Move();
        Console.ReadKey();
```

```
    }
```

В якої ситуації може дійсно знадобитися явна реалізація інтерфейсу? Наприклад, коли клас застосовує кілька інтерфейсів, але вони мають один і той же метод з одним і тим же повертаємим типом і одним і тим же набором параметрів:

```
    interface ISchool
    {
     void Study ();
    }

    interface IUniversity
    {
     void Study ();
    }

class Person: ISchool, IUniversity
{
    // неявна реалізація
     public void Study ()
     {
         Console.WriteLine ("Навчання в школі або в університеті");
     }
}
```

Клас Person визначає один метод Study(), створюючи одну спільну реалізацію для обох застосованих інтерфейсів. І незалежно від того, чи будемо ми розглядати об'єкт Person як об'єкт типу ISchool або IUniversity, результат методу буде один і той же.

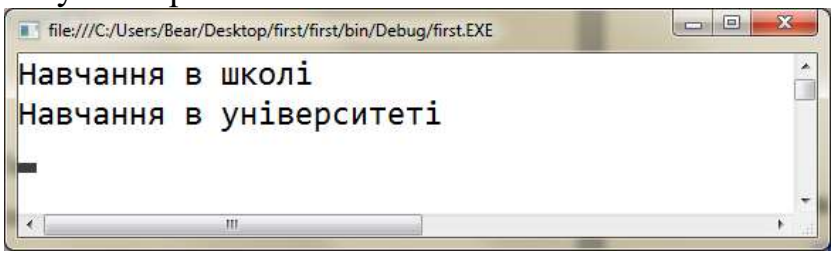
Щоб розмежувати реалізовані інтерфейси, треба явним чином застосувати інтерфейс:

```
    class Person: ISchool, IUniversity
    {
         // явна реалізація
         void ISchool.Study ()
         {
             Console.WriteLine ("Навчання в школі");
         }
         // явна реалізація
         void IUniversity.Study ()
         {
             Console.WriteLine ("Навчання в університеті");
         }
    }
```

Використання:

```
  static void Main(string[] args)
    {
        Person p = new Person();
        ISchool IS = p;
        IS.Study();
        IUniversity IU = p;
        IU.Study();

        Console.ReadKey();
    }
```

Результат роботи:



11.6 Динамічна зміна поведінки

У розділі 6.4 розглядался приклад застосування патерна Strategy для динамічой зміни алгоритмів під час виконання програми. Той самий приклад, реалізований на мові C#.

Для програми стиснення файлів оголошуємо інтерфейс ICompression загальний для всіх алгоритмів і використовуваний класом Compressor.

```
interface ICompression
    {
        void Compress();
    }
    class Compressor
    {
        public ICompression Compression { get; set; }
        public Compressor(ICompression Compression)
        {
            this.Compression = Compression;
        }
        public void ExecuteCompress()
        {
            Compression.Compress();
        }
        public void ChangeAlgorythm(ICompression Compression)
        {
            this.Compression = Compression;
        }

    }
```

```
class ZipCompression :  ICompression
{
   public void Compress()
   {
    Console.WriteLine("Метод реалізує алгоритм ZIP-компресії");
   }
}
class RarCompression : ICompression
{
    public void Compress()
    {
     Console.WriteLine("Метод реалізує алгоритм Rar-компресії");
    }
}
    class ArjCompression : ICompression
{
    public void Compress()
    {
    Console.WriteLine("Метод реалізує алгоритм Arj-компресії");
    }
}
class MyCompression : ICompression
{
   public void Compress()
   {
    Console.WriteLine("Реалізація власного алгоритму компресії");
   }
}
internal class Program
{
    static void Main(string[] args)
    {
        Console.OutputEncoding = Encoding.UTF8;

        Compressor comp = new Compressor(new ZipCompression());
        comp.ExecuteCompress();

        comp = new Compressor(new RarCompression());
        comp.ExecuteCompress();

        comp = new Compressor(new ArjCompression());
        comp.ExecuteCompress();

        comp = new Compressor(new MyCompression());
        comp.ExecuteCompress();

        Console.Write("Динамічна зміна алгоритму.Тепер ");
        comp.ChangeAlgorythm(new RarCompression());
        comp.ExecuteCompress();
        Console.ReadKey();
    }
}
```

```
H:\Методички\посібник ООП 3 видання\StrategyCS\bin\Debug\net7.0\StrategyCS.exe
Метод реалізує алгоритм ZIP-компресії
Метод реалізує алгоритм Rar-компресії
Метод реалізує алгоритм Arj-компресії
Реалізація власного алгоритму компресії
Динамічна зміна алгоритму. Тепер Метод реалізує алгоритм Rar-компресії
```

На рисунку 11.1. показано діаграма класів для патерна Strategy на C#.

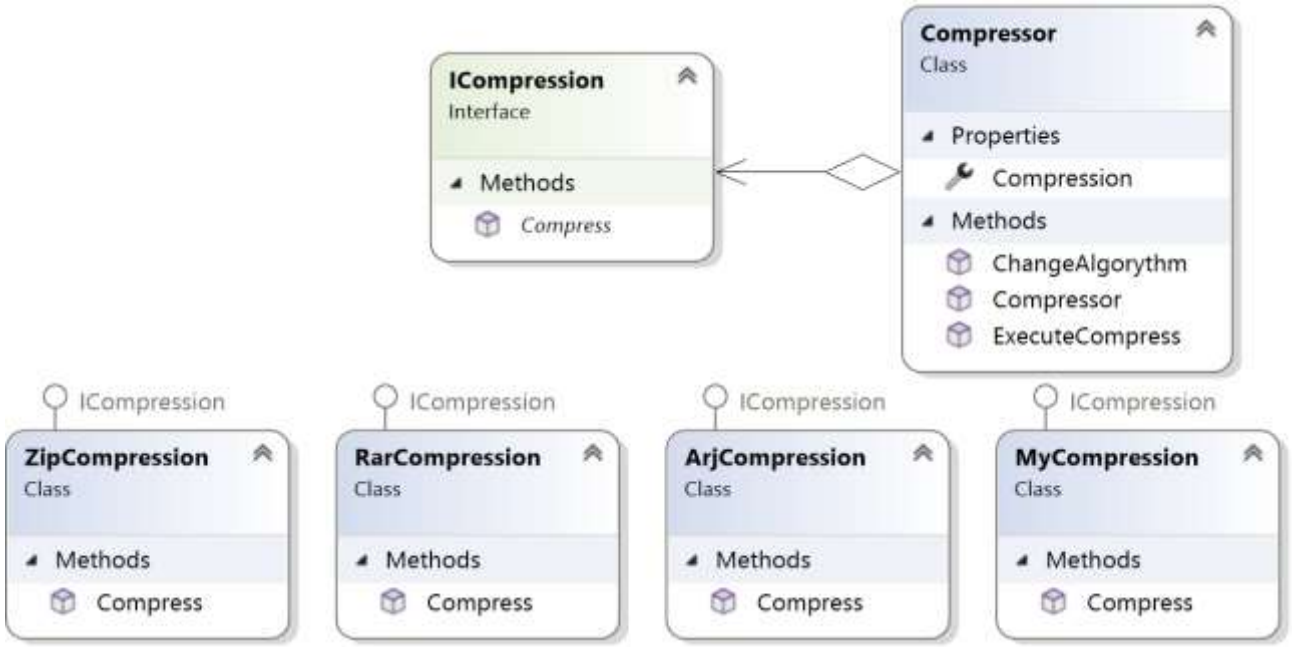


Рис. 11.1. Діаграма класів для патерна Strategy

Резюме

1. Інтерфейс представляє посилальний тип, який визначає набір методів і властивостей, але не реалізує їх. Потім цей функціонал реалізують класи і структури, які застосовують ці інтерфейси.
2. З точки зору синтаксису інтерфейси подібні абстрактним класам. Але в інтерфейсі у жодного з методів не повинно бути тіла.
3. В оголошенні методів інтерфейсу використовуються тільки їх повертаємий тип і сигнатура. Вони, по суті, є абстрактними методами.
4. Всі члени інтерфейсу не мають модифікаторів доступу, але фактично за замовчуванням доступ public.
5. Для визначення інтерфейсу використовується ключове слово interface.
6. Клас або структура повинні реалізувати всі методи і властивості застосовуваних інтерфейсів.
7. Методи, що реалізують інтерфейс, повинні бути оголошені як public.
8. В C# ми можемо успадкувати клас тільки від одного базового класу і від богатьох інтерфейсів.
9. Інтерфейси, як і класи, можуть успадковуватися.

10. При явній реалізації інтерфейсів вказується назва методу або властивості разом з назвою інтерфейсу, при цьому ми не можемо використовувати модифікатор public.

11. Явна реалізація інтерфейсу потрібна, коли клас застосовує кілька інтерфейсів, але вони мають один і той же метод з одним і тим же повертаємим типом і одним і тим же набором параметрів.

Завдання до виконання

1. Скласти програму, яка виконує наступне:
   – Створює масив з N екземплярів класу, створеного відповідно до варіанта. Клас є нащадком інтерфейсу IComparable.
   – Виводить інформацію про об'єкти масиву.
   – Сортує масив за заданою ознакою (див. Варіант), використовуючи метод Sort класу Array.
   – Виводить інформацію про об'єкти відсортованого масиву.

Варіанти завдань:

Варіант 1. Створити базовий клас «циліндр», такий, що:

a) його екземпляр містить розмір радіусу і висоти.

b) його конструктор без параметра створює екземпляр зі значенням 0, а конструктор з параметрами створює екземпляр з відповідним значенням радіусу і висоти.

c) його методи дозволяють отримувати і привласнювати значення радіуса, висоти і об'єму, виконуючи при цьому відповідні перетворення.

d) функція print_val () виводить на екран значення його об'єму.

Сортувати масив за об'ємом.

Варіант 2. Створити базовий клас «куля», такий, що:

a) його екземпляр містить розмір радіуса.

b) його конструктор без параметра створює екземпляр зі значенням 0, а конструктор з параметрами створює екземпляр з відповідним значенням радіусу.

c) його методи дозволяють отримувати і привласнювати значення радіусу і об'єму, виконуючи при цьому відповідні перетворення.

d) функція print_val () виводить на екран значення його об'єму.

Сортувати масив по радіусу.

Варіант 3. Створити базовий клас «конус», такий, що:

a) його екземпляр містить розмір радіусу і висоти.

b) його конструктор без параметра створює екземпляр зі значенням 0, а конструктор з параметрами створює екземпляр з відповідним значенням радіусу і висоти.

c) його методи дозволяють отримувати і привласнювати значення радіуса, висоти і об'єму, виконуючи при цьому відповідні перетворення.

d) функція print_val () виводить на екран значення його об'єму. Сортувати масив за об'ємом.

Варіант 4. Створити базовий клас «трапеція», такий, що:

a) його екземпляр містить розмір двох підстав і висоти.

b) його конструктор без параметра створює екземпляр зі значенням 0, а конструктор з параметрами створює екземпляр з відповідним значенням підстав і висоти.

c) його методи дозволяють отримувати і привласнювати значення підстав, висоти і площі, виконуючи при цьому відповідні перетворення.

d) функція print_sqrt () виводить на екран значення її площі.

Сортувати масив по площі.

Варіант 5. Створити базовий клас «смайлик», такий, що:

a) його екземпляр містить розмір радіусу особи R і радіуса r очей.

b) його конструктор без параметра створює екземпляр зі значенням 0, а конструктор з параметрами створює екземпляр з відповідним значенням R і r.

c) його методи дозволяють отримувати і привласнювати значення R, r і площі, виконуючи при цьому відповідні перетворення.

d) функція print_sqrt () виводить на екран значення її площі (без площі очей).

Сортувати масив по площі.

Тести для закріплення матеріалу

1. Що неправильно у визначенні наступних інтерфейсів?

```
public abstract interface IAction
{
    void Move();
}
internal interface IRunAction : IAction
{
    void Run();
}
```

а) інтерфейси не можуть мати модифікатор доступу internal;

б) інтерфейси не можуть мати модифікатор abstract;

в) інтерфейс IRunAction повинен визначити метод Move, тому що базовий інтерфейс IAction є абстрактним;

г) від абстрактного інтерфейсу можна успадковуватися.

2. Є наступна програма:

```
class Program
{
    static void Main(string[] args)
    {
        IAction action = new RunAction();
        action.Move();

        Console.ReadKey();
    }
}

class RunAction : IRunAction
{
    internal void Move()
    {
        Console.WriteLine("I am running");
    }
}

interface IAction
{
    void Move();
}
interface IRunAction : IAction
{
    new void Move();
}
```

Чому ця програма не скомпілюється?

а) Тому що метод Move в IRunAction визначено з ключовим словом new.

б) Тому що в обох інтерфейсах визначено метод Move, і програма не знає, який саме з цих методів викликати.

в) Тому що метод Move в класі RunAction визначено з модифікатором internal.

г) Тому що клас RunAction повинен визначити дві реалізації методу Move для кожного з інтерфейсів.

3. Чи підтримує C # множинне спадкування?

а) Не підтримує множинне спадкування.

б) Підтримує множинне спадкування у вигляді спадкування від класу і декількох інтерфейсів.

в) Не підтримує множинне спадкування у вигляді спадкування від класу і декількох інтерфейсів.

г) Підтримує множинне спадкування у вигляді спадкування від декількох класів.

4. Клас Tester реалізує інтерфейси IFoo і IBar:

```
interface IFoo
 {
     void Execute();
 }
 interface IBar
 {
     void Execute();
 }
 class Tester : IFoo, IBar
 {
     public void Execute()
     {
         Console.WriteLine("Tester Executes");
     }
 }
```

Метод Execute () якого саме інтерфейсу реалізує клас Tester?

а) Реалізує метод інтерфейсу Ifoo.
б) Реалізує метод інтерфейсу IBar.
в) Помилка компіляції через невизначеність.
г) Реалізує методи обох інтерфейсів.

5. Клас Tester реалізує інтерфейси IFoo і IBar:

```
interface IFoo
 {
     void Execute();
 }
 interface IBar
 {
     void Execute();
 }
 class Tester : IFoo, IBar
 {
     void IFoo.Execute()
     {
         Console.WriteLine("IFoo Executes");
     }
     void IBar.Execute()
     {
         Console.WriteLine("IBar Executes");
     }
 }
```

Реалізація методу Execute () з якого інтерфейсу буде викликана в наступному коді:

```
Tester t = new Tester();
t.Execute();
```

а) З інтерфейсу Ifoo.
б) З інтерфейсу IBar.
в) Помилка компіляції.
г) Почергово з обох інтерфейсів.

РОЗДІЛ 12. ДЕЛЕГАТИ, ПОДІЇ, ЛЯМБДА-ВИЗАЗИ

12.1 Делегати

Делегат – це об'єкт, який може посилатися на метод. Отже, коли створюється делегат, то в результаті виходить об'єкт, що містить посилання на метод. Більш того, метод можна викликати за цим посиланням. Іншими словами, делегат дозволяє викликати метод, на який він посилається. Нижче буде показано, наскільки дієвим виявляється такий принцип.

Слід особливо підкреслити, що один і той же делегат може бути використаний для виклику різних методів під час виконання програми, для чого достатньо змінити метод, на який посилається делегат. Таким чином, метод, що викликається делегатом, визначається під час виконання, а не в процесі компіляції. У цьому, власне, і полягає головна перевага делегата.

Визначення делегатів

Для оголошення делегата використовується ключове слово delegate, після якого йде повертаємий тип, назва і параметри. Наприклад:

```
delegate void Message();
```

Делегат Message в якості повертаємого типу має тип void (тобто нічого не повертає) і не приймає ніяких параметрів. Це означає, що цей делегат може вказувати на будь-який метод, який не приймає ніяких параметрів і нічого не повертає.

Розглянемо застосування цього делегата:

```
class Program
 {
     delegate void Message(); // 1. Оголошуємо делегат

     static void Main(string[] args)
     {
         Message mes; // 2. Створюємо змінну делегата
         if (DateTime.Now.Hour < 12)
         {
           mes = GoodMorning; // 3. Надаємо цієї змінної адресу методу
         }
         else
         {
             mes = GoodEvening;
         }
         mes(); // 4. Викликаємо метод
         Console.ReadKey();
     }
     private static void GoodMorning()
     {
         Console.WriteLine("Good Morning");
     }
     private static void GoodEvening()
     {
```

```
            Console.WriteLine("Good Evening");
        }
    }
```

Тут спочатку ми визначаємо делегат:

```
delegate void Message(); // 1. Оголошуємо делегат
```

В даному випадку делегат визначається всередині класу, але також можна визначити делегат поза класом всередині простору імен.

Для використання делегата оголошується змінна цього делегата:

```
Message mes; // 2. Створюємо змінну делегата
```

За допомогою властивості DateTime.Now.Hour отримуємо поточну годину. І в залежності від часу в делегат передається адреса певного методу. Зверніть увагу, що методи ці мають той же повертаємий тип значення і той же набір параметрів (в даному випадку відсутність параметрів), що і делегат.

```
mes = GoodMorning; // 3. Надаємо цієї змінної адресу методу
```

Потім через делегат викликаємо метод, на який посилається цей делегат:

```
mes(); // 4. Викликаємо метод
```

Виклик делегата проводиться подібно викликом методу.

Подивимося на прикладі іншого делегата:

```
class Program
{
    delegate int Operation(int x, int y);

    static void Main(string[] args)
    {
        // присвоювання адреси методу через контруктор
        Operation del = Add; // делегат вказує на метод Add
        int result = del(4, 5); // фактично Add (4, 5)
        Console.WriteLine(result);

        del = Multiply; // тепер делегат вказує на метод Multiply
        result = del(4, 5); // фактично Multiply (4, 5)
        Console.WriteLine(result);

        Console.Read();
    }
    private static int Add(int x, int y)
    {
        return x + y;
    }
    private static int Multiply(int x, int y)
    {
        return x * y;
    }
}
```

В даному випадку делегат Operation повертає значення типу int і має два параметри типу int. Тому цьому делегату відповідає будь-який метод, який повертає значення типу int і приймає два параметри типу int. В даному випадку

це методи Add і Multiply. Тобто ми можемо привласнити змінної делегата будь-який з цих методів і викликати.

Оскільки делегат бере два параметра типу int, то при його виклику необхідно передати значення для цих параметрів: del (4,5).

Делегати необов'язково можуть вказувати тільки на методи, які визначені в тому ж класі, де визначена змінна делегата. Це можуть бути також методи з інших класів і структур.

