МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

В. М. Порєв

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ

Конспект лекцій

Навчальний посібник

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для здобувачів ступеня бакалавра
за освітньою програмою «Інженерія програмного забезпечення комп'ютерних систем»
спеціальності 121 «Інженерія програмного забезпечення»

Електронне мережне навчальне видання

Київ
КПІ ім. Ігоря Сікорського
2022

Об'єктно-орієнтоване програмування: конспект лекцій [Електронний ресурс]: навч. посіб. для студ. освітньої програми «Інженерія програмного забезпечення комп'ютерних систем» спеціальності 121 «Інженерія програмного забезпечення» / Порєв В.М.; КПІ ім. Ігоря Сікорського. – Електронні текстові дані (1 файл: 4,8 МБайт). – Київ : КПІ ім. Ігоря Сікорського, 2022. – 271 с. – Назва з екрана.

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського
(протокол № 1 від 02.09.2022 р.)
за поданням Вченої ради факультету інформатики та обчислювальної техніки
(протокол № 11 від 11.07.2022 р.)

Електронне мережне навчальне видання

Порєв Віктор Миколайович, к.т.н.

Рецензент: Лісовиченко О.І., к.т.н., доцент, КПІ ім. Ігоря Сікорського, факультет інформатики та обчислювальної техніки, кафедра інформатики та програмної інженерії

Відповідальний редактор: Волокита А.М., к.т.н., доцент, КПІ ім. Ігоря Сікорського, факультет інформатики та обчислювальної техніки, кафедра обчислювальної техніки

У навчальному посібнику викладені матеріали лекційних занять для дисципліни «Об'єктно-орієнтоване програмування». Наведені відомості щодо об'єктно-орієнтованого підходу до створення програмного забезпечення комп'ютерних систем. Розглянуто модульність проєктів, використання патернів об'єктно-орієнтованого проектування. Викладено положення щодо реалізації об'єктно-орієнтованого підходу на основі мови програмування C++. Приділено увагу, зокрема, основним аспектам розроблення керованих повідомленнями програмних додатків для операційної системи Windows.

Навчальний посібник призначений для здобувачів ступеня бакалавра за освітньою програмою «Інженерія програмного забезпечення комп'ютерних систем»» спеціальності 121 «Інженерія програмного забезпечення». Може бути також корисним студентам, які вивчають програмування за іншими спеціальностями.

Реєстр. № НП 22/23-015.Обсяг 8 авт. арк.

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
проспект Перемоги, 37, м. Київ, 03056
https://kpi.ua
Свідоцтво про внесення до Державного реєстру видавців, виготовлювачів
і розповсюджувачів видавничої продукції ДК № 5354 від 25.05.2017 р.

ЗМІСТ

ВСТУП

Об'єктно-орієнтований підхід є достатньо популярним та актуальним натепер поняттям, незважаючи на те, що основні ідеї щодо нього були сформульовані та проголошені ще наприкінці минулого сторіччя. Одна й та сама україномовна абревіатура «ООП» може означати дещо різні речі, а саме: об'єктно-орієнтований підхід, об'єктно-орієнтоване проектування та об'єктно-орієнтоване програмування. Можна сказати, що визначальною рисою є об’єктна-орієнтованість, яка може мати місце при створенні, аналізі та супроводженні різноманітних систем (і не тільки програмних).

У навчальному посібнику викладені матеріали лекційних занять для дисципліни «Об'єктно-орієнтоване програмування». Викладено основні відомості щодо впровадження об'єктно-орієнтованого підходу для розробки програм на основі мови програмування С++.

Одною з проблем створення та підтримки програмного забезпечення є поріг складності проекту, починаючи з якого стає дуже важко виправляти помилки та неточності, імплементувати нові вимоги щодо нарощування можливостей системи, розширювати функціонал тощо. У даному курсі значну увагу приділено модульності програмних систем, забезпеченню низької зв’язаності та взаємозалежності окремих елементів цих систем.

Значним кроком у теорії та практиці програмного забезпечення стала класифікація архітектурних патернів об’єктно-орієнтованого проектування. Патернам проектування присвячено декілька останніх лекцій цього курсу.

Викладення матеріалу не обмежується тільки ООП на С++. В ході викладення порівнюються особливості реалізації ООП на різних мовах програмування. Також висвітлюються деякі обмеження та проблеми ООП, зокрема, “*expression problem*”.

Лекційний матеріал охоплює не тільки теоретичні, а й практичні аспекти ООП. Одне із завдань, яке поставив перед собою автор – навчити студентів створювати повноцінні програмні додатки, зокрема, з графічним інтерфейсом користувача. Тому окрему лекцію присвячено розгляду особливостей операційної системи Windows (яку саму по собі можна вважати об'єктно-орієнтованою системою) та створенню додатків для цієї ОС.