```
class Math
{
    public int Sum(int x, int y) { return x + y; }
}
class Program
{
    delegate int Operation(int x, int y);

    static void Main(string[] args)
    {
        Math math = new Math();
        Operation del = math.Sum;
        int result = del(4, 5);     // math.Sum(4, 5)
        Console.WriteLine(result);  // 9
        Console.ReadKey();
    }
}
```

Присвоєння посилання на метод

Вище змінної делегата безпосередньо присвоювався метод. Є ще один спосіб – створення об'єкта делегата за допомогою конструктора, в який передається потрібний метод:

```
class Program
{
    delegate int Operation(int x, int y);

    static void Main(string[] args)
    {
        Operation del = Add;
        Operation del2 = new Operation(Add);

        Console.ReadKey();
    }
    private static int Add(int x, int y) { return x + y; }
}
```

Обидва способи рівноцінні.

Додавання методів в делегат

У прикладах вище змінна делегата вказувала на один метод. В реальності ж делегат може вказувати на безліч методів, які мають ту ж сигнатуру і повертаємий тип. Всі методи в делегаті потрапляють в спеціальний список –

список виклику або invocation list. І при виклику делегата все методи з цього списку послідовно викликаються. І ми можемо додавати в цей спіок не один, а кілька методів:

```
class Program
{
    delegate void Message();

    static void Main(string[] args)
{
     Message mes1 = Hello;
     mes1 += HowAreYou; // тепер mes1 вказує на два методу
     mes1 (); // викликаються обидва методи - Hello і HowAreYou
     Console.Read ();
}
    private static void Hello()
    {
        Console.WriteLine("Hello");
    }
    private static void HowAreYou()
    {
        Console.WriteLine("How are you?");
    }
}
```

В даному випадку в список виклику делегата mes1 додаються два метода – Hello і HowAreYou. І при виклику mes1 викликаються відразу обидва метода.

Для додавання делегатів застосовується операція +=. Однак варто зазначити, що в реальності буде відбуватися створення нового об'єкта делегата, який отримає методи старої копії делегата і новий метод, і новий створений об'єкт делеагата буде присвоєно змінної mes1.

При додаванні делегатів слід враховувати, що ми можемо додати посилання на один і той же метод кілька разів, і в списку виклику делегата тоді буде кілька посилань на один і той же метод. Відповідно при виклику делегата доданий метод буде викликатися стільки раз, скільки він був доданий:

```
Message mes1 = Hello;
mes1 += HowAreYou;
mes1 += Hello;
mes1 += Hello;
mes1();
```

Результат роботи програми:

Подібним чином ми можемо видаляти методи з делегата за допомогою операції –=:

```
static void Main(string[] args)
{
Message mes1 = Hello;
mes1 += HowAreYou;
Message mes2 = HowAreYou;
Message mes3 = mes1 + mes2;
mes1 (); // викликаються всі методи з mes1 і mes2
mes1 -= HowAreYou; // видаляємо метод HowAreYou
mes1 (); // викликається метод Hello
Console.ReadKey();
}
```

При видаленні методів з делегата фактично буде создаватся новий делегат, який в списку виклику методів буде містити на один метод менше.

При видаленні слід враховувати, що якщо делегат містить кілька посилань на один і той же метод, то операція –= починає пошук з кінця списку виклику делегата і видаляє тільки перше знайдене входження. Якщо подібного методу в списку виклику делегата немає, то операція –= не має ніякого ефекту.

Об'єднання делегатів

Делегати можна об'єднувати в інші делегати. Наприклад:

```
class Program
{
    delegate void Message();

    static void Main(string[] args)
    {
        Message mes1 = Hello;
        Message mes2 = HowAreYou;
        Message mes3 = mes1 + mes2; // об'єднуємо делегати
        mes3(); // викликаються всі методи з mes1 і mes2

        Console.ReadKey();
    }
    private static void Hello()
    {
        Console.WriteLine("Hello");
    }
    private static void HowAreYou()
    {
        Console.WriteLine("How are you?");
    }
}
```

У цьому випадку об'єкт mes3 представляє об'єднання делегатів mes1 і mes2. Об'єднання делегатів значить, що в список виклику делегата mes3 потраплять всі

методи з делегатів mes1 і mes2. І при виклику делегата mes3 всі ці методи одночасно будуть викликані.

Виклик делегата

У прикладах вище делегат викликався як звичайний метод. Якщо делегат брав параметри, то при його виклику для параметрів передавалися необхідні значення:

```
class Program
{
    delegate int Operation(int x, int y);
    delegate void Message();

    static void Main(string[] args)
    {
        Message mes = Hello;
        mes();
        Operation op = Add;
        op(3, 4);
        Console.ReadKey();
    }
    private static void Hello() { Console.WriteLine("Hello"); }
    private static int Add(int x, int y) { return x + y; }
}
```

Інший спосіб виклику делегата представляє метод **Invoke():**

```
class Program
{
    delegate int Operation(int x, int y);
    delegate void Message();

    static void Main(string[] args)
    {
        Message mes = Hello;
        mes.Invoke();
        Operation op = Add;
        op.Invoke(3, 4);
        Console.Read();
    }
    private static void Hello() { Console.WriteLine("Hello"); }
    private static int Add(int x, int y) { return x + y; }
}
```

Якщо делегат бере параметри, то в метод Invoke передаються значення для цих параметрів.

Слід враховувати, що якщо делегат порожній, тобто в його списку виклику немає посилань ні на один з методів (тобто делегат дорівнює null), то при виклику такого делегата ми отримаємо виняток, як, наприклад, в наступному випадку:

```
Message mes = null;
// mes (); //! Помилка: делегат дорівнює null
```

```
Operation op = Add;
op -= Add; // делегат op порожній
op (3, 4); //! Помилка: делегат дорівнює null
```

Тому при виклику делегата завжди краще перевіряти, чи не рівний він null. Або можна використовувати метод Invoke і оператор умовного null:

```
Message mes = null;
mes?.Invoke (); // помилки немає, делегат просто не викликається

Operation op = Add;
op -= Add; // делегат op порожній
op?.Invoke (3, 4); // помилки немає, делегат просто не викликається
```

Якщо делегат повертає деяке значення, то повертається значення останнього методу зі списку виклику (якщо в списку виклику кілька методів). Наприклад:

```
class Program
{
    delegate int Operation(int x, int y);

    static void Main(string[] args)
    {
        Operation op = Subtract;
        op += Multiply;
        op += Add;
        Console.WriteLine(op(7, 2));    // Add(7,2) = 9
        Console.Read();
    }
    private static int Add(int x, int y) { return x + y; }
    private static int Subtract(int x, int y) { return x - y; }
    private static int Multiply(int x, int y) { return x * y; }
}
```

Делегати як параметри методів

Також делегати можуть бути параметрами методів:

```
class Program
{
delegate void GetMessage();

static void Main(string[] args)
{
    if (DateTime.Now.Hour < 12)
    {
        Show_Message(GoodMorning);
    }
    else
    {
        Show_Message(GoodEvening);
```

```
        }
        Console.ReadLine();
    }
    private static void Show_Message(GetMessage _del)
{
    _del?.Invoke();
}
    private static void GoodMorning()
    {
        Console.WriteLine("Good Morning");
    }
    private static void GoodEvening()
    {
        Console.WriteLine("Good Evening");
    }
}
```

Узагальнені делегати

Делегати можуть бути узагальненими, наприклад:

```
delegate T Operation<T, K>(K val);

class Program
{
    static void Main(string[] args)
    {
        Operation<decimal, int> op = Square;

        Console.WriteLine(op(5));
        Console.Read();
    }

    static decimal Square(int n)
    {
        return n * n;
    }
}
```

12.2 Застосування делегатів

Найбільш сильна сторона делегатів полягає в тому, що вони дозволяють делегувати виконання деякого коду ззовні. І на момент написання програми ми можемо не знати, що за код буде виконуватися. Ми просто викликаємо делегат. А який метод буде безпосередньо виконуватися при виклику делегата, буде вирішуватися потім. Крім того, на основі делегатів можна створити функціонал методів зворотного виклику, повідомляючи інші об'єкти про події, що відбулися.

Розглянемо детальний приклад. Нехай у нас є клас, що описує рахунок в банку:

```
class Account
{
    int _sum; // Змінна для зберігання суми

    public Account(int sum)
    {
        _sum = sum;
    }

    public int CurrentSum
    {
        get { return _sum; }
    }

    public void Put(int sum)
    {
        _sum += sum;
    }

    public void Withdraw(int sum)
    {
        if (sum <= _sum)
        {
            _sum -= sum;
        }
    }
}
```

Припустимо, в разі виведення грошей за допомогою методу Withdraw нам треба якось повідомляти про це самого клієнта і, мабуть, інші об'єкти. Для цього створимо делегат AccountStateHandler. Щоб використовувати делегат, нам треба створити змінну цього делегата, а потім привласнити йому метод, який буде викликатися делегатом.

Отже, додамо в клас Account наступні рядки:

```
    // Оголошуємо делегат
    public delegate void AccountStateHandler(string message);
    // Створюємо змінну делегата
    AccountStateHandler _del;

    // Реєструємо делегат
    public void RegisterHandler(AccountStateHandler del)
    {
        _del = del;
    }

        // Далі інші рядки класу Account
```

Тут фактично робляться ті ж кроки, що були вище, і є практично всі крім виклику делегата. В даному випадку у нас делегат приймає параметр типу string. Тепер змінимо метод Withdraw наступним чином:

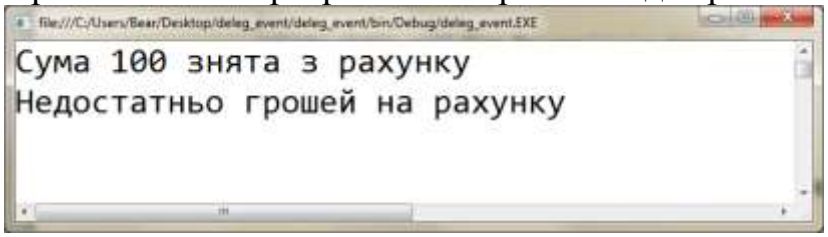
```
        public void Withdraw(int sum)
        {
            if (sum <= _sum)
            {
                _sum -= sum;

                if (_del != null)
                    _del($"Сума {sum} знята з рахунку");
            }
            else
            {
                if (_del != null)
                    _del("Недостатньо грошей на рахунку");
            }
        }
```

Тепер при знятті грошей через метод Withdraw ми спочатку перевіряємо, чи має делегат посилання на який-небудь метод (інакше він має значення null). І якщо метод встановлений, то викликаємо його, передаючи відповідне повідомлення в якості параметра.

Тепер протестуємо клас в основній програмі:

```
    class Program
        {
            static void Main(string[] args)
            {
      // створюємо банківський рахунок
      Account account = new Account(200);
      // Додаємо в делегат посилання на метод Show_Message
      // а сам делегат передається як параметр методу RegisterHandler
account.RegisterHandler(new Account.AccountStateHandler(Show_Message));
      // Два рази поспіль намагаємося зняти гроші
                account.Withdraw(100);
                account.Withdraw(150);
                Console.ReadLine();
            }
            private static void Show_Message(String message)
            {
                Console.WriteLine(message);
            }
        }
```

При виконанні програми ми отримаємо два різних повідомлення:



Таким чином, ми створили механізм зворотного виклику для класу Account, який спрацьовує в разі зняття грошей. Оскільки делегат оголошений всередині

класу Account, то щоб до нього отримати доступ, використовується вираз Account.AccountStateHandler.

Знову ж може виникнути питання: чому б у коді методу Withdraw() не виводити повідомлення про зняття грошей? Навіщо потрібно задіяти якийсь делегат?

Справа в тому, що не завжди у нас є доступ до коду класів. Наприклад, частина класів може створюватися і компілюватися однією людиною, який не знатиме, як ці класи будуть використовуватися. А використовувати ці класи буде інший розробник.

Так, тут ми виводимо повідомлення на консоль. Однак для класу Account не важливо, як це повідомлення виводиться. Класу Account навіть не відомо, що взагалі буде робитися в результаті списання грошей. Він просто посилає повідомлення про це через делегат.

В результаті, якщо ми створюємо консольне застосування, ми можемо через делегат виводити повідомлення на консоль. Якщо ми створюємо графічне застосування Windows Forms або WPF, то можна виводити повідомлення у вигляді графічного вікна. А можна не просто виводити повідомлення, а, наприклад, записати інформацію про цю дію в файл або відправити повідомлення на електронну пошту. Загалом будь-якими способами обробити виклик делегата. І спосіб обробки не буде залежати від класу Account.

Хоча в прикладі наш делегат брав адресу на один метод, в дійсності він може вказувати відразу на кілька методів. Крім того, при необхідності ми можемо видалити посилання на адреси певних методів, щоб вони не викликалися при виклику делегата. Отже, змінимо в класі Account метод RegisterHandler і додамо новий метод UnregisterHandler, який буде видаляти методи зі списку методів делегата:

```
// Реєструємо делегат
public void RegisterHandler(AccountStateHandler del)
{
    Delegate mainDel = System.Delegate.Combine(del, _del);
    _del = mainDel as AccountStateHandler;
}
// Скасування реєстрації делегата
public void UnregisterHandler(AccountStateHandler del)
{
    Delegate mainDel = System.Delegate.Remove(_del, del);
    _del = mainDel as AccountStateHandler;
}
```

У першому методі метод Combine об'єднує делегати _del і del в один, який потім присвоюється змінної _del. У другому методі метод Remove повертає делегат, зі списку викликів якого вилучено делегат del.

Далі текст всієї програми:

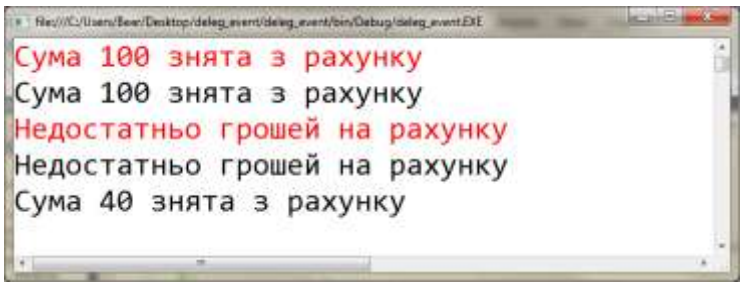
```
class Account
{
    // Оголошуємо делегат
    public delegate void AccountStateHandler(string message);
    // Створюємо змінну делегата
    AccountStateHandler _del;
    // Реєструємо делегат
    public void RegisterHandler(AccountStateHandler del)
    {
        Delegate mainDel = System.Delegate.Combine(del, _del);
        _del = mainDel as AccountStateHandler;
    }
    // Скасування реєстрації делегата
    public void UnregisterHandler(AccountStateHandler del)
    {
        Delegate mainDel = System.Delegate.Remove(_del, del);
        _del = mainDel as AccountStateHandler;
    }
    int _sum; // Змінна для зберігання суми

    public Account(int sum)
    {
        _sum = sum;
    }

    public int CurrentSum
    {
        get { return _sum; }
    }

    public void Put(int sum)
    {
        _sum += sum;
    }

    public void Withdraw(int sum)
    {
        if (sum <= _sum)
        {
            _sum -= sum;

            if (_del != null)
                _del($"Сума {sum} знята з рахунку");
        }
        else
        {
            if (_del != null)
                _del("Недостатньо грошей на рахунку");
        }
    }
}
```

```
class Program
    {
        static void Main(string[] args)
        {
            Account account = new Account(200);
            Account.AccountStateHandler colorDelegate = new
Account.AccountStateHandler(Color_Message);

            // Додаємо в делегат посилання на методи
            account.RegisterHandler(new
Account.AccountStateHandler(Show_Message));
            account.RegisterHandler(colorDelegate);
            // Два рази поспіль намагаємося зняти гроші
            account.Withdraw(100);
            account.Withdraw(150);

            // Видаляємо делегат
            account.UnregisterHandler(colorDelegate);
            account.Withdraw(40);

            Console.ReadLine();
        }
        private static void Show_Message(String message)
        {
            Console.WriteLine(message);
        }
        private static void Color_Message(string message)
        {
            // Встановлюємо червоний колір символів
            Console.ForegroundColor = ConsoleColor.Red;
            Console.WriteLine(message);
            // Скидаємо настройки кольору
            Console.ResetColor();
        }
    }
```

З метою тестування ми створили ще один метод – Color_Message, який виводить те ж саме повідомлення тільки червоним кольором. Для першого делегата створюється окрема змінна. Але великої різниці між передачею обох в метод account.RegisterHandler немає: просто в одному випадку ми відразу передаємо об'єкт, створюваний конструктором account.RegisterHandler (new Account.AccountStateHandler (Show_Message));

У другому випадку створюємо змінну і її вже передаємо в метод account.RegisterHandler (colorDelegate) ;.

У рядку account.UnregisterHandler (colorDelegate); цей метод видаляється зі списку дзвінків делегата, тому цей метод більше не буде спрацьовувати. Вивід на консоль буде мати наступну форму:



Також ми можемо використовувати скорочену форму додавання та видалення делегатів. Для цього перепишемо методи RegisterHandler і UnregisterHandler наступним чином:

```
// Реєструємо делегат
  public void RegisterHandler(AccountStateHandler del)
  {
      _del += del; // додаємо делегат
  }
  // Скасування реєстрації делегата
  public void UnregisterHandler(AccountStateHandler del)
  {
      _del -= del; // видаляємо делегат
  }
```

12.3 Події

Події засновані на делегатах. Події сигналізують системі про те, що сталася певна дія.

Події оголошуються в класі за допомогою ключового слова event, після якого йде назва делегата:

```
// Оголошуємо делегат
public delegate void AccountStateHandler(string message);
// Подія, що виникає при виведенні грошей
public event AccountStateHandler Withdrawn;
```

Будь-який об'єкт, що має доступ до делегату, може посилатися на такий делегат, проте тільки тип, який визначає подію, може викликати цю подію.

Делегат повинен бути, принаймні, рівня доступу події (як правило, визначаються рівня доступу public). Делегат не повинен повертати значення.

Коли клас визначає подію, інші класи можуть визначити метод, відповідний сигнатурі делегата, пов'язаного з подією, і підписатися на подію, вказавши, що їх метод повинен бути запущений при виникненні події.

Але! Події – це не екземпляри делегатів.

Делегат може існувати на рівні простору імен або класу, подія може визначатися тільки на рівні класу.

Події можна визначити в інтерфейсі, делегат – ні.

Використання події

Після того як подія визначена в класі, її можна використовувати в застосуванні, для цього на подію потрібно підписатися в класах, які використовують цю подію і генерувати подію в класі, де вона визначена.

Для підписки на подію використовується складова операція присвоювання «+ =»:

```
Withdrawn += new AccountStateHandler(HandlingMethod);
```

Відмовитися від підписки на подію можна за допомогою складовою операції присвоювання «– =»:

```
Withdrawn -= HandlingMethod;
```

При виклику події необхідно спочатку перевірити, що вона не є нульовою:

```
if (MyEvent != null)
   {
       MyEvent(str);
   }
```

Отже, розглянемо використання події на прикладі. Для цього візьмемо клас Account з попередньої теми і змінимо його наступним чином:

```
class Account
{
    // Оголошуємо делегат
    public delegate void AccountStateHandler(string message);
    // Подія, що виникає при виведенні грошей
    public event AccountStateHandler Withdrawn;
    // Подія, що виникає при додавання на рахунок
    public event AccountStateHandler Added;

    int _sum; // Змінна для зберігання суми

    public Account(int sum)
    {
        _sum = sum;
    }

    public int CurrentSum
    {
        get { return _sum; }
    }

    public void Put(int sum)
    {
        _sum += sum;
        if (Added != null)
            Added($"На рахунок надійшло {sum}");
    }
    public void Withdraw(int sum)
    {
        if (sum <= _sum)
```

```
            {
                _sum -= sum;
                if (Withdrawn != null)
                    Withdrawn($"Сума {sum} знята з рахунку");
            }
            else
            {
                if (Withdrawn != null)
                    Withdrawn("Недостатньо грошей на рахунку");
            }
        }
    }
```

Тут ми визначили дві події: Withdrawn і Added. Обидві події оголошені як екземпляри делегата AccountStateHandler, тому для обробки цих подій буде потрібен метод, який приймає рядок в якості параметра.
Потім в методах Put і Withdraw ми викликаємо ці події. Перед викликом ми перевіряємо, чи закріплені за цими подіями обробники (`if(Withdrawn != null)`. Ці події представляють делегат AccountStateHandler, що приймає в якості параметра рядок, тому і при виклику подій ми передаємо в них рядок.

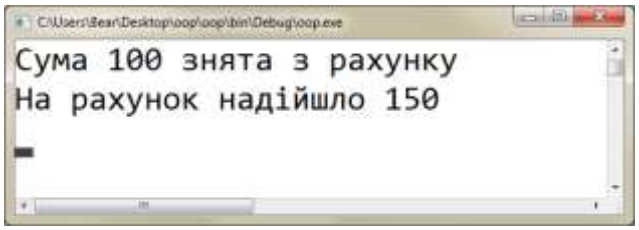
Тепер використовуємо події в основній програмі:

```
    class Program
   {
     static void Main (string [] args)
     {
         Account account = new Account (200);
         // Додаємо обробники події
         account.Added += Show_Message;
         account.Withdrawn += Show_Message;

         account.Withdraw (100);
         // Видаляємо обробник події
         account.Withdrawn -= Show_Message;

         account.Withdraw (50);
         account.Put (150);

         Console.ReadLine ();
     }
     private static void Show_Message (string message)
     {
         Console.WriteLine (message);
     }
  }
```

Для прикріплення обробника події до певної події використовується операція += і відповідно для відкріплення – операція –=:

```
account.Added += Show_Message;
account.Added -= Show_Message;
```

Знову ж звертаємо увагу, що метод обробника повинен мати такі ж параметри, як і делегат події, і повертати той же тип. У підсумку ми отримаємо наступний вивід на консоль:



Крім використаного вище способу прикріплення обробників є й інший, з використанням делегата. Але обидва способи будуть рівноцінні:

```
account.Added += Show_Message;
account.Added += new Account.AccountStateHandler(Show_Message);
```

Клас даних події AccountEventArgs

Нерідко при виникненні події обробнику події потрібно передати деяку інформацію про подію. Наприклад, додамо і в нашу програму новий клас AccountEventArgs з наступним кодом:

```
class AccountEventArgs
{
    // Повідомлення
    public string Message { get; set; }
    // Сума, на яку змінився рахунок
    public int Sum { get; set; }

    public AccountEventArgs(string mes, int sum)
    {
        Message = mes;
        Sum = sum;
    }
}
```

Цей клас має дві властивості: Message – для зберігання повідомлення, що виводиться, і Sum – для зберігання суми, на яку змінився рахунок.

Тепер застосуємо клас AccoutEventArgs, змінивши клас Account наступним чином:

```
class Account
 {
     // Оголошуємо делегат
     public delegate void AccountStateHandler(object sender,
AccountEventArgs e);
     // Подія, що виникає при виведенні грошей
     public event AccountStateHandler Withdrawn;
     // Подія, що виникає при додаванні на рахунок
     public event AccountStateHandler Added;
```

```
            int _sum; // Змінна для зберігання суми

            public Account(int sum)
            {
                _sum = sum;
            }

            public int CurrentSum
            {
                get { return _sum; }
            }

            public void Put(int sum)
            {
                _sum += sum;
  if (Added != null)
     Added(this, new AccountEventArgs($"На рахунок надійшло {sum}", sum));
            }
       public void Withdraw(int sum)
        {
           if (_sum >= sum)
           {
             _sum -= sum;
              if (Withdrawn != null)
                 Withdrawn(this, new AccountEventArgs($"Сума {sum} знята з
рахунку", sum));
            }
           else
                {
                    if (Withdrawn != null)
                Withdrawn(this, new AccountEventArgs("Недостатньо грошей
на рахунку", sum));
                }
            }
        }
```

У порівнянні з попередньою версією класу Account тут змінилася тільки кількість параметрів у делегата і відповідно кількість параметрів при виклику події. Тепер вони також приймають об'єкт AccountEventArgs, який зберігає інформацію про подію, що отримується через конструктор.

Тепер змінимо основну програму:

```
class Program
 {
    static void Main(string[] args)
    {
        Account account = new Account(200);
        // Додаємо обробники події
        account.Added += Show_Message;
        account.Withdrawn += Show_Message;
        account.Withdraw(100);
        // Видаляємо обробник події
        account.Withdrawn -= Show_Message;
```

```
            account.Withdraw(50);
            account.Put(150);
            Console.ReadLine();
        }

        private static void Show_Message(object sender, AccountEventArgs e)
        {
            Console.WriteLine($"Сума транзакції: {e.Sum}");
            Console.WriteLine(e.Message);
        }
    }
```

У порівнянні з попереднім варіантом тут ми тільки змінюємо кількість параметрів і сутність їх використання в обробнику Show_Message.

12.4 Анонімні методи

З делегатами тісно пов'язані анонімні методи. Анонімні методи використовуються для створення екземплярів делегатів.

Визначення анонімних методів починається з ключового слова delegate, після якого йде в дужках список параметрів і тіло методу в фігурних дужках:

```
delegate (параметри)
    {
        // інструкції
    };
```

наприклад:

```
class Program
{
    delegate void MessageHandler(string message);
    static void Main(string[] args)
    {
        MessageHandler handler = delegate (string mes)
        {
            Console.WriteLine(mes);
        }; //   крапка з комою після } дужки.

        handler("hello world!");

        Console.Read();
    }
}
```

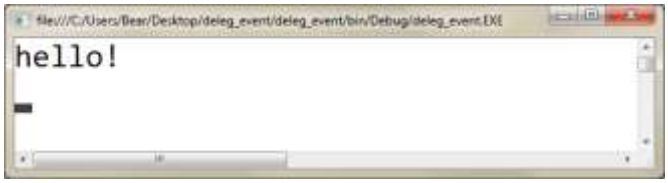
Анонімний метод не може існувати сам по собі, він використовується для ініціалізації екземпляра делегата, як в даному випадку змінна handler представляє анонімний метод. І через цю змінну делегата можна викликати даний анонімний метод.

І важливо відзначити, що на відміну від блоку методів або умовних і циклічних конструкцій, блок анонімних методів повинен закінчуватися крапкою з комою після фігурної дужки.

Інший приклад анонімних методів – передача в якості аргументу параметра, який представляє делегат:

```
class Program
{
    delegate void MessageHandler(string message);
    static void Main(string[] args)
    {
        ShowMessage // виклик методу ShowMessage
        (
           "hello!",  // 1-й параметр
    delegate (string mes)  // 2-й параметр - делегат (анонімний метод)
           {
               Console.WriteLine(mes);
           }
        ); // а тут  крапка з комою після ) дужки.

        Console.ReadKey();
    }
    static void ShowMessage(string mes, MessageHandler handler)
    {
        handler(mes);
    }
}
```

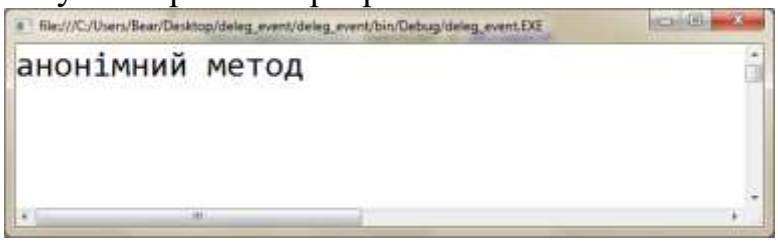
Результат роботи програми:



Якщо анонімний метод використовує параметри, то вони повинні відповідати параметрам делегата. Якщо для анонімного методу не потрібно параметрів, то дужки з параметрами опускаються. При цьому навіть якщо делегат бере кілька параметрів, то в анонімному методі можна зовсім опустити параметри:

```
class Program
{
    delegate void MessageHandler(string message);
    static void Main(string[] args)
    {
        MessageHandler handler = delegate
        {
            Console.WriteLine("анонімний метод");
        };
        handler("hello world!");    // анонимный метод

        Console.Read();
    }
}
```

Результат роботи програми:



Тобто якщо анонімний метод містить параметри, вони обов'язково повинні відповідати параметрам делегата. Або анонімний метод взагалі може не містити ніяких параметрів, тоді він відповідає будь-якому делегату, який має той же тип значення, ще повертється.

При цьому параметри анонімного методу не можуть бути опущені, якщо один або декілька параметрів визначені з модифікатором out.

Також, як і звичайні методи, анонімні можуть повертати результат:

```
class Program
{
    delegate int Operation(int x, int y);
    static void Main(string[] args)
    {
        Operation operation = delegate (int x, int y)
        {
            return x + y;
        };

        int d = operation(4, 5);
        Console.WriteLine(d);       //Вивід 9
        Console.Read();
    }
}
```

При цьому анонімний метод має доступ до всіх змінних, яки визначені в зовнішньому коді:

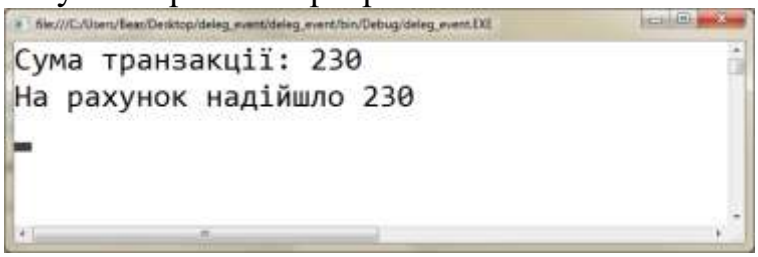
```
class Program
{
    delegate int Operation(int x, int y);
    static void Main(string[] args)
    {
        int z = 8;
        Operation operation = delegate (int x, int y)
        {
            return x + y + z;
        };
        int d = operation(4, 5);
        Console.WriteLine(d);       // 17
        Console.Read();
    }
}
```

В яких ситуаціях використовуються анонімні методи? Коли нам треба визначити одноразову дію, яка не має багато інструкцій і ніде більше не використовується. Іноді такі методи потрібні для обробки однієї події і більше цінності не представляють і ніде не застосовуються. Наприклад, можна застосувати анонімні методи для обробки подій:

```
   delegate void AccountStateHandler(object sender, AccountEventArgs e);
    class AccountEventArgs
    {
        public string Message { get; }
        public int Sum { get; }
        public AccountEventArgs(string message, int sum)
        {
            Message = message; Sum = sum;
        }
    }

    class Account
    {
        int _sum;
        public event AccountStateHandler Added;
        public event AccountStateHandler Withdrawn;
        public Account(int sum)
        {
            _sum = sum;
        }
        public void Put(int sum)
        {
            _sum += sum;
            if (Added != null) Added(this,
                new AccountEventArgs($"На рахунок надійшло {sum}", sum));
        }

        public void Withdraw(int sum)
        {
            if (_sum >= sum)
            {
                _sum -= sum;
                if (Withdrawn != null)
Withdrawn(this, new AccountEventArgs($"З рахунку знято {sum}", sum));
            }
            else
            {
              if (Withdrawn != null)
Withdrawn(this, new AccountEventArgs("На рахунку недостатньо коштів", 0));
            }
        }
    }
```

```
class Program
{
    static void Main(string[] args)
    {
        Account account = new Account(200);
        // Додаємо обробники події
        account.Added += delegate (object sender, AccountEventArgs e)
        {
            Console.WriteLine($"Сума транзакції: {e.Sum}");
            Console.WriteLine(e.Message);
        };
        account.Put(230);

        Console.Read();
    }
}
```

Результат роботи програми:



12.5 Лямбда-вирази

Лямбда-вирази являють собою спрощену запис анонімних методів. Лямбда-вирази дозволяють створити ємкі лаконічні методи, які можуть повертати деяке значення і які можна передати в якості параметрів в інші методи.

Лямбда-вирази мають наступний синтаксис: зліва від лямбда-оператора => визначається список параметрів, а праворуч блок виразів, що використовує ці параметри: (спісок_параметров) => вираз. наприклад:

```
class Program
{
    delegate int Operation(int x, int y);
    static void Main(string[] args)
    {
        Operation operation = (x, y) => x + y;
        Console.WriteLine(operation(10, 20));       // 30
        Console.WriteLine(operation(40, 20));       // 60
        Console.Read();
    }
}
```

Тут код (x, y) => x + y; представляє лямбда-вираз, де x і y - це параметри, а x + y – вираз. При цьому нам не треба вказувати тип параметрів, а при поверненні результату не треба використовувати оператор return.

При цьому треба враховувати, що кожен параметр в лямбда-виразі неявно перетворюється у відповідний параметр делегата, тому типи параметрів повинні бути однаковими. Крім того, кількість параметрів повинна бути такою же, як і у делегата. І значення, яке повертає лямбда-вираз, має бути тим же, що і у делегата. Тобто в даному випадку лямбда-вираз відповідає делегату Operation як за типом значення, що повертається, так і за типом і кількістю параметрів.

Якщо лямбда-вираз приймає один параметр, то дужки навколо параметра можна опустити:

```
    class Program
    {
        delegate int Square(int x); // оголошуємо делегат, що приймає int
і повертає int
        static void Main(string[] args)
        {
            Square square = i => i * i; // об'єкту делегата присвоюється
лямбда-вираз

            int z = square(6); // використовуємо делегат
            Console.WriteLine(z); // виводить число 36
            Console.Read();
        }
    }
```

Буває, що параметри не потрібно. У цьому випадку замість параметра в лямбда-виразі використовуються порожні дужки. Також буває, що лямбда-вираз не повертає ніякого значення:

```
    class Program
    {
        delegate void Hello(); // делегат без параметрів
        static void Main(string[] args)
        {
            Hello hello1 = () => Console.WriteLine("Hello");
            Hello hello2 = () => Console.WriteLine("Welcome");
            hello1();       // Hello
            hello2();       // Welcome
            Console.Read();
        }
    }
```

В даному випадку лямда-вираз нічого не повертає, тому що після лямбда-оператора йде дія, яке нічого не повертає.

Як видно, з прикладів вище, нам необов'язково вказувати тип параметрів у лямбда-вирази. Однак, нам обов'язково потрібно вказувати тип, якщо делегат, якому має відповідати лямбда-вираз, має параметри з модифікаторами **ref** і **out**:

```
    class Program
    {
        delegate void ChangeHandler(ref int x);
        static void Main(string[] args)
```

```
    {
        int x = 9;
        ChangeHandler ch = (ref int n) => n = n * 2;
        ch(ref x);
        Console.WriteLine(x);   // 18
        Console.Read();
    }
}
```

Лямбда-вирази також можуть виконувати інші методи:

```
class Program
{
    delegate void Hello(); // делегат без параметрів
    static void Main(string[] args)
    {
        Hello message = () => Show_Message();
        message();
    }
    private static void Show_Message()
    {
        Console.WriteLine("Привіт світ!");
    }
}
```

Лямбда-вирази в обробці подій

Одним з частих прикладів використання лямбда-виразів є обробка подій. Візьмемо клас Account з попередньої теми:

```
delegate void AccountStateHandler(object sender, AccountEventArgs e);

class AccountEventArgs
{
    public string Message { get; }
    public int Sum { get; }
    public AccountEventArgs(string message, int sum)
    {
        Message = message; Sum = sum;
    }
}
class Account
{
    int _sum;
    public event AccountStateHandler Added;
    public event AccountStateHandler Withdrawn;
    public Account(int sum)
    {
        _sum = sum;
    }
    public void Put(int sum)
    {
        _sum += sum;
```

```
            if (Added != null) Added(this,
            new AccountEventArgs($"На рахунок прийшло {sum}", sum));
        }
        public void Withdraw(int sum)
        {
            if (_sum >= sum)
            {
                _sum -= sum;
                if (Withdrawn != null)
                     Withdrawn(this, new AccountEventArgs($"З рахунку
знято {sum}", sum));
            }
            else
            {
              if (Withdrawn != null)
                Withdrawn(this,  new AccountEventArgs("На рахунку
недостатньо коштів", 0));
            }
        }
    }
```

І перепишемо прикріплення обробника події за допомогою лямбда-виразу:

```
    class Program
    {
        static void Main(string[] args)
        {
            Account account = new Account(100);
            account.Added += (sender, e) =>
            {
                Console.WriteLine($"Сума транзакції: {e.Sum}");
                Console.WriteLine(e.Message);
            };
            account.Put(200);
            account.Put(109);
            Console.ReadKey();
        }
    }
```

Оскільки тут використовується кілька параметрів, то вони укладаються в дужки. І тому що в тілі лямбда-виразу застосовується кілька виразів, то вони укладаються в блок з фігурних дужок.

Лямбда-вирази як аргументи методів

Як і делегати, лямбда-вирази можна передавати в якості аргументів методу для тих параметрів, які представляють делегат, що досить зручно:

```
    class Program
    {
        delegate bool IsEqual(int x);
```

```
        static void Main(string[] args)
        {
            int[] integers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

            // знайдемо суму чисел більше 5
            int result1 = Sum(integers, x => x > 5);
            Console.WriteLine(result1); // 30

            // знайдемо суму парних чисел
            int result2 = Sum(integers, x => x % 2 == 0);
            Console.WriteLine(result2);  //20

            Console.Read();
        }

        private static int Sum(int[] numbers, IsEqual func)
        {
            int result = 0;
            foreach (int i in numbers)
            {
                if (func(i))
                    result += i;
            }
            return result;
        }
    }
```

Метод Sum приймає як параметр масив чисел і делегат IsEqual і повертає суму чисел масиву у вигляді об'єкта int. У циклі проходимо по всім числам і складаємо їх. Причому складаємо тільки ті числа, для яких делегат IsEqual func повертає true. Тобто делегат IsEqual тут фактично задає умову, якої повинні відповідати значення масиву. Але на момент написання методу Sum нам невідомо, що це за умова.

При виклику методу Sum йому передається масив і лямбда-вираз:

```
int result1 = Sum(integers, x => x > 5);
```

Тобто параметр x тут буде представляти число, яке передається у делегат:

```
            if (func(i))
                    result += i;
```

А вираз x> 5 представляє умову, якої повинно відповідати число. Якщо число відповідає цієй умові, то лямбда-вираз повертає true, а передане число складається з іншими числами.

Подібним чином працює другий виклик методу Sum, тільки тут вже йде перевірка числа на парність, тобто якщо залишок від ділення на 2 дорівнює нулю:

```
int result2 = Sum(integers, x => x % 2 == 0);
```

12.6 Коваріантність і контраваріантність делегатів

Делегати можуть бути *коваріантними* і *контраваріантними*. Коваріантність делегата передбачає, що типом, який повертається, може бути більш похідний тип. Контрваріантність делегата передбачає, що типом параметра може бути більш універсальний тип.

Коваріантність

Коваріантність дозволяє повертати з методу об'єкт, тип якого є похідним від типу, що повертається делегатом.

Припустимо, є наступна структура класів:

```
class Person
{
    public string Name { get; set; }
}
    class Client : Person { }
```

В цьому випадку коваріантність делегата може виглядати наступним чином:

```
delegate Person PersonFactory(string name);
static void Main(string[] args)
{
    PersonFactory personDel;
    personDel = BuildClient; // коваріантність
    Person p = personDel("Tom");
    Console.WriteLine(p.Name);
    Console.Read();
}
private static Client BuildClient(string name)
{
    return new Client { Name = name };
}
```

Тобто тут делегат повертає об'єкт Person. Однак завдяки коваріантності даний делегат може вказувати на метод, який повертає об'єкт похідного типу, наприклад, Client.

Контрваріантність

Контрваріантність передбачає можливість передавати в метод об'єкт, тип якого є більш універсальним по відношенню до типу параметра делегата. Наприклад, візьмемо визначені раніше класи Person і Client і використовуємо їх в наступному прикладі:

```
delegate void ClientInfo(Client client);
static void Main(string[] args)
{
```

```
        ClientInfo clientInfo = GetPersonInfo; // контраваріантність
        Client client = new Client { Name = "Alice" };
        clientInfo(client);
        Console.Read();
    }
    private static void GetPersonInfo(Person p)
    {
        Console.WriteLine(p.Name);
    }
```

Незважаючи на те, що делегат в якості параметра приймає об'єкт Client, йому можна присвоїти метод, який приймає як параметр об'єкт базового типу Person. Може здатися на перший погляд, що тут є певна суперечність, тобто використання більш універсального тип замість більш похідного. Однак в реальності в делегат при його виклику ми все одно можемо передати тільки об'єкти типу Client, а будь-який об'єкт типу Client є об'єктом типу Person, який використовується в методі.

Коваріантність і контраваріантність в узагальнених делегатах

Узагальнені делегати також можуть бути коваріантними і контраваріантними, що дає нам більше гнучкості в їх використанні.

Наприклад, візьмемо таку ієрархію класів:

```
class Person
{
public string Name { get; set; }
public virtual void Display() => Console.WriteLine($"Person {Name}");
}
class Client : Person
{
public override void Display() => Console.WriteLine($"Client {Name}");
}
```

Тепер оголосимо і використовуємо коваріантний узагальнений делегат:

```
class Program
{
    delegate T Builder<out T>(string name);
    static void Main(string[] args)
    {
        Builder<Client> clientBuilder = GetClient;
Builder<Person> personBuilder1 = clientBuilder;     // коваріантність
Builder<Person> personBuilder2 = GetClient;         // коваріантність

        Person p = personBuilder1("Tom"); // виклик делегата
        p.Display(); // Client: Tom

        Console.Read();
    }
    private static Person GetPerson(string name)
    {
        return new Person { Name = name };
```

```
    }
    private static Client GetClient(string name)
    {
        return new Client { Name = name };
    }
}
```

Завдяки використанню out ми можемо присвоїти делегату типу Builder <Person> делегат типу Builder <Client> або посилання на метод, який повертає значення Client.

Розглянемо контраваріантний узагальнений делегат:

```
class Program
{
    delegate void GetInfo<in T>(T item);
    static void Main(string[] args)
    {
        GetInfo<Person> personInfo = PersonInfo;
      GetInfo<Client> clientInfo = personInfo;  // контраваріантність

        Client client = new Client { Name = "Tom" };
        clientInfo(client); // Client: Tom

        Console.Read();
    }
    private static void PersonInfo(Person p) => p.Display();
    private static void ClientInfo(Client cl) => cl.Display();
}
```

Використання ключового слова **in** дозволяє привласнити делегат з більш універсальним типом (GetInfo <Person>) делегату з похідним типом (GetInfo <Client>).

Як і у випадку з узагальненими інтерфейсами параметр коваріантного типу застосовується тільки до типу значення, які повертається делегатом. А параметр контраваріантного типу застосовується тільки до вхідних аргументів делегата.

12.7 Делегати Action, Predicate і Func

В .NET є кілька вбудованих делегатів, які використовуються в різних ситуаціях. І найбільш використовуваними, з якими часто доводиться стикатися, є Action, Predicate і Func.

Action

Делегат Action є узагальненим, приймає параметри і повертає значення void:

```
public delegate void Action<T>(T obj);
```

Даний делегат має ряд перевантажених версій. Кожна версія приймає різну кількість параметрів: від Action <in T1> до Action <in T1, in T2, ... in T16>. Таким чином можна передати до 16 значень в метод.

Як правило, цей делегат передається як параметр методу і передбачає виклик певних дій у відповідь на що відбулися дії. наприклад:

```
static void Main(string[] args)
{
    Action<int, int> op;
    op = Add;
    Operation(10, 6, op);
    op = Substract;
    Operation(10, 6, op);

    Console.Read();
}

static void Operation(int x1, int x2, Action<int, int> op)
{
    if (x1 > x2)
        op(x1, x2);
}

static void Add(int x1, int x2)
{
    Console.WriteLine("Сума чисел: " + (x1 + x2));
}

static void Substract(int x1, int x2)
{
    Console.WriteLine("Різниця чисел: " + (x1 - x2));
}
```

Predicate

Делегат Predicate <T>, як правило, використовується для порівняння, співставлення деякого об'єкта T певній умові. В якості вихідного результату повертається значення true, якщо умова дотримана, і false, якщо не дотримано:

```
Predicate<int> isPositive = delegate (int x) { return x > 0; };

Console.WriteLine(isPositive(20));
Console.WriteLine(isPositive(-20));
```

В даному випадку повертається true або false залежно від того, більше нуля число чи ні.

Func

Ще одним поширеним делегатом є Func. Він повертає результат дії і може приймати параметри. Він також має різні форми: від Func <out T> (), де T - тип значення, що повертається, до Func <in T1, in T2, ... in T16, out TResult> (), тобто може приймати до 16 параметрів.

```
TResult Func<out TResult > ()
```

```
TResult Func<in T, out TResult > (T arg)
TResult Func<in T1, in T2, out TResult > (T1 arg1, T2 arg2)
TResult Func<in T1, in T2, in T3, out TResult > (T1 arg1, T2 arg2, T3 arg3)

//.....................................
```

Цей делегат також часто використовується в якості параметра в методах:

```
static void Main(string[] args)
{
    Func<int, int> retFunc = Factorial;
    int n1 = GetInt(6, retFunc);
    Console.WriteLine(n1);  // 720

    int n2 = GetInt(6, x => x * x);
    Console.WriteLine(n2); // 36

    Console.Read();
}

static int GetInt(int x1, Func<int, int> retF)
{
    int result = 0;
    if (x1 > 0)
        result = retF(x1);
    return result;
}
static int Factorial(int x)
{
    int result = 1;
    for (int i = 1; i <= x; i++)
    {
        result *= i;
    }
    return result;
}
```

Метод GetInt () в якості параметра приймає делегат Func <int, int>, тобто посилання на метод, який приймає число int і повертає також значення int.

При першому виклику методу GetInt () йому передається посилання на метод обчислення факторіала. У другому випадку передається лямбда-вираз x => x * x, тобто знову ж вираз, яке приймає параметр int x і повертає результат x * x.

Резюме

1. Делегат – це об'єкт, який може посилатися на метод.
2. Для оголошення делегата використовується ключове слово delegate, після якого йде повертаємий тип, назва і параметри.
3. Делегат може вказувати на безліч методів, які мають ту ж сигнатуру і повертаємий тип.

4. Всі методи в делегаті потрапляють в спеціальний список – список виклику або invocation list, і при виклику делегата все методи з цього списку послідовно викликаються.
5. При виклику делегата завжди краще перевіряти, чи не рівний він null.
6. Делегати можуть бути параметрами методів.
7. Делегати можуть бути узагальненими.
8. Події засновані на делегатах. Події оголошуються в класі за допомогою ключового слова event, після якого йде назва делегата.
9. Будь-який об'єкт, що має доступ до делегату, може посилатися на такий делегат, проте тільки тип, який визначає подію, може викликати цю подію.
10. Після того як подія визначена в класі, її можна використовувати в застосуванні, для цього на подію потрібно підписатися в класах, які використовують цю подію і генерувати подію в класі, де вона визначена.
11. Для підписки на подію використовується складова операція присвоювання «+ =». Відмовитися від підписки на подію можна за допомогою складовою операції присвоювання «– =».
12. При виклику події необхідно спочатку перевірити, що вона не є нульовою.
13. Анонімні методи використовуються для створення екземплярів делегатів.
14. Якщо анонімний метод використовує параметри, то вони повинні відповідати параметрам делегата. Якщо для анонімного методу не потрібно параметрів, то дужки з параметрами опускаються.
15. Лямбда-вирази дозволяють створити ємкі лаконічні методи, які можуть повертати деяке значення і які можна передати в якості параметрів в інші методи.
16. Делегати можуть бути коваріантними і контраваріантними. Коваріантність делегата передбачає, що типом, який повертається, може бути більш похідний тип. Контрваріантність делегата передбачає, що типом параметра може бути більш універсальний тип.

Завдання до виконання

1. Методом трапеції обчислити інтеграл $\int_a^b f(x)dx$. Реалізувати пошук інтеграла у вигляді метода, одним із параметрів якого є делегат, який посилається на функцію, для якої розв’яується задача. Функції, для яких треба реалізувати пошук інтеграла, перелічені у варіантах завдань.
2. Створити подію, яка пов’язана з натисканням на клавіатурі символа, з которого починається ваше ім’я. Метод-обробник цієї події виводить ваше ім’я повністю.

Варіанти завдань:

1. $f(x)=\frac{1}{x}$, $f(x)=\sqrt{4+x^2}$, $f(x)=\sin(x)$
2. $f(x)=\frac{1}{\sqrt[3]{x}}$, $f(x)=\frac{1}{\sqrt{x^2}}$, $f(x)=\cos(x)$
3. $f(x)=\frac{\cos(x)}{x}$, $f(x)=\frac{\sin(x)}{\sqrt{x^2}}$, $f(x)=\sqrt[3]{x}$
4. $f(x)=\frac{1}{x^2}$, $f(x)=\frac{1}{x^2+1}$, $f(x)= x^3+x^2+x+1$
5. $f(x)=\frac{1}{\sqrt{|x|}}$, $f(x)=\frac{1}{x^2+1}$, $f(x)=\ln(x)$
6. $f(x)=\frac{1}{\sqrt[3]{x}}$, $f(x)=\frac{1}{\sqrt{x^2}}$, $f(x)=\sin^2(x)$
7. $f(x)=\frac{1}{\sqrt[3]{|\cos(x)|}}$, $f(x)=\frac{1}{\sqrt{x^2}}$, $f(x)=\cos^2(x)$
8. $f(x)=\frac{1}{x^3}$, $f(x)=\frac{1}{\sqrt{x}}$, $f(x)=e^{-x}+1$
9. $f(x)=\frac{1}{e^x}$, $f(x)=\frac{1}{\sqrt{x^2}}$, $f(x)=\frac{1}{x}\cos(x)$
10. $f(x)=\frac{1}{x^3+x^2+x+1}$, $f(x)=\frac{\ln(x)}{\sqrt{x^2}}$, $f(x)=\cos(x)$
11. $f(x)=\frac{1}{\sqrt[3]{x}}$, $f(x)=\frac{e^x}{\sqrt{x}}$, $f(x)=\log_2(x)$
12. $f(x)=\frac{1}{\log_2(x)}$, $f(x)=\frac{1}{\sqrt{x^2}}$, $f(x)=\frac{1}{e^x}\cos(x)$

Тести для закріплення матеріалу

1. Що є класом-делегатом для представлення методу обробки події натискання кнопки миші?

а) Click.
б) EventHandler.
в) MouseEventArgs.
г) object sender.

2. Яку подію треба пов'язати з кнопкою, щоб при натисканні на кнопку відбувалися будь-які дії?
   а) Click;
   б) mrButton;
   в) mrButton.text;
   г) EventHandler.

3. Що з нижчепереліченого є подією натискання клавіші?
   а) KeyPress;
   б) EventHandler;
   в) KeyPressEventArgs
   г) KeyPressEventHandler.

4. В якому місці в програмі може бути визначений делегат?
   а) всередині методів
   б) тільки як член класу або структури.
   в) в будь-якому місці, де може бути визначено клас
   г) тільки як член інтерфейсу

5. З якої причини даний код не скомпілюється?

```
public delegate void EventHandler();
public class Control
{
    public event EventHandler Invalidate;
}
public class Button : Control
{
    public void Draw()
    {
        if (Invalidate != null)
            Invalidate();
    }
}
```

   а) викликати події безпосередньо може тільки клас, в якому вони оголошені;
   б) не можна порівнювати подію з null;
   в) подія завжди дорівнює null;
   г) у події відсутні обов'язкові частини add {} і remove {}.

6. Є наступний делегат:

```
delegate int Operation(int val);
```

Які з наступних методів відповідають цьому делегату?

а)
```
static void Method1(int x)
{
   Console.WriteLine(x * x);
}
```

б)
```
static int Method2(ref int x)
{
   return x * x;
}
```

в)
```
static int Method3(int x, int y)
{
    return x * y;
}
```

г)
```
static int Method5(out int x)
{
    x = 7;
    return x * x;
}
```

7. Що буде виведено на консоль при виконанні наступної програми?