Навчальний посібник призначений для здобувачів ступеня бакалавра за освітньою програмою «Інженерія програмного забезпечення комп'ютерних систем»» спеціальності 121 «Інженерія програмного забезпечення». Посібник може бути також корисним студентам, які вивчають програмування за іншими спеціальностями.

Лекція 1

Тема 1.1. Огляд основних понять розробки програмного забезпечення

Основні поняття

Об’єктно-орієнтоване програмування (ООП) – є одним з підходів, який можливо використати при розробці програмного забезпечення.

Зазвичай розрізнюють:

- об’єктно-орієнтоване **програмування** (*object-oriented programming, OOP*)
- об’єктно-орієнтоване **проектування** (*object-oriented design, OOD*).

Дисципліна ООП в теперішній час стала вже класичною, академічною дисципліною навчання фахівців з розробки програмного забезпечення. В даному курсі ми розглянемо особливості як об’єктно-орієнтованого програмування так і об’єктно-орієнтованого проектування

Місце та роль програмного забезпечення

Програмою ми будемо називати втілення алгоритмів, що змушує комп'ютер виконувати необхідні дії.

Алгоритм – це кінцева послідовність дій, яка призводить до потрібного результату.

Операційна система (ОС) – це програма або множина програм, яка керує ресурсами комп'ютера та іншими програмами. Зазвичай ОС складається з модулів – множини системних програм та бібліотек.

Прикладна програма – це програма, яка працює під керуванням операційної системи, використовує її сервіс.

Програмне забезпечення (ПЗ) (англ. *software*) – набір програм, які можуть бути встановлені в комп'ютерній системі користувачем цієї системи. Можна виділити такі типи ПЗ: системне, прикладне, інструментальне.

Мікропрограмне забезпечення (англ. *firmware*) – постійне або напівпостійне програмне забезпечення, для встановлення та зміни якого потрібні спеціальні знання про систему, значні зусилля та майстерність.

Апаратне забезпечення (англ. *hardware*) – фізичні, апаратні компоненти системи. Зазвичай вони є найбільш постійними і незмінними, проте деякі комп’ютерні системи дозволяють відносно легко змінювати апаратні компоненти.

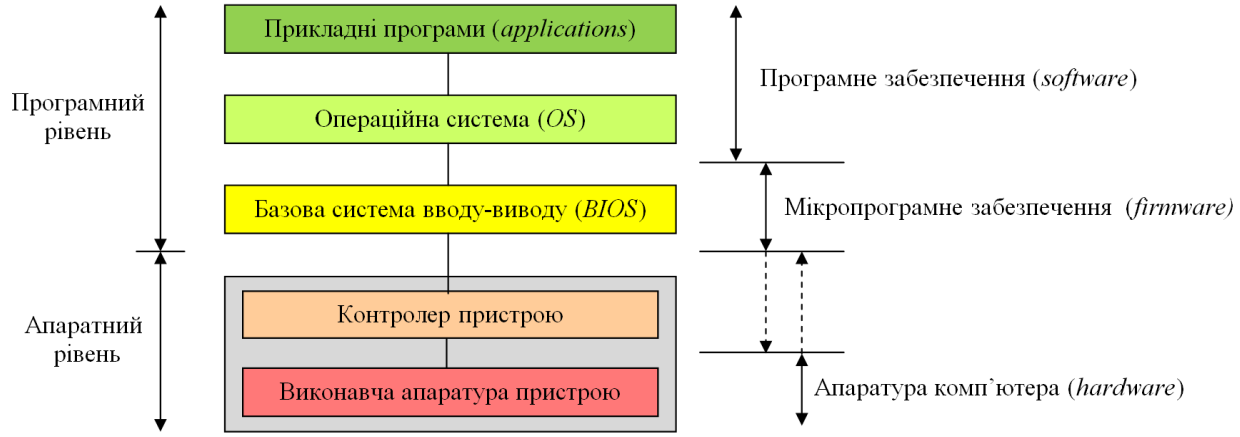


Рис. 1.1. Ієрархія взаємодії програмного забезпечення та апаратури

Основними функціями операційної системи будемо вважати керування ресурсами комп'ютера і надання можливостей користувачу та прикладним програмам використовувати ці ресурси.

В даний час найбільше поширення одержали дискові операційні системи. Для них характерним є те, що більшість необхідних для роботи комп'ютера програм і даних зберігається на диску у вигляді **файлів**. Операційна система являє собою набір програмних модулів, що читаються з файлів у міру необхідності. До таких модулів відносяться бібліотеки підтримки, драйвери пристроїв і інші. Та частина операційної системи, що повинна постійно знаходитися в оперативній пам'яті (ОП), називається її **ядром**.

Операційна система також дає користувачу комп'ютера свій інтерфейс для настроювання системи, роботи з файлами, для запуску прикладних програм і інших операцій. Цей інтерфейс в деяких ОС реалізований на основі **командної мови,** а в інших ОС - на основі **графічного інтерфейсу користувача** GUI (*Graphical User Interface*). Графічний інтерфейс є сучаснішим та зручнішим.

Прикладні програми зазвичай зберігаються у вигляді файлів. У комп'ютерах типу IBM PC під керуванням ОС Windows прикладні програми зберігаються у вигляді файлів із розширенням EXE. При запуску прикладної програми ОС забезпечує завантаження програми з диска в оперативну пам'ять. Після завантаження всієї програми або необхідної її частини в ОП, ОС передає процесору адресу точки входу. До цього часу програма повинна бути розгорнута в ОП – як інструкції так і дані у вигляді машинних кодів. Під час виконання програми процесор зчитує чергову інструкцію, виконує її і переходить до чергової інструкції.

Прикладна програма в ході свого виконання має необхідність у використанні ресурсів комп’ютера - для виводу на екран, для читання - запису файлів, для роботи з масивами в пам'яті, для введення з клавіатури або миші і т.д. Щоб відокремити прикладну програму від безпосереднього доступу до ресурсів комп'ютера, ОС надає **інтерфейс API** (*Application Program Interface*). Цей набір функцій і повинна використовувати прикладна програма для усіх своїх потреб.

Розробка програмного забезпечення

При розробці програм використовуються різні інструментальні засоби – зазвичай це програми. Програми, призначені для розробки інших програм, працюють на комп'ютерах, аналогічних тим комп'ютерам, на яких буде виконуватися розроблена програма. Або ж на комп'ютерах класом вище. Якщо програма, що призначена для роботи на комп’ютері одного типу, створюється на комп’ютері принципово іншого типу, то вживають термін "кросплатформенна" (*cross platform*) розробка.

Програма в комп'ютері зазвичай являє собою файл. Головний результат роботи програміста – в**иконуваний файл** (*executed file*). Це двійковий файл, що містить машинні коди, тобто набір команд і даних згідно із прийнятою системою команд процесора. Такий файл при виконанні програми завантажується операційною системою у оперативну пам’ять без істотних змін. У персональному комп'ютері виконувані файли зазвичай мають розширення EXE. **Файли вихідних текстів** (*source code англ.*) є сукупністю рядків, написаних зрозумілішою для людини мовою, аніж машинні коди. Такі файли не можуть бути безпосередньо завантажені на виконання операційною системою – їх потрібно обробити деякою програмою.

Інтерпретатори та компілятори

Інтерпретатор - це програма, яка сприймає файл вихідного тексту програми і відповідно виконує усі потрібні функції. У цьому сенсі програма-інтерпретатор є процесором для файлів вихідного тексту.

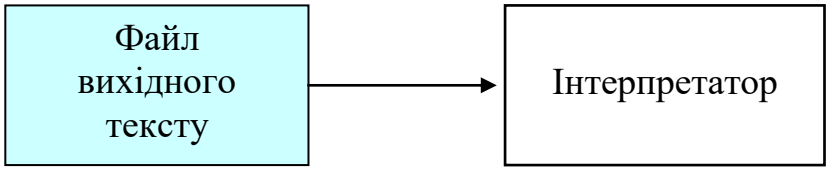


Рис. 1.2. Ілюстрація інтерпретації

Найпростіший варіант: інтерпретатор зчитує один рядок тексту програми, виконує його, потім зчитує наступний рядок, виконує його і так далі. Так працював BASIC на перших персональних комп'ютерах сімейства IBM PC.

Взагалі можна сказати, що розвинена програма повинна вміти працювати у режимі інтерпретатора. Це дозволяє досягти гнучкості, ефективності використання програми, якщо послідовність виконання декількох операцій можна описати деяким текстом, який буде відпрацьовуватися, можливо, і в пакетному режимі. Інтерпретатором може бути як системна так і прикладна програма, наприклад, СУБД, офісний пакет (текстовий редактор, електронна таблиця, база даних), web-браузер, геоінформаційна система, пакет CAD тощо.

Усі розвинені програми дозволяють інтерпретувати програмні файли, написані відповідною мовою – такі файли звуться "скриптами", макросами тощо. Часто трапляється так, що можливості, які надаються через інтерфейс програмування є більшими, аніж можливості, які доступні через графічний інтерфейс користувача програми.