```
class Program
{
    delegate void Message();
    static void Main(string[] args)
    {
        Message mes1 = Hello;
        mes1 += Hi;
        mes1 += Hello;
        mes1 += Hello;
        mes1 -= Hello;
        mes1();

        Console.ReadKey();
    }
    private static void Hello() { Console.WriteLine("Hello"); }
    private static void Hi() { Console.WriteLine("Hi"); }
}
```

а) Hello;
б) Hello
Hi;
в) Hi;
г) Hello
Hi
Hello.

РОЗДІЛ 13. СТВОРЕННЯ ГРАФІЧНИХ ІНТЕРФЕЙСІВ

Для створення графічних інтерфейсів за допомогою платформи .NET застосовуються різні технології – Window Forms, WPF, застосування для магазину Windows Store. Однак найбільш простою і зручною платформою досі залишається Window Forms.

Застосування Window Forms є керованими застосуваннями. Вони містять загальну проміжну мову CIL і метадані, використовують бібліотеку класів FCL і виконуються у загальномовному середовищі CLR.

Головне достоїнство застосування технології Windows Forms в розробці Windows-застосувань в тому, що ця технологія стандартизує призначений для користувача інтерфейс і при цьому позбавлена маси помилок і нестиковок, якими грішить Windows API.

Так, кожен досвідчений Windows-програміст знає, що деякі віконні стилі можна застосувати тільки при створенні вікна. Windows Forms усуває цю проблему. Якщо ви застосовуєте такий стиль до існуючого вікна, інфраструктура непомітно знищить це вікно і відтворить його за вказаним стилем. Крім того, FCL набагато багатше Windows API, і коли ви пишете застосування Window Forms, у вашому розпорядженні вся міць FCL. Як правило, застосування Window Forms компактніше відповідних Windows-застосувань.

13.1 Створення графічного застосування

У Windows Forms термін форма – синонім вікна. Головне вікно програми є формою. Якщо у програми є інші вікна верхнього рівня, вони також є формами. Діалогові вікна – теж форми. Незважаючи на свою назву, застосування Windows Forms не виглядають як просто форми. Вони, як і традиційні Windows-застосування, здійснюють повний контроль над тим, що відбувається в їх вікнах.

Застосування Windows Forms у великій мірі залежать від класів FCL з простору імен System.Windows.Forms, що включає такі класи, як Form, який моделює поведінку вікон або «форм».

Створимо проєкт графічного застосування за допомогою середовища розробки Visual Studio Community 2022. Для цього в пункті меню File (Файл) виберемо New -> Project (Створити -> Проєкт). Після цього відкриється діалогове вікно створення нового проєкту (рисунок 13.1):

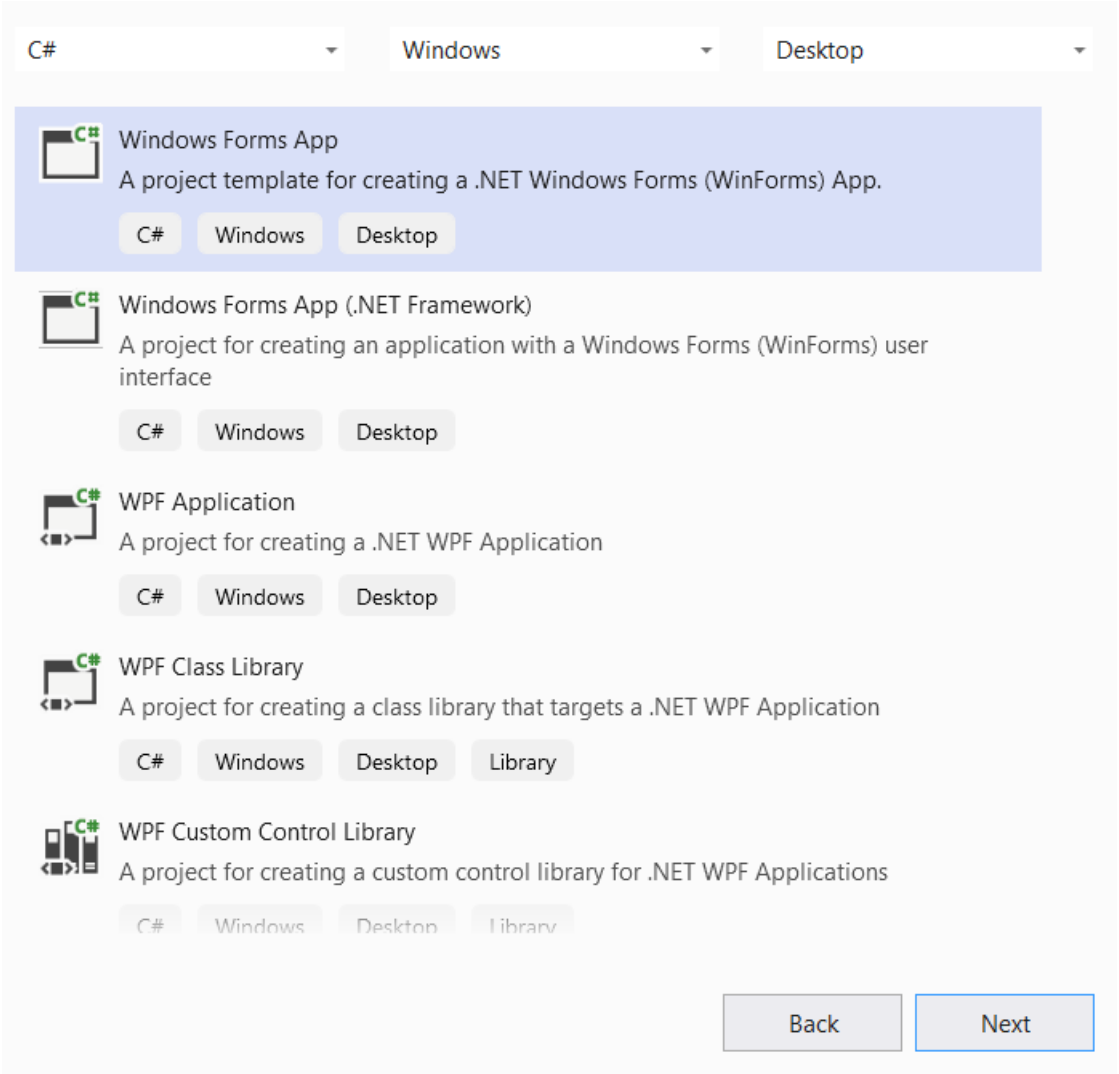


Рис. 13.1. Діалогове вікно створення нового проєкту Windows Form.

Виберемо Windows Forms App і після натискання «Next» дамо йому якесь ім'я в поле внизу. Наприклад, назвемо його HelloAppWinForms. Після цього натискаємо «Next» (Рис. 13.2)

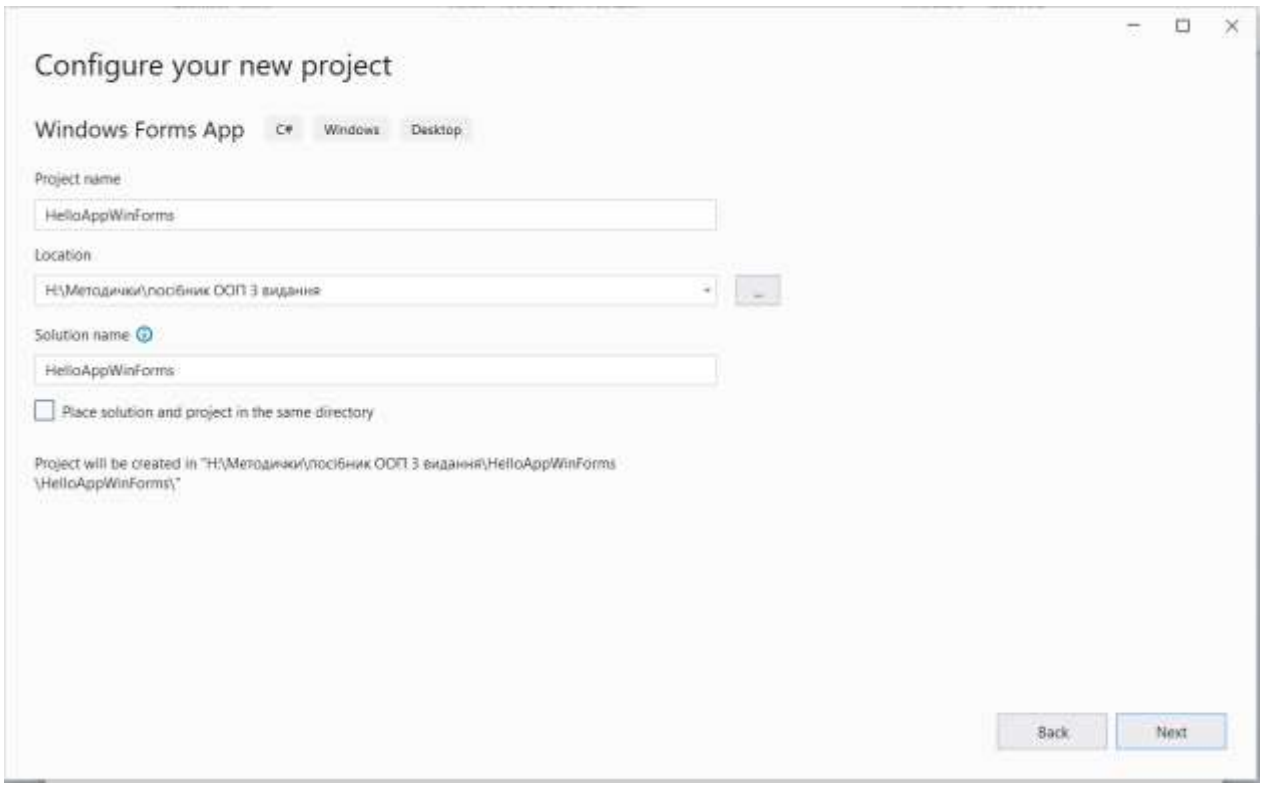


Рис. 13.2. Діалогове вікно вибору назви проєкту Windows Form.

Далі обираємо версію .NET і натискаємо Create.

Після цього Visual Studio відкриє вікно проєкту HelloAppWinForms зі створеними за замовчуванням файлами (рисунок 13.3):

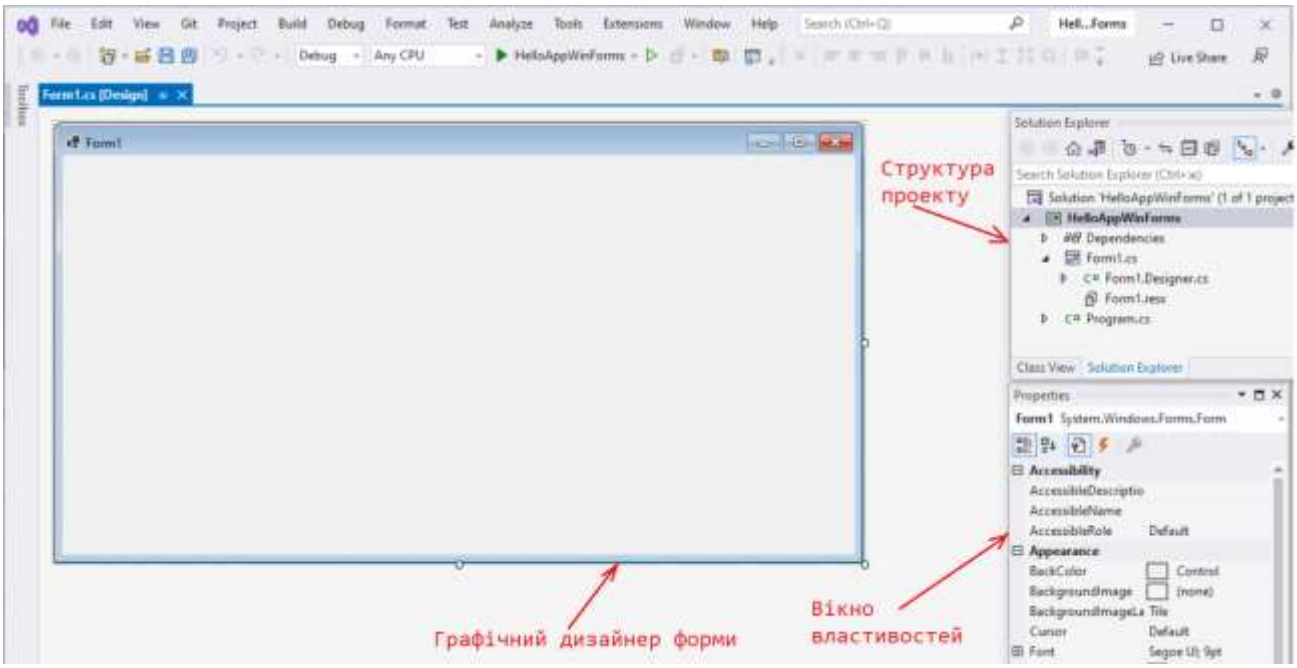
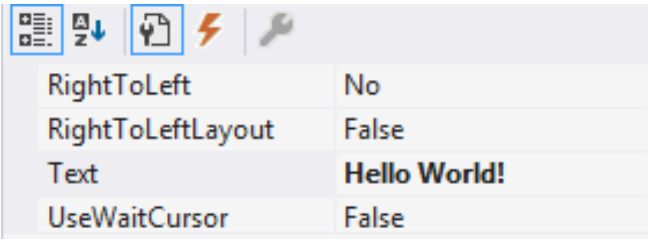


Рис. 13.3. Вікно проєкту HelloAppWinForms.

Найбільшу частину вікна Visual Studio займає графічний дизайнер, який містить форму майбутнього застосування. Поки вона порожня і має тільки заголовок Form1. Праворуч знаходиться вікно оглядача файлів рішення – Solution Explorer. Там і знаходяться всі пов'язані з нашим застосуванням файли, в тому числі файли форми Form1.cs.

Внизу праворуч знаходиться вікно властивостей Properties. З причини того, що в даний момент обрана форма як елемент управління, то в цьому полі відображаються властивості, пов'язані з формою.

Тепер знайдемо в цьому вікні властивість форми Text і змінимо його значення на «Hello World!»:



Таким чином ми поміняли заголовок форми.

Щоб запустити застосування в режимі налагодження, натиснемо на клавішу F5. Після цього відкриється вікно нашої форми (рисунок 13.4).

Рис. 13.4. Вигляд вікна форми.

13.2 Робота з формами

Зовнішній вигляд застосування представляється переважно через форми. Форми є основними будівельними блоками. Вони надають контейнер для різних елементів управління. А механізм подій дозволяє елементам форми відгукуватися на дії користувача, і, таким чином, взаємодіяти з користувачем.

При відкритті проєкту в Visual Studio можна побачити візуальну частину форми – ту частину, яку ми бачимо після запуску програми і куди ми переносимо елементи з панелі управління. Але насправді форма приховує потужний функціонал, що складається з методів, властивостей, подій тощо. Розглянемо основні властивості форм.

Якщо ми запустимо застосування, то нам відобразиться одна порожня форма. Однак навіть такий простий проєкт з порожньою формою має кілька компонентів (рисунок 13.5):

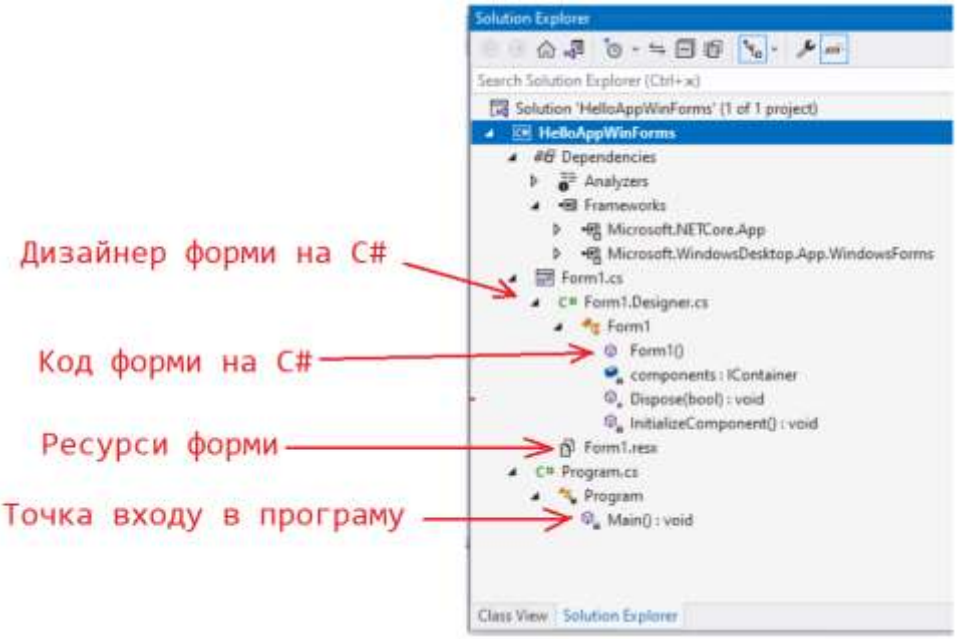


Рис. 13.5. Компоненти рішення HelloAppWinForms.

Незважаючи на те, що ми бачимо тільки форму, але стартовою точкою входу в графічне застосування є клас Program, розташований в файлі Program.cs:

```
namespace HelloAppWinForms
{
    static class Program
    {
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}
```

Спочатку програмою запускається цей клас, потім за допомогою виразу `Application.Run(new Form1())` він запускає форму Form1. Якщо раптом ми захочемо змінити стартову форму в застосуванні на якусь іншу, то нам треба змінити в цьому виразі Form1 на відповідний клас форми.

Сама форма складна за змістом. Вона ділиться на ряд компонентів. Так, в структурі проєкту є файл Form1.Designer.cs, який виконує в основному дизайнерські функції – встановлення властивостей об'єктів, установка змінних.

Тут оголошується частковий клас форми Form1, в якому є два методи: Dispose(), який виконує роль деструктора об'єкта, і InitializeComponent(), який встановлює початкові значення властивостей форми.

При додаванні елементів управління, їх опис також додається в цей файл.

Ще один файл – Form1.resx - зберігає ресурси форми. І більш важливий файл Form1.cs, який в структурі проєкту називається просто Form1 і містить код або програмну логіку форми.

```
namespace HelloAppWinForms
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
    }
}
```

13.3 Основні властивості форм

За допомогою спеціального вікна Property (Властивості), фрагмент якого зображений на рисунку 13.6, Visual Studio надає нам зручний інтерфейс для управління властивостями форми.

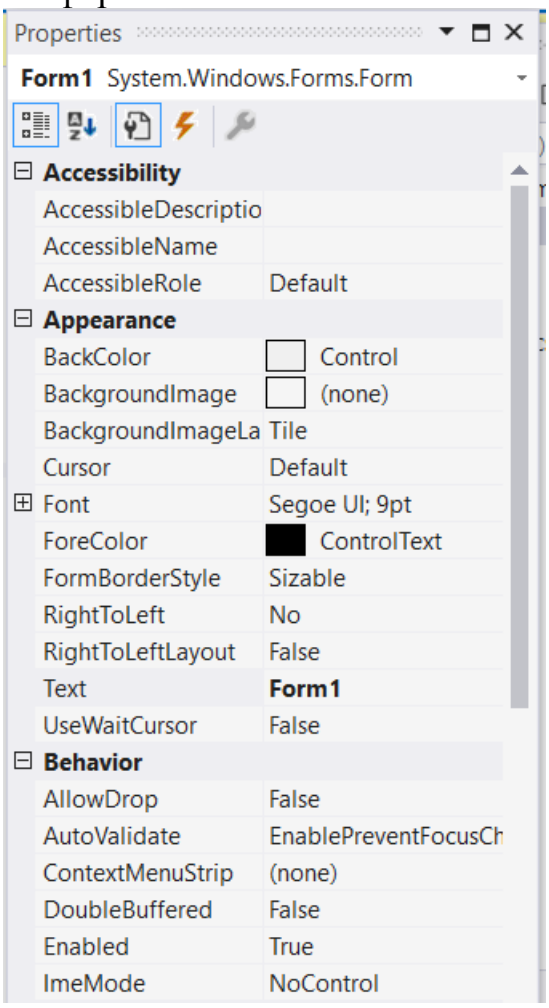


Рис. 13.6. Вигляд вікна властивостей форми.

Більшість цих властивостей впливає на візуальне відображення форми.

Ось деякі з цих властивостей:

Name: встановлює ім'я форми - точніше ім'я класу, який успадковується від класу Form;

BackColor: вказує на фоновий колір форми. Клацнувши на це властивість, ми зможемо вибрати той колір, який нам підходить зі списку запропонованих кольорів або кольорової палітри;

BackgroundImage: вказує на фонове зображення форми;

BackgroundImageLayout: визначає, як зображення, задане у властивості BackgroundImage, буде розташовуватися на формі;

ControlBox: вказує, чи відображається меню форми. В даному випадку під меню розуміється меню самого верхнього рівня, де знаходяться іконка програми, заголовок форми, а також кнопки мінімізації форми і хрестик. Якщо ця властивість має значення false, то ми не побачимо а ні іконку, а ні хрестика, за допомогою якого зазвичай закривається форма;

Cursor: визначає тип курсора, який використовується на формі;

Enabled: якщо ця властивість має значення false, то вона не зможе отримувати введення від користувача, тобто ми не зможемо натиснути на кнопки, ввести текст в текстові поля тощо;

Font: задає шрифт для всієї форми і всіх поміщених на неї елементів управління. Однак, задавши у елементів форми свій шрифт, ми можемо тим самим перевизначити його;

ForeColor: колір шрифту на формі;

FormBorderStyle: вказує, як буде відображатися межа форми і рядок заголовка. Встановлюючи ця властивість в None можна створювати зовнішній вигляд програми довільної форми;

HelpButton: вказує, чи відображається кнопка довідки форми;

Icon: задає іконку форми;

Location: визначає положення по відношенню до верхнього лівого кута екрана, якщо для властивості StartPosition встановлено значення Manual;

MaximizeBox: вказує, чи буде доступна кнопка максимізації вікна в заголовку форми;

Opacity: задає прозорість форми;

Size: визначає початковий розмір форми;

Text: визначає заголовок форми;

Visible: видима чи форма, якщо ми хочемо приховати форму від користувача, то можемо поставити даній властивості значення false.

За допомогою значень властивостей у вікні Властивості ми можемо змінити на свій розсуд зовнішній вигляд форми, але все те ж саме можна зробити програмно. Натиснемо правою кнопкою миші на формі і виберемо в контекстному меню View Code (Перегляд коду). Перед нами відкривається файл коду Form1.cs. Змінимо його наступним чином:

```
public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
            Text = "Hello World!";
            this.BackColor = Color.Aquamarine;
            this.Width = 350;
            this.Height = 150;
        }
    }
```

В даному випадку ми налаштували кілька властивостей відображення форми: заголовок, фоновий колір, ширину і висоту (рисунок 13.7).

Рис. 13.7. Вигляд вікна після зміни властивостей

13.4 Події форми

Для взаємодії з користувачем в Windows Forms використовується механізм подій. Події у Windows Forms представляють стандартні події на C#, тільки вони застосовуються до візуальних компонентів і підкоряються тим же правилам, що події в C#. Але створення обробників подій в Windows Forms все ж має деякі особливості.

У Windows Forms є певний стандартний набір подій, який здебільшого є у всіх візуальних компонентів. Окремі елементи додають свої події, але принципи роботи з ними будуть схожі. Щоб подивитися всі події елемента, нам треба вибрати цей елемент в поле графічного дизайнера і перейти до вкладки подій (Events) на панелі форм (рисунок 13.8):

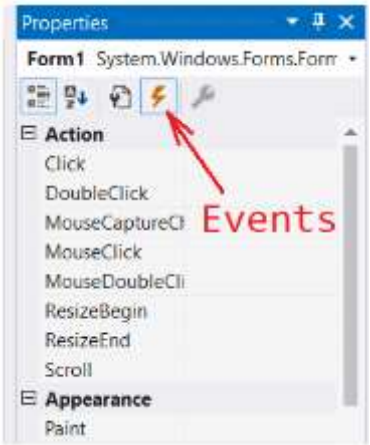
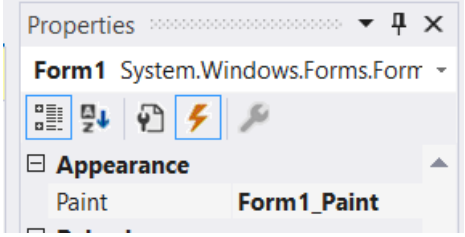


Рис. 13.8. Вигляд вікна властивостей форми зі вкладкою Events (події).

Щоб додати оброблювач, можна просто два рази натиснути по порожньому полю поруч з назвою події, і після цього Visual Studio автоматично згенерує обробник події. Наприклад, натиснемо для створення обробника для події Paint:



Як правило, більшість обробників різних візуальних компонентів мають два параметри: sender - об'єкт, який ініціював подію, і аргумент, який зберігає інформацію про подію (в даному випадку PaintEventArgs e).

Обробник події Form1_Paint автоматично буде розташований у файлі Form1.cs.

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    // тут має бути розташований код обробника події Paint
}
```

Але це тільки обробник. Додавання ж обробника, створеного таким чином, проводиться в файлі Form1.Designer.cs:

```
this.Paint += new System.Windows.Forms.PaintEventHandler(this.Form1_Paint);
```

13.4 Елементи управління

Windows Forms налічує понад 20 вбудованих типів елементів управління. Їх використання дозволяє розробникам економити час і забезпечує сумісність зовнішнього вигляду і призначеного для користувача інтерфейсу в Windows-застосуваннях.

Елементи управління – це візуальні класи, які отримують введені користувачем дані і можуть ініціювати різні події. Все вони представлені в таблиці 13.1.

Таблиця 13.1. Класи елементів управління із System.Windows.Forms:

Клас	**Опис**
Button	Командні кнопки
CheckBox	Прапорці
CheckedListBox	Списки з елементами, що містять прапорці
ComboBox	Поля зі списком
DataGrid	Елементи управління для відображення табличних даних
DataGridTextBox	Поля введення в елементах управління DataGrid
DateTimePicker	Елементи управління для введення дати і часу

Клас	Опис
GroupBox	Рамки
HScrollBar	Горизонтальні смуги прокрутки
Label	Відображається не редагований текст
LinkLabel	Відображається гіперпосилання
MontbCalendar	Елементи управління «місячний календар»
PictureBox	Елементи управління для відображення картинок
PrintPreviewControl	Елементи управління для попереднього перегляду друку
ProgressBar	Індикатори виконання
PropertyGrid	Елементи управління для відображення списків властивостей об'єктів
RadioButton	Перемикачі
RichTextBox	Поля введення з форматуванням
StatusBar	Рядки стану
TabControl	Закладки
TextBox	Поля введення
ToolBar	Панелі інструментів
TrackBar	Повзунки
TreeVieiv	Деревовидні списки
VScrollBar	Вертикальні смуги прокрутки

3.7 Динамічне додавання елементів

За допомогою перетягування елементів з Панелі Інструментів ми можемо легко додати нові елементи на форму. Однак такий спосіб досить обмежений, оскільки дуже часто потрібно динамічно створювати (видаляти) елементи на формі. Наприклад, нам треба відобразити сто кнопок і програмно змінювати їх властивості. Уявіть собі, скільки зусиль вам знадобиться, якщо ви будете це робити використовуючи елементи з панелі інструментів.

Нижче представлений перероблений зміст файлу Form1.cs. В конструкторі форми динамічно створюється масив із 26-х кнопок, на яких відображаються літери літери латинського алфавіту. При натисканні на певну кнопку обробник події ідентифікує цю кнопку і виводить повідомлення в поле Text форми. Після цього кнопка стає недоступною для подальшого використання.

```
public partial class Form1 : Form
{
  Button[] b;
  public Form1()
    {
    InitializeComponent();
    b = new Button[26];
      for (int i = 0; i < 26; i++)
       {
        b[i] = new Button(); //динамічно створюємо об'єкт Button
```

```
            b[i].Name = "b" + i.ToString();
            b[i].Size = new System.Drawing.Size(30, 30);
            b[i].Text = ((char)((int)'A' + i)).ToString();
            b[i].Top = 50;
            b[i].Left = i * 31;
            b[i].Click += new System.EventHandler(this.button1_Click);
            Controls.Add(b[i]); //додає вказаний елемент в колекцію елементів
                                //управління
        }
    }
        //Обробник події Click для кнопок
private void button1_Click(object sender, EventArgs e)
{
Button bi = (Button)sender; // посилання на об'єкт, для якого відбулася
                            //подія
bi.Enabled = false; // заборона подальшого використання
Text = "Натиснута кнопка з літерою " + bi.Text;
        }
  }
```

В результаті при запуску програми з’явиться форма з 32-а кнопками, які реагують на натискання. (Рисунок 13.9)

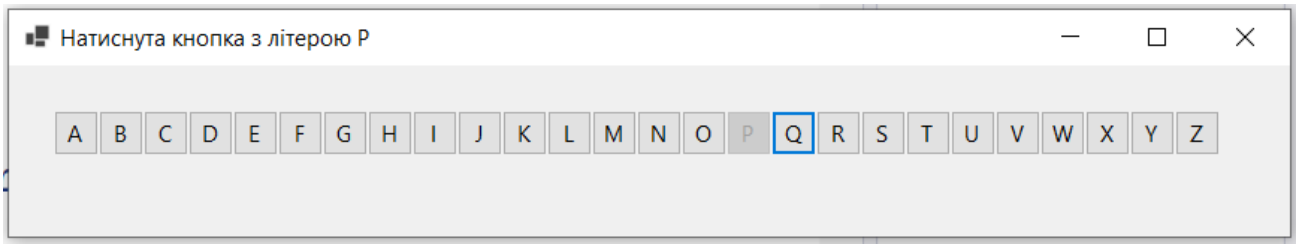


Рис. 13.9. Масив динамічно створених кнопок.

Отже Windows Forms – це важлива частина .NET Framework, тому що дозволяє розробникам створювати застосування з потужним графічним інтерфейсом користувача.

13.8 Діалогові вікна

Діалогове вікно – це форма, що застосовується для збору даних, які вводяться користувачем. Діалогові вікна, або, як їх часто називають, «діалоги» бувають модальними і немодальними. При відображенні на екрані модального діалогу його власник, яким зазвичай є головна форма програми, тимчасово стає неактивним. Вікно «Відкрити файл», що дуже часто використовується у Windows-застосуваннях, – приклад модального діалогу. Немодальні діалоги не роблять своїх власників неактивними - користувач може в будь-який момент перейти до головної формі, не закриваючи такий діалог. Приклад такого діалогу – вікно «Знайти і замінити» в Microsoft Word.

У Windows Forms діалогові вікна не мають ніяких особливостей – це звичайні форми. Фактично тільки три характеристики відрізняють форму діалогового вікна від звичайної форми:

– стилі діалогів трохи відрізняються від стилів звичайних форм, наприклад, у них рідко бувають кнопки «Згорнути» і «Розгорнути», тому властивості форми MinimizeBox і MaximizeBox для діалогів зазвичай рівні false;

– зазвичай в діалогах є кнопки «ОК» і «Скасувати», пара рядків коду в діалозі Windows Forms дозволяє повідомити інфраструктурі про ці кнопки і дозволити їй автоматично закрити діалог після натискання на них;

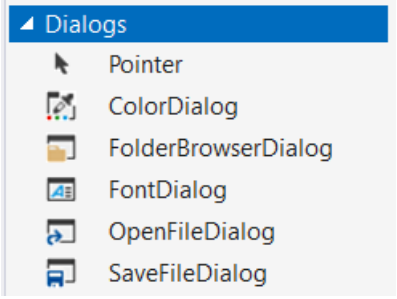
– щоб відобразити на екрані звичайну форму, викликається її метод Show, а модальний діалог – ShowDialog.

Windows надає набір системних або стандартних діалогів, які можна задіяти в застосуваннях для виведення діалогів відкриття файлу, друку тощо. FCL надає зручні класи стандартних діалогів, перераховані в таблиці 13.2. Всі ці класи визначені в просторі імен System.Windows.Forms.

Таблиця 13.2. Класи стандартних діалогів.

Клас	**Тип діалогу**
ColorDialog	Діалоги вибору кольору
FontDialog	Діалоги вибору шрифтів
OpenFileDialog	Діалоги для вибору файлів
SaveFileDialog	Діалоги введення імен файлів для збереження
FolderBrowserDialog	Діалоги для вибору каталогів
PageSetupDialog	Діалоги налаштування параметрів друкованої сторінки
PrintDiatog	Діалоги настройки параметрів друку

Далі розглянемо приклад застосування діалогових вікон OpenFileDialog і SaveFileDialog при обробці пунктів меню «Відкрити» і «Зберегти».



Перетягнемо за допомогою миші з панелі елементів в конструктор форми елементи OpenFileDialog, SaveFileDialog і MenuStrip. Вони відобразяться в конструкторі форми так, як показано на рисунку 13.10. Додамо пункти меню «Відкрити» і «Зберегти».

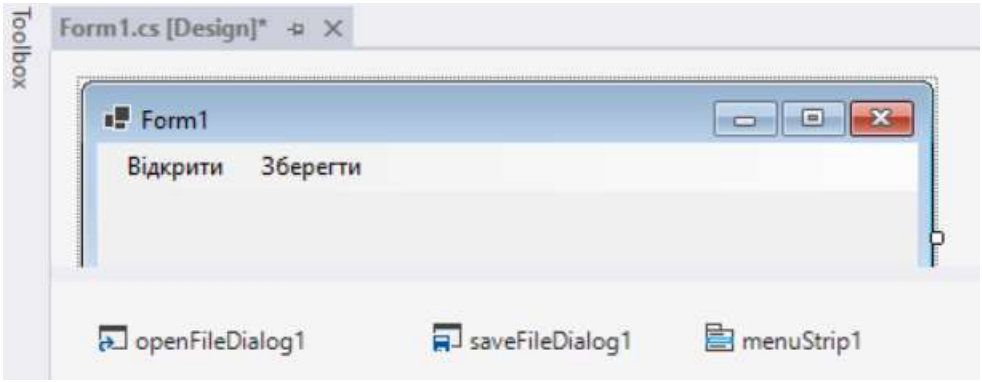


Рис. 13.10. Вікно конструктора форми з діалоговими вікнами і меню.

Додамо обробники подій для пунктів меню «Відкрити» і «Зберегти». Нижче представлений код обробника події для пункту меню «Відкрити».

```
static string fileText="";
private void відкритиToolStripMenuItem_Click(object sender, EventArgs e)
{
 openFileDialog1.Filter = "Text files(*.txt)|*.txt|All files(*.*)|*.*";
    // відкриття файлу
    if (openFileDialog1.ShowDialog() == DialogResult.Cancel)
       return;
    // отримуємо обраний файл
    string filename = openFileDialog1.FileName;
    // читаємо файл в рядок
    fileText = System.IO.File.ReadAllText(filename);
    MessageBox.Show($"Файл відкритий. Ось його зміст: {fileText}");
}
```

При запуску програми при натисканні на пункт меню «Відкрити» з’явиться діалогове вікно (рисунок 3.11)

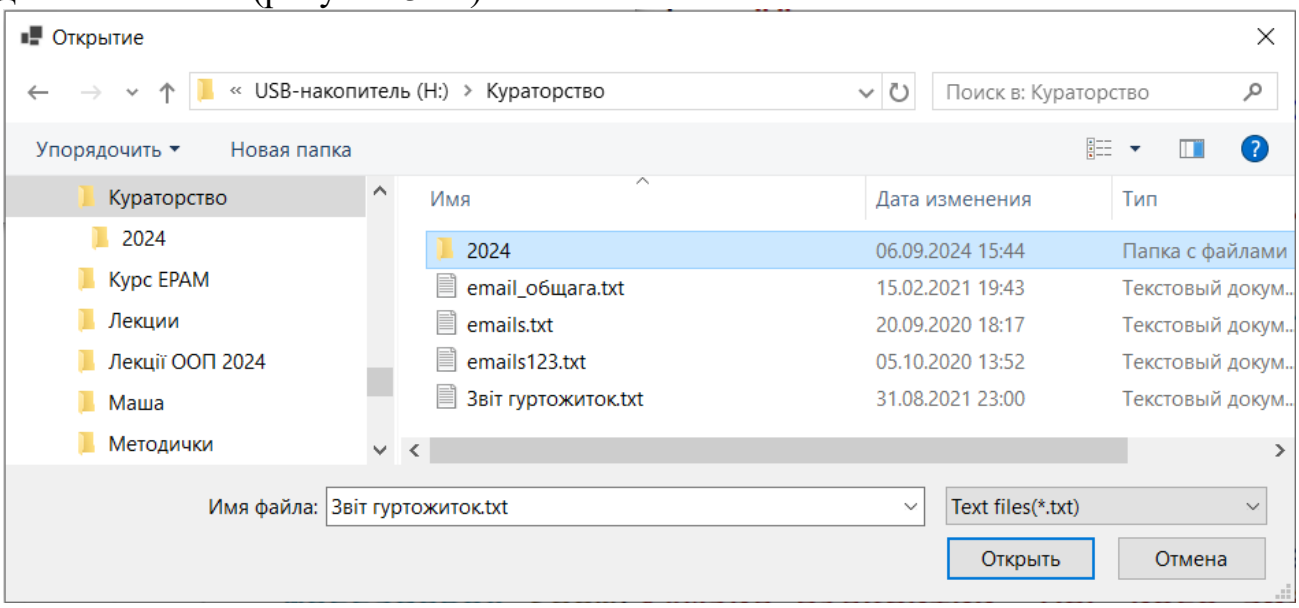


Рис. 3.11. Вигляд діалогового вікна openFileDialog.

Зверніть увагу, що завдяки властивості Filter у вікні відовражаються тільки файли з розширенням .txt.

Аналогічно обробляється і подія для пункту меню «Зберегти».

Нижче представлений код обробника події для пункту меню «Зберегти».

```
private void зберегтиToolStripMenuItem_Click(object sender, EventArgs e)
{
   // збереження файл
   if (saveFileDialog1.ShowDialog() == DialogResult.Cancel)
      return;
   // отримуємо обраний файл
   string filename = saveFileDialog1.FileName;
   // зберігаємо текст в файл
   System.IO.File.WriteAllText(filename, textBox1.Text);
   MessageBox.Show("Файл збережений");
}
```

При запуску програми при натисканні на пункт меню «Зберегти» з’явиться відповідне діалогове вікно.

Отже, зрозуміло, що Windows Forms – це важлива частина .NET Framework, тому що дозволяє розробникам створювати застосування з потужним графічним інтерфейсом користувача.

Резюме

1. У Windows Forms термін форма – синонім вікна. Головне вікно програми є формою. Якщо у програми є інші вікна верхнього рівня, вони також є формами.

2. У застосуванні Windows Forms кожна форма представлена екземпляром класу, похідного від класу Form.

3. У Windows Forms еквівалент WM_PAINT повідомлення – це віртуальний метод OnPaint класу Form. Похідний клас може перевизначити його для відображення себе у відповідь на повідомлення WM_PAINT.