Компілятор - це програма, що перетворює файли вихідного тексту у виконуваний файл. Як приклад розглянемо роботу компілятора мови C++. Зазвичай компіляція робиться, як мінімум, у два етапи:

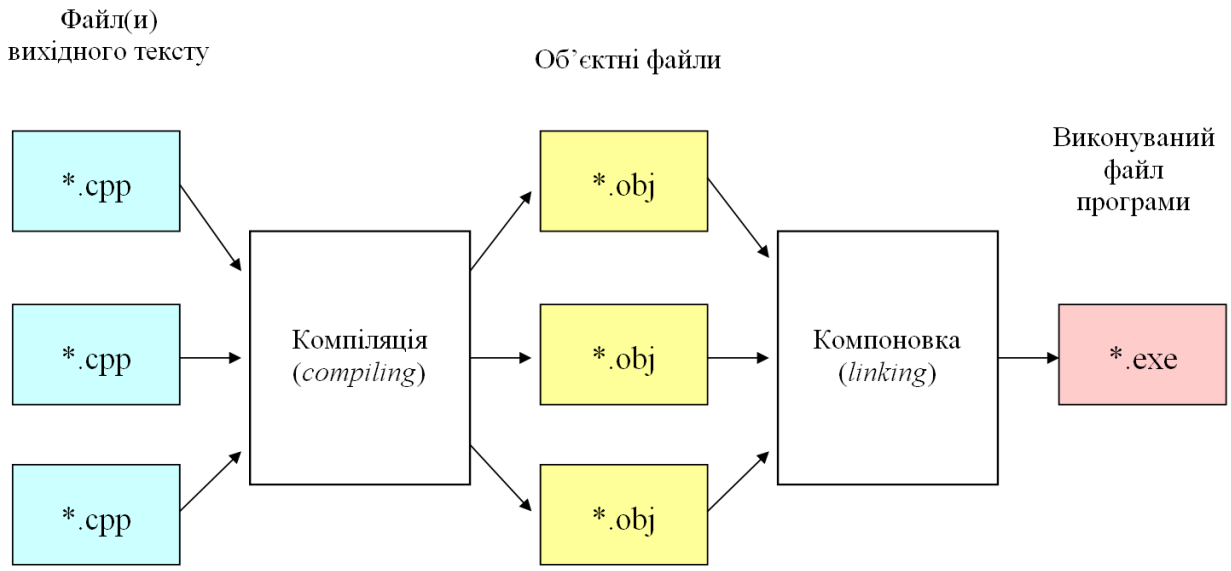


Рис. 1.3. Робота компілятора C++

На першому етапі виконується власне компіляція - перетворення файлів вихідного тексту в об'єктні файли. Об'єктний файл є відтворенням програми або модуля в машинних кодах, однак, він не є виконуваним файлом, а деяким напівфабрикатом. Потім Linker зв’язує об'єктні файли усіх модулів програми

та бібліотек і записує один виконуваний exe-файл. Така двохступінчата схема дозволяє спростити розробку складних програм. Зокрема, велика по об'єму вихідного тексту програма може бути розчленована на декілька програмних модулів. Кожний модуль може компілюватися окремо. При розробці програм на основі модульного принципу досягається прискорення компіляції за рахунок того, що при внесенні змін в окремий модуль перекомпілюється тільки він.

Компіляція і бібліотеки. Повторне використання програмного коду

Часто вихідні тексти пишуться в розрахунку на використання функцій, що є в складі бібліотек.

Статичне зв'язування програмних кодів. Можуть бути використані бібліотеки, що розроблені для конкретної мови програмування. Такі бібліотеки зберігається у файлах *. lib і у результаті компонування машинні коди бібліотечних функцій записуються у виконуваний файл *.exe. Такий спосіб зв'язування програмних кодів називається статичним.

Динамічне зв'язування програмних кодів. Програма, розроблена, наприклад, для ОС Windows, представляється у виді виконуваного файлу ***.exe** і одного або декількох файлів бібліотек DLL (*.dll). Файли DLL (*Dynamic Link Library*) можуть містити в собі програмні коди функцій, на які посилається основний програмний модуль (*.exe). При виклику функції, що знаходиться в DLL, читається відповідний файл *.dll і програмний код завантажується в ОП. Використання бібліотек DLL дає можливість заощадити пам'ять - як дискову, так і ОП за рахунок того, що

- необхідні програмні коди завантажуються в пам'ять не одразу для усіх бібліотек DLL, а тільки тих, що потрібні у даний момент;
- декілька програм, що працюють у багатозадачному захищеному режимі можуть використовувати ту саму DLL, причому вона завантажується у пам'ять в єдиному екземплярі.

Сама операційна система Windows складається з великого числа різноманітних DLL.

Загалом, бібліотеки являють собою засіб повторного використання програмного коду.

Поєднання інтерпретації та компіляції

Створення програм у вигляді виконуваних файлів, які містять машинні коди процесора, дозволяє досягти найбільшої швидкості роботи програм. Виконуваний файл створюється традиційною технологією компілювання-

лінкування. Завантаження з виконуваного файлу (*.exe) двійкових кодів команд та даних в оперативну пам'ять робить операційна система.

Існує багато проблем та ризиків при створенні програм людьми, зокрема можна вказати наступні моменти:

- програма фактично прив’язана до відповідного типу процесора, машинний код не може виконуватися на комп’ютерних платформах з іншими процесорами. Тому для інших платформ доводиться або просто перекомпільовувати або навіть і переписувати тексти програм.
- людям властиво робити помилки, і програмістам також. Кожна помилка може бути загрозою не тільки неправильної роботи окремої програми та втрати даних, а й може призвести до пошкодження комп’ютерної системи.