4. Для взаємодії з користувачем в Windows Forms використовується механізм подій. Події у Windows Forms представляють стандартні події на C#, тільки вони застосовуються до візуальних компонентів і підкоряються тим же правилам, що події в C#.

5. Щоб додати оброблювач, можна просто два рази натиснути по порожньому полю поруч з назвою події, і після цього Visual Studio автоматично згенерує обробник події.

6. Як правило, більшість обробників різних візуальних компонентів мають два параметри: sender – об'єкт, який ініціював подію, і аргумент, який зберігає інформацію про подію (в даному випадку PaintEventArgs e).

7. Елементи управління – це візуальні класи, які отримують введені користувачем дані і можуть ініціювати різні події. Всі елементи управління успадковуються від класу Control

8. Діалогове вікно – це форма, що застосовується для збору даних, які вводяться користувачем. Діалогові вікна, або, як їх часто називають, «діалоги» бувають модальними і немодальними.

9. Щоб відобразити на екрані звичайну форму, викликається її метод Show, а модальний діалог – ShowDialog.

10. Windows надає набір системних або стандартних діалогів, які можна задіяти в застосуваннях для виведення діалогів відкриття файлу, друку тощо. FCL надає зручні класи стандартних діалогів.

Завдання до виконання

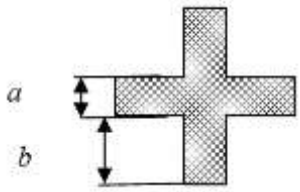
1. Створити Windows-форму фіксованого розміру, розташовану в центрі екрану (параметри розташування обчислюються в момент ініціалізації форми), одну кнопку закриття форми, діалогові вікна та інші елементи управління – для введення даних відповідно варіанту завдання. Функції-оброблювачі подій від елементів управління відповідно варіанту завдання. Клієнтська програма повинна містити перевірку усіх розроблених елементів класу та обробку виключних ситуацій.

2. Створити Windows-проєкт, який після введення користувачем із клавіатури прізвища та імені при натисканні кнопки OK або клавіші Enter виводить слова на форму у зворотному порядку, а в окремому полі повідомляє кількість символів рядка. Зберігати текст введеного рядка у файл із ім'ям, що відображається при запуску програми в одному з полів форми, при цьому передбачити можливість редагування імені файла. Зчитати вміст файлу та вивести результат в одному з полів.

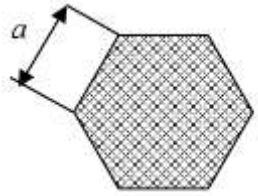
Варіанти завдань

1. Геометрична фігура «Смайлік»: його екземпляр містить розмір радіуса обличчя R і радіуса r очей, його конструктор отримує параметри із вікна Windows-форми і створює екземпляр з відповідними значеннями R і r. Метод Draw() рисує у вікні форми зображення фігури.

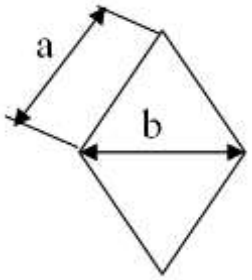
2. Геометрична фігура «хрест», такий, що: його екземпляр містить два розміри a і b, його конструктор отримує параметри із вікна Windows-форми і створює екземпляр з відповідними значеннями a і b. Метод Draw() рисує у вікні форми зображення фігури.



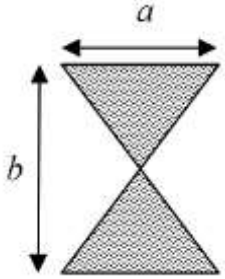
3. Геометрична фігура «Шайба» правильний шестикутник, такий, що: його екземпляр містить розмір сторони a, його конструктор отримує параметри із вікна Windows-форми і створює екземпляр з відповідним значенням a. Метод Draw() рисує у вікні форми зображення фігури.



4. Геометрична фігура «Ромб»: його екземпляр містить розміри a і b, його конструктор отримує параметри із вікна Windows-форми і створює екземпляр з відповідними значеннями a і b. Метод Draw() рисує у вікні форми зображення фігури.



5. Геометрична фігура "пісочний годинник" : його екземпляр містить розміри a і b, його конструктор отримує параметри із вікна Windows-форми і створює екземпляр з відповідними значеннями a і b. Метод Draw() рисує у вікні форми зображення фігури.



Тести для закріплення матеріалу

1. Яку подію треба пов'язати з кнопкою, щоб при натисканні на кнопку відбувалися будь-які дії?

а) Click;
б) mrButton;
в) mrButton.text;
г) EventHandler.

2. Що з нижчепереліченого є подією , що пов’язана з мишею?

а) MouseMove
б) EventHandler
в) MouseEventArgs
г) MouseEventHandler.

3. Що з нижчепереліченого є подією натискання клавіші ?

а) KeyPress;
б) EventHandler;
в) KeyPressEventArgs
г) KeyPressEventHandler.

4. Що є класом-делегатом для представлення методу обробки події натискання клавіші?

а) KeyPress;
б) EventHandler;
в) KeyPressEventArgs
г) KeyPressEventHandler.

5. Ви створили застосування Winows Forms і додали у вікні конструктора форми кілька об’єктів – елементів управління. В якому файлі виконується встановлення властивостей об'єктів?

а) Form1.cs;
б) Form1.Designer.cs;
в) Form1.resx;
г) App.config.

6. Ви створили застосування Winows Forms і додали у вікні конструктора форми кілька об’єктів – елементів управління. В якому файлі ви будете розміщувати код або програмну логіку форми?

а) Form1.cs;
б) Form1.Designer.cs;
в) Form1.resx;
г) Program.cs.

РОЗДІЛ 14. ОБРОБКА ВИНЯТКОВИХ СИТУАЦІЙ

Іноді при виконанні програми виникають помилки, які важко, а іноді і зовсім неможливо передбачити. Наприклад, при передачі файлу по мережі може несподівано обірватися підключення до мережі. Такі ситуації називаються винятковими або виключеннями.

Коли під час виконання застосування виникають проблеми, .NET сповіщає про них, генеруючи виключення. Деякі виключення генерує CLR.

В .NET маса типів виключень, і якщо програма має працювати в цьому середовищі, краще заздалегідь підготуватися до їх обробці.

Вся принадність виключень в тому, що вони є невід'ємною частиною .NET. У минулому мови програмування (і навіть окремі компілятори) генерували і обробляли виключення по-своєму. Не можна було видати виключення в Visual Basic, а перехопити його в C. Не можна було навіть згенерувати виключення всередині функції, зкомпільованою одним компілятором C, а обробити його функцією, зкомпільованою іншим компілятором. CLR визначає порядок генерації та обробки виключень. Згенерувати виключення можна на одній мові програмування, а обробити - на будь-який інший. Більш того, можна навіть передати виключення на іншу машину. І найголовніше: в таких мовах, як C# і Visual Basic .NET, обробка виключень виконується вкрай просто.

14.1 Перехоплення виключень

У C# механізм обробки виключень CLR представлений 4 ключовими словами:

try, **catch**, **finally** і **throw**. Загальна ідея в тому, щоб укласти код, який може викликати виключення, в блок try, а обробники виключень помістити в блок catch, Приклад, є код:

```
Hashtable table = new Hashtable ();
table.Add ("First", 1);
string entry = (string) table ["First"]; // здійснити приведення.
```

Ціле не є рядком, тому спроба привести 1 до рядка викличе генерацію виключення InvalidCastException (рисунок 14.1)

Рис.14.1. Вікно повідомлення про виникнення виняткової ситуації.

А ось код, який містить обробник виключень:

```
try {
    Hashtable table = new Hashtable ();
    table.Add ("First", 1);
    string entry = (string) table ["First"]; // здійснити приведення.
    }
catch (InvalidCastException e)
    {
        Console.WriteLine (e.Message);
    }
```

При виникненні виключення, воно активізує обробник InvalidCastException, який в цьому прикладі виведе на консоль повідомлення, інкапсульоване в об'єкті виключення:

```
H:\Методички\посібник ООП 3 видання\ConsoleApp2\bin\Debug\net7.0\ConsoleApp2.exe
Unable to cast object of type 'System.Int32' to type 'System.String'.
```

Щоб написати загальний обробник catch, який буде реагувати на будь-який виняток, вкажіть в якості типу виключення Exception.

```
catch (Exception e)
{
    ...
}
```

Щоб реакції на різні типи виключень відрізнялися один від одного, просто додайте обробник catch для кожного типу, який вас цікавить:

```
try
{
    ...
}
catch (InvalidCastException e)
{
    ...
}
catch (FileNotFoundException e)
{
    ...
}
catch (Exception e)
{
    ...
}
```

CLR викликає обробник, який найбільш точно відповідає типу виключення. У цьому прикладі при отриманні InvalidCastException або FileNotFoundException

виконання буде віддане одному з перших двох обробників catch. Будь-який інший тип виключення активізує останній обробник.

І в даному випадку блок catch (Exception e) {} буде обробляти всі винятки крім InvalidCastException і FileNotFoundException. При цьому блоки catch для більш загальних, найбільш фундаментальних винятків слід поміщати в кінці – після блоків catch для більш конкретних, спеціалізованих типів, тому що CLR вибирає для обробки винятку перший блок catch, який відповідає типу згенерованого виключення.

До речі, вам не треба знищувати отримані об'єкти Exception – за вас це зробить збирач сміття.

Всі визначені в FCL типи винятків є прямими або непрямими похідними від System.Exception, в якому визначено базовий набір властивостей, притаманних типам виключень FCL. Наприклад, завдяки System.Exception в усіх класах виключень FCL є властивість Message, яка містить повідомлення про помилку з описом проблеми, а властивість StackTrace містить докладні відомості про ланцюжок викликів, що призвели до виникнення виключення. У похідних класах часто додаються власні властивості. Так, клас FileNotFoundException включає властивість FileName, яка містить ім'я файлу, що призвів до появи виключення.

У FCL визначені десятки класів виключень. Вони визначені не в якомусь одному просторі імен, а розподілені приблизно у 100 просторах імен FCL.

В таблиці 14.1 наведено деякі найбільш поширені типи виключень.

Таблиця 14.1. Загальні класи виключень FCL

Клас	**Умови виникнення**
ArgumentNullException	Недозволена передача в якості аргументу порожнього посилання.
ArgumentOutOJRangeException	Неприпустимий аргумент (поза діапазону).
DivideByZeroException	Спроба ділення на 0
IndexOutOfRangeException	Використано невірний індекс масиву.
InvalidCetstException	Значення приведено до невідповідному типу.
NullReferenceException	Порожне посилання.
OutOfMemoryException	Невдача при виділенні пам'яті через її нестачі.
WebException	Помилка при обробці HTTP-запиту.

Обробники виключень можуть бути вкладеними. Якщо метод А викликає метод В, а В сповіщає про виключення, викликається обробник виключень В за умови, що потрібний обробник там є. Якщо в методі В немає обробника виключення певного типу, CLR шукає його в методі А. Якщо в А теж немає відповідного обробника, повідомлення про виключення передається наверх методу, який викликав А, і так далі.

Що робить .NET з необробленими винятками? Це залежить від виду програми. Коли необроблене виключення виявляється у консольному застосуванні, інфраструктура завершує програму і видає у вікні консолі повідомлення про помилку. Якщо ж застосування відноситься до Windows

Forms, інфраструктура формує повідомлення з попередженням для користувача. Для застосувань Web Forms інфраструктура показує сторінку помилок. Взагалі краще передбачити виключення і обробляти їх, ніж ставити користувача перед фактом виникнення необробленого виключення.

14.2 Блок catch і фільтри виключень

За обробку виключення відповідає блок catch, який може мати такі форми:

1. Обробляє будь яке виключення, яке виникло в блоці try:

```
catch
  {
      //  інструкції, що виконуються
  }
```

2. Обробляє тільки ті виключення, які відповідають типу, що зазначений в дужках після оператора catch:

```
catch(тип_виключення)
{
    //  інструкції, що виконуються
}
```

Наприклад, обробимо тільки виключення типу DivideByZeroException:

```
try
{
  int x = 5, y = 0;
  int z = x / y;
  Console.WriteLine ($"Результат: {z}");
}
catch (DivideByZeroException)
{
 Console.WriteLine ("Виникло виключення DivideByZeroException");
}
```

Однак якщо в блоці try виникнуть виключення якихось інших типів, відмінних від DivideByZeroException, то вони не будуть оброблені.

3. Обробляє тільки ті виключення, які відповідають типу, що зазначений в дужках після оператора catch. А вся інформація про виключення поміщається в змінну даного типу.

```
catch(тип_виключення ім'я змінної)
{
    //  інструкції, що виконуються
}
```

Наприклад:

```
try
{
  int x = 5, y = 0;
  int z = x / y;
  Console.WriteLine ($ "Результат: {y}");
}
catch(DivideByZeroException ex)
{
  Console.WriteLine($"Виникло виключення {ex.Message}");
}
```

Фактично цей випадок аналогічний попередньому за тим винятком, що тут використовується змінна. В даному випадку в змінну ex, яка представляє тип DivideByZeroException, поміщається інформація про виключення. І за допомогою властивості Message ви можете отримати повідомлення про помилку.

Якщо вам не потрібна інформація про виключення, то змінну можна не використовувати як в попередньому випадку.

Фільтри виключень

Фільтри виключень дозволяють обробляти виключення в залежності від певних умов. Для їх застосування після ключового слова catch йде вираз when, після якого в дужках вказується умова:

```
catch when(умова)
{

}
```

В цьому випадку обробка виключення в блоці catch проводиться тільки в тому випадку, якщо умова у виразі when істинно (true). Наприклад:

```
int x = 5, y = 0;
try
{
    int z = x / y;
    Console.WriteLine($"Результат: {z}");
}
catch (DivideByZeroException) when (y == 0 && x == 0)
{
    Console.WriteLine("y не може дорівнювати 0");
}
catch (DivideByZeroException ex)
{
    Console.WriteLine(ex.Message);
}
```

В даному випадку виникне виключення, тому що y = 0. Тут є два блока catch, і обидва вони обробляють виключення типу DivideByZeroException, тобто по суті всі винятки, які генеруються при діленні на нуль. Але оскільки для першого блоку зазначена умова `y == 0 && x == 0`, то вона не буде обробляти виключення бо умова, що вказана після оператора when, повертає false. Тому

CLR буде далі шукати відповідні блоки catch далі і для обробки виключення вибере другий блок catch. В результаті якщо ми приберемо другий блок catch, то виключення взагалі не буде оброблятися.

14.3 Гарантоване виконання

Код блоку finally виповнюється гарантовано, незалежно від того, чи виникло виключення. Ключове слово finally дуже зручно, коли доводиться працювати з класами-оболонками, що включають дескриптори файлів та інші некеровані ресурси. Якщо ви написали код:

```
StreamReader file= File.OpenText("readme.txt");
file.Close();
```

файл залишиться відкритим при виникненні виключення після відкриття файлу, але до виклику Close. Однак ви можете відчувати себе спокійно, якщо структурували свій код так:

```
StreamReader file = null;
try
{
    file = File.OpenText("readme.txt");
}
catch (FileNotFoundException e)
{
    Console.WriteLine(e.Message);
}
finally
{
    if(file != null)
        file.Close();
}
```

Тепер Close буде викликаний незалежно від виникнення виключення. Майте на увазі: супроводжувані блоками finally блоки try не зобов'язані мати блоки catch.

Припустимо, в попередньому прикладі ви вирішили переконатися, що файл закритий, але вам не хотілося б самим обробляти виключення. Краще б передати його методу, що знаходиться вище в стеку викликів. Ось як це зробити:

```
StreamReader file = null;
try
{
    file = File.OpenText("readme.txt");
}
finally
{
    if(file != null)
        file.Close();
}
```

Цей код абсолютно коректний і демонструє, як реагувати на виключення: краще його обробляти викликаючому методу, а не методу, який викликається. Розробникам бібліотек класів треба особливо уважно ставитися до того, щоб не «проковтнути» виключення, про яке слід сповістити того, хто викликав клас.

14.4 Створення класів виключень

Якщо вас не влаштовують вбудовані типи виключень, то ви можете створити свої типи. Базовим класом для всіх виключень є клас Exception, відповідно для створення своїх типів ви можете успадкувати даний клас.

Припустимо, у нас в програмі буде обмеження за вагою для багажу літака:

Є клас Baggage:

```
class Baggage
{
    double weight;
    public double Weight
    {
        get { return weight; }
        set
        {
            if (value > 7)
            {
              throw new Exception("Вага не може перевищувати 7 кг.");
            }
            else
            {
                weight = value;
            }
        }
    }
}
```

Ключове слово **throw** можна застосовувати для ініціювання виключень, визначених у FCL, а також для ініціювання виключень, визначених вами самими.

У методі Main міститься такий обробник виключень:

```
            try
            {
                Baggage b = new Baggage { Weight = 100 };
            }
            catch (Exception ex)
            {
                Console.WriteLine($"Увага: {ex.Message}");
            }
```

У класі Baggage при встановленні ваги відбувається перевірка, і якщо вага перевищує 7 кг., тоді виникає виключення.

Клас Exception приймає в конструкторі як параметр рядок, який потім передається в його властивість Message.

Але іноді зручніше використовувати свої класи виключень. Наприклад, в якійсь ситуації ми хочемо обробити певним чином тільки ті виключення, які відносяться до класу Baggage. Для цих цілей ми можемо зробити спеціальний клас BaggageException:

```
class BaggageException : Exception
{
    public BaggageException(string message) : base(message)
    { }
}
```

По суті клас крім порожнього конструктора нічого не має, і то в конструкторі ми просто звертаємося до конструктора базового класу Exception, передаючи в нього рядок message. Але тепер ми можемо змінити клас Baggage, щоб він викликав виключення саме цього типу і відповідно в основній програмі обробляти це виключення:

```
class Baggage
{
    double weight;
    public double Weight
    {
        get { return weight; }
        set
        {
            if (value > 7)
            {
        throw new BaggageException("Вага не може перевищувати 7 кг.");
            }
            else
            {
                weight = value;
            }
        }
    }
}
```

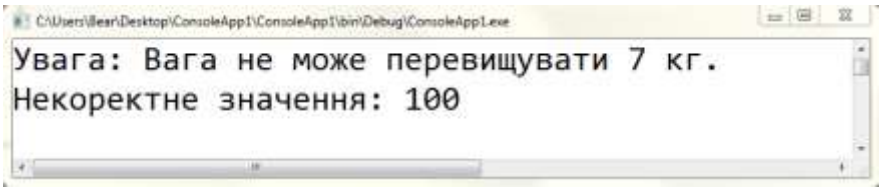
Кожен тип виключень може визначати якісь свої властивості. Наприклад, в даному випадку ми можемо визначити в класі властивість для зберігання значення, що встановлюється:

```
class BaggageException : Exception
{
  public double Value { get; }
  public BaggageException(string message, double val) : base(message)
  {
      Value = val;
```

```
    }
}
```

У конструкторі класу ми встановлюємо цю властивість і при обробці виключення ми його можемо отримати:

```
try
{
    Baggage b = new Baggage { Weight = 100 };
}
catch (BaggageException ex)
{
Console.WriteLine($"Увага: {ex.Message}");
Console.WriteLine($"Некоректне значення: {ex.Value}");
}
```



14.5 Пошук блоку catch при обробці виключень

Якщо код, який викликає виключення, не розміщен в блоці try або поміщений в конструкцію try..catch, яка не містить відповідного блоку catch для обробки виниклого виключення, то система здійснює пошук відповідного обробника виключення в стеці викликів.

Розглянемо наступний приклад, є клас Cat в якому є два метода – Meow і CatchMouse:

```
class Cat
  {
      public static void Meow()
      {
          try
          {
              CatchMouse();
          }
          catch (IndexOutOfRangeException ex)
          {
              Console.WriteLine($"Catch в методі Meow : {ex.Message}");
          }
          finally
          {
              Console.WriteLine("Блок finally в Meow");
          }
          Console.WriteLine("Завершення методу Meow");
      }
```

```
    static void CatchMouse()
    {
        try
        {
            File.ReadAllText("C:\\mouse.eat");
        }
        finally
        {
            Console.WriteLine("Блок finally в CatchMouse");
        }
        Console.WriteLine("Завершення методу CatchMouse");
    }
}
```

Метод Main містить наступний код:

```
        try
        {
            Cat.Meow();
        }
        catch (FileNotFoundException ex)
        {
            Console.WriteLine($"Catch в Main : {ex.Message}");
        }
        finally
        {
            Console.WriteLine("Блок finally в Main");
        }
            Console.WriteLine("Завершення методу Main");
```

В даному випадку стек викликів виглядає наступним чином: метод Main викликає метод Meow, який, в свою чергу, викликає метод CatchMouse. І в методі CatchMouse генерується виключення FileNotFoundException. Візуально стек викликів можна представити як зображено на рисунку 14.2:

Рис.14.2. Стек викликів обробників виключень.

Внизу стека метод Main, з якого почалося виконання, і на самому верху метод CatchMouse.

Що буде відбуватися в даному випадку при генерації виключення?

1. Метод Main викликає метод Meow, а той викликає метод CatchMouse, в якому генерується виключення FileNotFoundException.
2. Система бачить, що код, який викликав виключення, поміщений в конструкцію try..catch.

```
try
{
    File.ReadAllText("C:\\mouse.eat");
}
finally
{
    Console.WriteLine("Блок finally в CatchMouse");
}
```

Система шукає в цій конструкції блок catch, який обробляє виключення FileNotFoundException. Однак такого блоку catch немає.

3. Система опускається в стеці викликів в метод Meow, який викликав CatchMouse. Тут виклик CatchMouse поміщений в конструкцію try..catch.

```
try
{
  CatchMouse();
}
catch (IndexOutOfRangeException ex)
{
  Console.WriteLine($"Catch в методі Meow : {ex.Message}");
}
finally
{
  Console.WriteLine("Блок finally в Meow");
}
```

Система також шукає в цій конструкції блок catch, який обробляє виключення FileNotFoundException. Однак тут також подібний блок catch відсутній.

4. Система далі опускається в стеці викликів в метод Main, який викликав Meow. Тут виклик Meow поміщений в конструкцію try..catch.

```
try
{
    Cat.Meow();
}
catch (FileNotFoundException ex)
{
    Console.WriteLine($"Catch в Main : {ex.Message}");
}
finally
{
    Console.WriteLine("Блок finally в Main");
}
```

Система знову шукає в цій конструкції блок catch, який обробляє виключення FileNotFoundException. І в даному випадку такий блок знайдений.

5. Система нарешті знайшла потрібний блок catch в методі Main, для обробки виключення, яке виникло в методі CatchMouse – тобто до початкового методу, де безпосередньо виникло виключення. Але поки цей блок catch НЕ виконується. Система піднімається назад по стеку викликів в самий верх в метод CatchMouse і виконує в ньому блок finally:

```
finally
{
    Console.WriteLine("Блок finally в CatchMouse");
}
```

6. Далі система повертається по стеку викликів вниз в метод Meow і виконує в ньому блок finally:

```
finally
{
    Console.WriteLine("Блок finally в Meow");
}
```

7. Потім система переходить по стеку викликів вниз в метод Main і виконує в ньому знайдений блок catch і наступний блок finally:

```
catch (FileNotFoundException ex)
{
    Console.WriteLine($"Catch в Main : {ex.Message}");
}
finally
{
    Console.WriteLine("Блок finally в Main");
}
```

8. Далі виконується код, який йде в методі Main після конструкції try..catch:

```
Console.WriteLine("Завершення методу  Main");
```

Варто зазначити, що код, який йде після конструкції try ... catch в методах Meow і CatchMouse, не виконується, тому що обробник виключення знайдений саме в методі Main.

Результат роботи програми:

Головна перевага обробки виняткових ситуацій полягає в тому, що вона дозволяє автоматизувати отримання більшої частини коду, який доводилося раніше вводити в будь-яку велику програму вручну для обробки помилок. Так, якщо програма написана мовою програмування без обробки виняткових ситуацій, то при невдалому виконанні методів доводиться повертати коди помилок, які необхідно перевіряти вручну за кожного виклику методу.

Резюме

1. Обробка виняткових ситуацій (exception handling) – механізм, призначений для опису реакції програми на помилки часу виконання та інші можливі проблеми (виключення), які можуть виникнути під час виконання програми та призводять до неможливості (безглуздості) подальшого відпрацювання програмою її базового алгоритму.

2. Платформа .NET надає розробникам можливості для обробки виняткових ситуацій. Для цього призначена конструкція try ... catch ... finally.

3. При використанні блоку try ... catch..finally спочатку виконуються всі інструкції в блоці try. Якщо в цьому блоці не виникло винятків, то після його виконання починає виконуватися блок finally. І потім конструкція try..catch..finally завершує свою роботу.

4. Якщо в блоці try виникає виняток, то середовище CLR починає шукати блок catch, який може обробити цей виняток. Якщо потрібний блок catch знайдений, то він виконується, і після його завершення виконується блок finally.

5. Ключове слово throw можна застосовувати для ініціювання виключень, визначених у FCL, а також для ініціювання своїх власних виключень.

6. Фільтри виключень дозволяють обробляти виключення в залежності від певних умов. Для їх застосування після ключового слова catch йде вираз when, після якого в дужках вказується умова.

7. Базовим класом для всіх виключень є клас Exception, відповідно для створення своїх власних виключень можна успадкувати даний клас, або будь-який інший його нащадок.

8. Якщо код, який викликає виключення, не поміщений в конструкцію try..catch, яка не містить відповідного блоку catch для обробки виниклого виключення, то система здійснює пошук відповідного обробника виключення в стеці викликів.

9. Блоки catch для більш загальних, винятків слід поміщати в кінці – після блоків catch для більш конкретних, спеціалізованих типів.

10. Головна перевага обробки виняткових ситуацій полягає в тому, що вона дозволяє автоматизувати отримання більшої частини коду, який доводилося раніше вводити в будь-яку велику програму вручну для обробки помилок.

Завдання до виконання

1. Створити клас відповідно варіанту завдання. При виникненні специфічних для класу помилок повинні вкидатися виключні ситуації.
2. Клієнтська програма повинна містити перевірку усіх розроблених елементів класу та обробку виключних ситуацій.

Варіанти завдання

Варіант 1. Розробити клас для роботи з матрицями. Реалізувати операції, що дозволяють виконувати додавання, віднімання і множення матриць, передбачивши можливість їх виконання, в іншому випадку має генеруватися призначене для користувача виключення.

Варіант 2. Розробити клас для представлення дати з можливістю встановлення відповідних полів: рік, місяць, день з перевіркою припустимості значень, що вводяться; У випадку отримання неприпустимих значень повинні виникати та оброблятися відповідні виключні ситуації.

Варіант 3. Розробити клас для представлення раціональних дробів. Повинні бути визначені операції додавання, віднімання, множення, ділення У випадку отримання неприпустимих значень повинні виникати та оброблятися відповідні виключні ситуації.

Варіант 4. Описати клас для представлення часу. Повинні бути визначені: можливість встановлення часу та зміни його окремих полів (години, хвилини, секунди) з перевіркою припустимості значень, що вводяться; У випадку отримання неприпустимих значень має генеруватися призначене для користувача виключення.

Варіант 5. Розробити клас, що визначає гравця в комп'ютерні ігри. Клас повинен мати такі властивості: ім'я гравця, його вік, рейтинг, тип гри. Деякі ігри вимагають обмежень по віку. При створенні екземпляра класу з недозволеним значенням віку, має генеруватися призначене для користувача виключення.

Варіант 6. Створити клас "розумний масив", у якого є індексатор, який перевіряє, чи знаходиться індекс елемента масиву в потрібному діапазоні. Якщо індекс виходить за межи допустимого діапазону, тоді генерується призначене для користувача виключення.

Варіант 7. Створіть клас Calculator. У тілі класу створіть чотири методи для арифметичних дій: (Add – додавання, Sub – віднімання, Mul – множення, Div – поділ).

Метод поділу повинен перевірити поділ на нуль, якщо перевірка не проходить, згенерувати виняток.

Користувач вводить значення, над якими хоче зробити операцію та вибрати саму операцію.

У разі виникнення помилок повинні викидатися винятки.

Варіант 8. Описати структуру з ім'ям Worker, що містить такі поля:

• прізвище та ініціали працівника;

• назва займаної посади;

• рік вступу працювати.

Написати програму, яка виконує такі дії:

• введення з клавіатури даних до масиву, що складається з п'яти елементів типу Worker (записи мають бути впорядковані за абеткою);

• якщо значення року введено не у відповідному форматі, видає виняток.

• виведення на екран прізвища працівника, стаж роботи якого перевищує введене значення.

Тести для закріплення матеріалу

1. Метод витягує деяку інформацію з бази даних і потім закриває з'єднання. Ви повинні переконатися, що ваш код коректно завершується разі помилки бази даних. Щоб впоратися з цією ситуацією, ви ставите код бази даних в блок try-catch-finally. Ви використовуєте два блока catch - один для перехоплення виключень типу SqlException, а другий для перехоплення виключення типу Exception. Яке з наступних місць повинно бути кращим вибором для закриття об'єкта SqlConnection?

а) Усередині блоку try, до першого блоку catch

б) Усередині блоку catch, який ловить об'єкти SqlException

в) Усередині блоку catch, який ловить об'єкти Exception

г) Усередині блоку finally.

2. Ви хочете зрозуміти помилку, яка виникає при компіляції вашої програми. Нижче наводиться проблемна частина вашого коду:

```
try
{
    bool success = ApplyPicardoRotation(100, 0);
}
catch (DivideByZeroException dbze)
       { // код обробки винятків  }
catch (NotFiniteNumberException nfne)
       { // код обробки винятків  }
catch (ArithmeticException ae)
       { // код обробки винятків  }
```

```
catch (OverflowException oe)
       { // код обробки винятків  }
```

Щоб усунути помилку компіляції, який із наступних способів ви повинні застосувати?

а)
```
try
{
    bool success = ApplyPicardoRotation(100, 0);
}
catch (DivideByZeroException dbze)
       { // код обробки винятків  }
catch (ArithmeticException ae)
       { // код обробки винятків  }
catch (OverflowException oe)
       { // код обробки винятків  }
```

б)
```
try
{
    bool success = ApplyPicardoRotation(100, 0);
}
catch (DivideByZeroException dbze)
       { // код обробки винятків  }
catch (Exception ae)
       { // код обробки винятків  }
catch (OverflowException oe)
       { // код обробки винятків  }
```

в)
```
try
    {
        bool success = ApplyPicardoRotation(100, 0);
    }
    catch (DivideByZeroException dbze)
        { // код обробки винятків  }
    catch (NotFiniteNumberException nfne)
        { // код обробки винятків  }
    catch (OverflowException oe)
        { // код обробки винятків  }
    catch (ArithmeticException ae)
        { // код обробки винятків  }
```

г)
```
try
    {
        bool success = ApplyPicardoRotation(100, 0);
    }
    catch (DivideByZeroException dbze)
        { // код обробки винятків  }
    catch (NotFiniteNumberException nfne)
        { // код обробки винятків  }
    catch (Exception e)
        { // код обробки винятків  }
```

```
catch (ArithmeticException ae)
     { // код обробки винятків  }
```

3. Є наступний код:

```
try
{
  int x = 5, y = 0;
  int z = x / y;
  Console.WriteLine ($"Результат: {z}");
}
catch (FileNotFoundException)
{
 Console.WriteLine ("Не можна ділити на 0");
}
```

Що буде виведено на екран при виконанні цієї програми?
а) Результат z;
б) Не можна ділити на 0;
в) Виникне необроблена виняткова ситуація;
г) Результат безмежність;

4. Що трапиться при виконанні наступного коду:

```
static void Method1()
{
    try
    {
        int x = 5, y = 0;
        int z = x / y;
    }
    finally
    {
        Console.WriteLine("Не можна ділити на 0");
    }
}
```

а) Буде виведено повідомлення "Не можна ділити на 0";
б) Виникне необроблена виняткова ситуація;
в) Система опуститься в стеці викликів в метод, який викликав метод Method1;
г) Помилка компіляції: не може бути блок try без блоку catch.

СПИСОК ТЕРМІНІВ

Термін	**№ сторінки**
Абстрактний клас	109, 224
Абстрактний метод	225
Абстракція	7
Автоматичні властивості	179
Алгоритм	7, 121
Анонімні методи	277
Байт (byte)	138
Виключення	123, 312
Віртуальні базові класи	93
Віртуальні методи і властивості	214
Віртуальні функції	104
Вказівник	36, 63
Вказівники на похідні типи	102
Властивості	176, 214
Декомпозиція	7
Делегат Action	290
Делегат Predicate	291
Делегат Predicate	291
Делегати	259
Деструктор	23
Дружні функції	39
Значення null	172, 173
Ієрархія	8
Індексатори	181
Інкапсуляція	9, 13, 190
Інтерфейс	12, 241
Керований код	149
Клас	17, 43, 121, 155
Ключове слово base	213

Термін	**№ сторінки**
Ключове слово this	55, 176
Коваріантність делегатів	286
Компілятор	20
Компілятор JIT	148
Конструктор	25, 49, 53, 157
Конструктор за замовчуванням	42, 157
Конструктор копії	49
Контраваріантність делегатів	286
Купа (heap)	181
Лямбда-вираз	282
Множинне спадкування	82
Мова програмування C#	6, 130
Мова програмування C++	6, 7, 17
Мова програмування	9
Модифікатор internal	174
Модифікатор override	214
Модифікатор private	18, 86, 174, 201
Модифікатор protected	86, 90, 174, 201
Модифікатор public	18, 86, 174, 243
Модифікатор sealed	225
Модифікатор static	201
Модифікатор virtual	225
Модифікатори доступу	174
Спадкування	15, 76, 211
Спадкування інтерфейсів	246
Неявна реалізація	249
Некерований код	149
Об’єкт	9, 25, 109, 157
Об'єктно-орієнтований підхід	9
Обробка виключень	123, 312
Парадигма	10

Термін	**№ сторінки**
Параметризрвані конструктори	26, 38,180
Перевантаження конструкторів	44
Перевантаження операторів	63, 197
Події	273, 302
Поліморфізм	13, 102, 200
Принципи об'єктно-орієнтованого програмування	8
Присвоювання об'єктів	46
Простір імен	151
Псевдокод MSIL	148
Середовище .NET	148
Середовище CLR	131
Сигнатура методу	241
Статичний метод	136, 189
Статичні члени	189
Стек (stack)	14, 167, 266
Структури	162
Суто віртуальні функції	109
Суфікс	141
Типи значень (value types)	165
Типи посилальні (reference types)	165
Узагальнені класи	121
Узагальнені функції	118
Шаблони	118
Явна реалізація	247

ЛІТЕРАТУРА

1. Порєв В.М. Об'єктно-орієнтоване програмування: конспект лекцій [Електронний ресурс]: навч. посіб. для студ спеціальності 121 «Інженерія програмного забезпечення» / КПІ ім. Ігоря Сікорського. – Київ : КПІ ім. Ігоря Сікорського, 2022. – 271 с. Режим доступу: http://library.megu.edu.ua:8180/jspui/bitstream/123456789/4128/1/2022-konsp_lec_oop_porev.pdf

2. Гришанович Т. О., Глинчук Л. Я. Основи об’єктно-орієнтованого програмування : навч. посібник / ВНУ імені Лесі Українки. Електронні текстові данні (1 файл: 998 КБ). Луцьк : ВНУ імені Лесі Українки, 2022. – 120 с.

3. Коноваленко І.В., Марущак П.О. Платформа .NET та мова програмування C# 8.0: навчальний посібник – Тернопіль: ФОП Паляниця В. А., 2020 – 320 с.

4. W3schoolsUA. українською. C# ООП. [Електронний ресурс]: Режим доступу: https://w3schoolsua.github.io/cs/cs_oop.html

5. Патерни проектування. REFACTORING GURU [Електронний ресурс] Режим доступу: https://refactoring.guru/uk/design-patterns

6. The Unified Modeling Language [Електронний ресурс] Режим доступу: https://www.uml-diagrams.org

7. Microsoft. Creating and Managing Visual C++ Projects [Електронний ресурс] Режим доступу: https://msdn.microsoft.com/en-us/library/4457htyc.aspx

8. OMG Standards Development Organization. Unified Modeling Language [Online] Available from: https://www.omg.org/spec/UML/

9. Microsoft. Nested Class Declarations [Online] Available from: https://learn.microsoft.com/en-us/cpp/cpp/nested-class-declarations?view=msvc170