Можна сказати, що виконання програм у режимі інтерпретації має менше ризиків – програма інтерпретатор виконує тільки дозволені функції і повинна гарантувати їхнє безпечне виконання. Крім того, сам текст програми можна зробити однаковим і незалежним від процесорної платформи, потрібно тільки, щоб в комп’ютерній системі була б інстальована відповідна програма інтерпретатор (віртуальна машина).

Поєднання компіляції та інтерпретації використовується в таких технологіях, як Java та .NET Framework

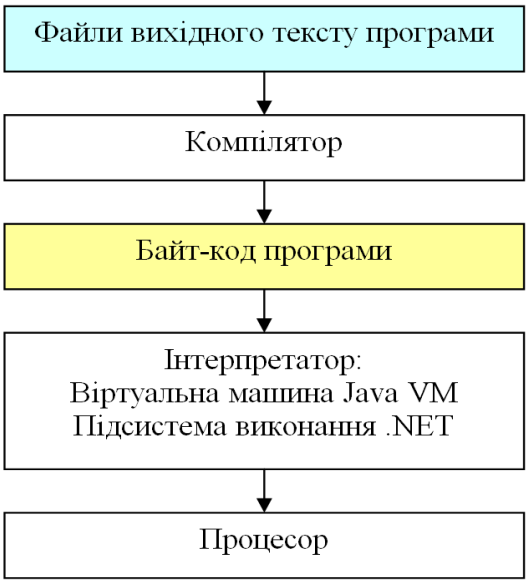


Рис. 1.4. Поєднання компіляції та інтерпретації

У результаті компіляції-лінкування тут створюється не машинний код для процесора, а так званий байт-код для віртуальної машини-інтерпретатора (JIT – *Just In Time compiler*). Швидкодія роботи суттєво залежіть від віртуальної машини, проте, оскільки вона є посередником між процесором,

то втрачається деякий час і завжди такі програми виконуються повільніше, аніж скомпільований машинний код, завантажений у пам'ять і виконуємий безпосередньо процесором.

Прикладом еволюції у співвідношенні інтерпртації та компіляції може слугувати ОС Android. Перші версії цієї ОС використовували віртуальну машину Java і JIT-компілятор. Починаючи з 5-ї версії додано додаткову компіляцію-оптимізацію, яка робиться при інсталяції файлів додатків *.apk.

Контрольні запитання

1. Що таке програмне забезпечення? Яка його роль?
2. Що таке операційна система? Які функції вона виконує? Які ви знаєте операційні системи?
3. Що таке прикладна програма?
4. Навіщо потрібні файли вихідного тексту програм?
5. Що виконує програма-інтерпретатор?
6. Які етапи роботи компілятора C++?
7. Яким чином може забезпечуватися повторне використання програмного коду?
8. Чим відрізняється динамічне зв’язування бібліотек від статичного?
9. Що таке віртуальна машина? Наведіть приклади
10. Що таке виконуваний файл і як його створити?

Тема 2.1. Базові елементи мови C++

Вступ до C++

Мова C++ є подальшим розвитком мови програмування C, розробленої Деннісом Рітчі (*Dennis Ritchie*) у 1971 році. Мова C й дотепер широко використовується для написання прикладних програм та операційних систем. Денніс Рітчі також був ключовим розробником операційної системи UNIX разом із Кеном Томпсоном (*Ken Thompson*). Операційна система UNIX написана на C. Основні ідеї та принципи UNIX активно використовуються і дотепер, втілені в сучасних операційних системах.

Автором мови C++ є Б'ярн Страуструп (*Bjarne Stroustrup*), який досить скромно назвав C++ "удосконаленим C".

Найпростіші програми на мовах C, C++

Існує традиція при порівнянні мов програмування наводити вихідний текст найпростішої програми. Найпростіша програма повинна виводити на монітор єдиний рядок Hello, World!

Найпростіша программа на C

```c
#include "stdio.h"

int main(void)
{
   printf("Hello, World!");

   return 0;
}
```

Найпростіша программа на C++

```cpp
#include <iostream>

using namespace std;

int main()
{
   cout << "Hello, World!";

   return 0;
}
```

Як видно з наведених прикладів, у вихідному тектсі програми має бути, як мінімум, головна функція – вона зветься main і з неї починається виконання програми. У фігурних дужках головної функції записуються рядки-інструкції, які і визначають послідовність виконання програми від початку. Завершує виконання головної функції у данному прикладі оператор return, який передає назовні цілочисельне значення – його сприймає операційна система, яка завершує роботу программного застосунку і звільняє виділені для нього ресурси.

Прості типи

Прості типи – це стандартні типи даних, які підтримуються в усіх реалізаціях мов програмування С та С++.

Цілі типи

Тип	Байтів в пам'яті	Діапазон значень
char	1 байт	від -128 до 127
unsigned char	1 байт	від 0 до 255
short	2 байти	від -32768 до 32767
unsigned short	2 байти	від 0 до 65535
int	залежить від реалізації	
unsigned int	залежить від реалізації	
long	4 байти	від -2 147 483 648 до 2 147 483 647
unsigned long	4 байти	від 0 до 4 294 967 295

Примітка. Параметри типів **int** і **unsigned int** визначаються довжиною машинного слова процесора, операційною системою і режимом компіляції.

Для компіляторів 16-розрядних програм в 16-бітових ОС, наприклад, MS-DOS, тип **int** тотожній short

Для компіляторів 32-, 64-розрядних програм в 32-, 64-бітових ОС, наприклад, Windows, тип **int** тотожній **long**.

Типи з плаваючою точкою

Тип	Байтів в пам'яті	Діапазон значень
float	4 байти	$\pm$ (від $3.4 \cdot 10^{-38}$ до $3.4 \cdot 10^{+38}$), а також нуль
double	8 байт	$\pm$ (від $1.7 \cdot 10^{-308}$ до $1.7 \cdot 10^{+308}$), а також нуль
long double	10 байтів	$\pm$ (від $3.4 \cdot 10^{-4932}$ до $1.1 \cdot 10^{+4932}$), а також нуль

Найширший діапазон значень у **long double,** найвужчий – у **float**

Найбільшу точність представлення чисел забезпечує **long double**, найменшу – **float**

Ключове слово void

Ключове слово **void** використовується у наступних випадках:

1. При оголошенні та визначенні типу функції слово **void** означає, що функція не повертає значень.

2. Слово **void** у списку параметрів функції означає, що функція не має параметрів.

3. В оголощенні вказівника слово **void** означає, що цей вказівник універсальний, для будь-якого типу (проте не може бути вказівником на функцію)

Специфікатор typedef

Означає перейменування типів. Може призначати нові імена-синоніми простим (стандартними) типам, а також типам, створеним користувачем.

```
typedef type-declaration synonym;
```

Приклади

```
typedef char CHAR;
typedef char *PSTR;

typedef char CHAR, *PSTR;           //2 в один рядок

typedef unsigned long ULONG;
typedef long *PLONG;
```

Типи, які створюються користувачем

Перечислимий тип enum

```
enum <ім'я типу> {< список елементів >};
```

Таке оголошення створює новий цілий тип, що має обмежений набір значень у вигляді поіменованих констант. Кожний елемент у списку значень за замовчуванням ініціалізується 0, 1, .. і т.д. Але елементи списку можуть бути явно ініціалізовані. Наприклад:

```
enum State {start, run, stop};
enum Colors {red, green, blue, white};
enum Position {start = 1, mid = 3, end = 99};
```

Структура

Структура дозволяє об'єднати в однім об'єкті сукупність перемінних (полів), що можуть мати різні типи. Окрім перемінних, елементами структур можуть бути також і функції.

```
struct <ім'я типу> {<список оголошень полів>};
```

Над структурою, як єдиним цілим, можуть виконуватися такі дії:

- присвоювання структур, якщо структури мають однаковий тип;
- передача функції в якості аргументу;
- повернення в якості значення функції;
- одержання адреси структури.

Структура займає в пам'яті число байтів, більше або рівне сумарній довжині поля. Більше - якщо використовується вирівнювання полів на межу слів для полів менше слова. Це залежить від режиму компіляції програми. Зазвичай це можна визначати настроюванням компілятора.

Приклади структур

```
struct Point
{
double x, y, z;
};

struct Point4d
{
long double time;
Point koord;       //поле koord  є структурою
};

Point4d A, B;      //оголошення перемінних A та B

A.time = 2016;     //присвоювання окремим полям
A.koord.x = 2.5;
A.koord.y = 10;
A.koord.z = -1;
B = A;             //присвоювання структури цілком
```

Об'єднання union

```
union <ім'я типу> {<список елементів>};
```

Об'єднання в кожний момент часу може берегти значення тільки одного елемента. Контроль за тим, якого типу елемент зберігається в даний

момент в об'єднанні, покладається на програміста. Об'єм пам'яті, що виділяється для об'єднання, визначається найдовшим елементом. Наприклад:

```
union Number
{
int svar;
unsigned int uvar;
};
```

Масив

```
<специфікація типу> <ім'я масиву> [<константне вираження>];
```

```
<специфікація типу> <ім'я масиву> [ ];
```

Масив зберігає як єдине ціле послідовність елементів однакового типу, наприклад:

```
int X[2000];
```

Специфікація типу може відповідати будь-якому типу, крім **void** і функції. Таким чином, елементи масивів можуть мати цілий, дробовий, перечислимий тип, можуть бути структурами, об'єднаннями, покажчиками. Індексація елементів масиву починається з 0. Константне вираження в квадратних скобках може бути опущено. Це припустимо, коли в оголошенні присутня ініціалізація, або коли масив об'являється як формальний параметр, або дане оголошення є посиланням на оголошення масиву десь в іншому місці програми. Приклади:

```
char tost [ ] = ''Будьмо ! '';  //оголошення з ініціалізацією
int XY[50][8];                  //двовимірний масив
int a[5] = {-1, 2, 0, 5, 99};
int b[3][2] = {{1, 2}, {3, 4}, {5, 6}};
```

Ім'я масиву саме по собі є адресою першого елемента цього масиву:

```
v = &v[0];
```

за виключенням використанням в операції **sizeof**. Наприклад, для

```
long c[10];
```

операція **sizeof** (c) дає 40 байт.

Покажчик

Покажчик, вказівник (*pointer* англ.) - це перемінна, що містить адресу деякого об'єкта, наприклад, перемінної або функції.

```
<специфікація типу> * <ім'я покажчика>;
```

Покажчик на функцію містить адресу точки входу у функцію. Приклади покажчиків:

```
int *lp;
char *lpstr;
```

Значення покажчика, що дорівнює 0 використовується для контролю. Гарантується, що будь-який об'єкт не може мати нульову адресу. Деякі функції можуть повертати нульовий покажчик як ознаку неможливості або помилкового виконання заданих дій.

Функції

Функції являють собою спеціально оформлені фрагменти програмного коду. Основне призначення функцій - структурування програм. Якщо в програмі зустрічаються два й більше однакових фрагментів коду, то уведення функцій зменшує вихідний текст та машинний код.

Для функцій розрізняють такі етапи:

- оголошення
- визначення
- виклик

У мовах C та C++ усі функції повинні бути оголошені або визначені **перед** їхнім викликом. Не допускається оголошення або визначення функції усередині визначення іншої функції.

Оголошення функції

```
<специфікація типу> <ім'я функції> (<список аргументів>);
```

- оголошення визначає ім'я функції, тип значення, яке функція повертає, та типи аргументів. В оголошенні імена аргументів можуть не вказуватися

- функція може бути перевантажена, тобто те саме ім'я можуть мати декілька функцій із різними типами аргументів і типами значення, що повертається

Приклад оголошення функцій

```
double Max (double a, double b);

int Max (int a, int b);        //перевантаження функції Max
```

Визначення функції

```
<специфікація типу> <ім'я функції> (<список аргументів>)
{
. . .       //тіло функції
}
```

Визначення описує код, що виконується при виклику функції.

Може бути зроблено **inline** для оптимізації виклику функцій. Як правило, визначення функцій **inline** міститься у файли заголовків і при компіляції код тіла функції безпосередньо записується замість виклику функції. Це дає економію часу виконання програми - не потрібні зайві операції для передачі значень аргументів.

Для функцій, що мають тип значення що повертається, повинен бути хоча б один оператор **return**. Для мови C/C++ усередині тіла функції не може бути оголошення й визначення іншої функції.

Приклад визначення

```
double Max (double a, double b)
{
return a > b ? a : b;
}
```

Виклик функції

Призводить до виконання коду, даного у визначенні функції. Може бути зроблений двома способами: прямо - по імені, або через покажчик.

Приклади викликів функції Max.

Найпростішим є синтаксис виклику функції по імені

```
c = Max(x,y);        //прямий виклик функції Max
```

Дещо екзотичним виглядає виклик функції через покажчик:

```
int (*pf)(int,int);  //pf - покажчик на будь-яку функцію
                     //з двома аргументами
pf = Max;            //pf отримує адресу функції Max
c = (*pf)(x,y);      //виклик Max(x,y) через покажчик pf
```

Функція main()

Це головна функція програми.

- функція, із якої починається виконання програми (точка входу);
- не може бути перевантажена, не може мати аргументів за замовчуванням;
- не може одержати свою адресу, не може бути оголошена inline або static;
- присутня у програмі тільки у вигляді визначення.
- необхідна всім програмам на C та C++. Проте деякі реалізації використовують інше ім'я – наприклад, WinMain у програмах для Windows.

```
int main(int argc, char *argv[])
{
. . .  //тіло програми
}
```

Аргументи main

argc – кількість аргументів, заданих при виклику програми;

***argv[]** – масив покажчиків на char, який зберігає слова командного рядка:

argv[0] – ім'я, що використовувалося при виклику програми,

argv[1], ..., argv[argc-1] – аргументи з командного рядка.

Контрольні запитання

1. Які сфери застосування мов програмування С та С++?
2. Хто є автором мови С++?
3. Які складові містять найпростіші програми на мовах С та С++?
4. Які є прості типи в мові С++?
5. Які є типи в С++ для представлення цілих чисел?
6. Який тип забезпечує найбільший діапазон представлення цілих чисел?
7. Які є типи в С++ для представлення дробових чисел?
8. Якій тип в С++ забезпечує найбільший діапазон та найвищу точність представлення чисел у форматі з плаваючою точкою?
9. Що означає тип void?
10. Які типи засобами мови С++ може створити користувач ?
11. Що може дати використання enum?
12. Що означає ключове слово struct?
13. Як можна оголосити масив?
14. Що таке покажчик?
15. Навіщо потрібне оголошення функції?
16. Що містить визначення функції?
17. Що таке функція main і яка її роль?
18. Які аргументи функції main?

Лекція 2

Тема 2.2. Препроцесор C++. Модульність програм

Препроцесор C++

Препроцесор – це частина компілятора, яка виконує команди попередньої обробки вихідних текстів до початку трансляції в об’єктний код. Такі команди називаються директивами препроцесора (*preprocessor directives*). Усі вони розпочинаються з символа '#'.

#define
#include
#if
#else
#endif
#ifdef
#ifndef
#pragma
та інші.

Препроцесор можна використовувати для умовної компіляції, вставки файлів, створення повідомлень для помилок компіляції, а також для застосування до розділів коду правил, які залежать від типу комп'ютера.

Директиви препроцесора, такі як #define і #ifdef, як правило, використовуються для того, щоб вихідні тексти програм було легко змінити і легко зібрати в різних середовищах виконання. Директиви в вихідному файлі вказують препроцесору виконувати певні дії. Наприклад, препроцесор може замінити маркери в тексті, вставити вміст інших файлів в вихідний файл, або блокувати компіляцію частини файлу шляхом видалення розділів тексту.

Директива #define

```
#define identifier  token-string необовязково
```

Директива **#define** вказує компілятору, що у вихідному тексті необхідно підставити рядок **token-string** замість кожного ідентифікатора **identifier**, Директиви **#define** розташовують зазвичай у початку файлу вихідного тексту, або записують у окремих заголовочних файлах, які підключають директивами **#include.** Приклад

```
#define COLOR_RED 9
. . .
if (clr == COLOR_RED) MessageBox(hWnd,"А чому червоний?","Який жах",MB_OK);
```

Ідентифікатор **identifier** замінюється, тільки якщо він формує токен. Це означає, що **identifier** не замінюється, якщо міститься у комментарі, рядку або є частиною іншого довгішого ідентифікатора. Аргумент **token-string** складається із множини токенів, наприклад, ключових слів, констант або повних операторів. Аргумент **token-string** повинен відокремлюватися від **identifier** одним або декількома символів проміжку. Ці проміжки не вважаються частиною заміненого тексту, як і усі решта проміжків, записані за останнім токеном текста.

Директива **#define** без аргумента, наприклад

```
#define WIN32
```

вилучає входження **identifier** із вихідного тексту файла. Ідентифікатор, який визначено аргументом **identifier**, залишається дійсним і може бути перевірений за допомогою директив **#if defined** або **#ifdef**, наприклад

```
#ifdef WIN32
. . .           //якийсь код
#endif
```

Для директив **#define** є традиція обов'язково записувати ідентифікатор усіма ВЕЛИКИМИ_БУКВАМИ.

Директива **#define** часто використовується для створення символічних констант. Наприклад, у файлі **math.h** можна знайти визначення таких констант:

```
#define M_E         2.71828182845904523536
#define M_LOG2E     1.44269504088896340736
#define M_LOG10E    0.434294481903251827651
#define M_LN2       0.693147180559945309417
#define M_LN10      2.30258509299404568402
```

Інше застосування #define – макроси з параметрами. Наприклад, у файлі **windef.h** визначаються такі макроси

```
. . .
#define max(a,b)      (((a) > (b)) ? (a) : (b))
. . .
#define min(a,b)      (((a) < (b)) ? (a) : (b))
. . .
#define LOWORD(l)     ((WORD)(((DWORD_PTR)(l)) & 0xffff))
#define HIWORD(l)     ((WORD)((((DWORD_PTR)(l)) >> 16) & 0xffff))
#define LOBYTE(w)     ((BYTE)(((DWORD_PTR)(w)) & 0xff))
#define HIBYTE(w)     ((BYTE)((((DWORD_PTR)(w)) >> 8) & 0xff))
```

та багато інших. Використання таких макросів зовні схоже на виклик функцій:

```
result = max(x,y);
if (LOWORD(wParam) == IDOK)
   {
   . . .
   }
wmId    = LOWORD(wParam);
wmEvent = HIWORD(wParam);
```

Проте це не виклики функцій, а проста заміна тексту препроцесором. Замість імені макроса вставляється текст його визначення з урахуванням імен "аргументів" у дужках

Директива #include

Ця директива включає (а точніше кажучи, вставляє) у поточне місце вихідного тексту:

- вміст будь-якого файлу
- або стандартний заголовок

Приклад:

```
#include "myfile.ext"
```

Що це означає? Замість цього рядка вставляється увесь текст файлу **myfile.ext**

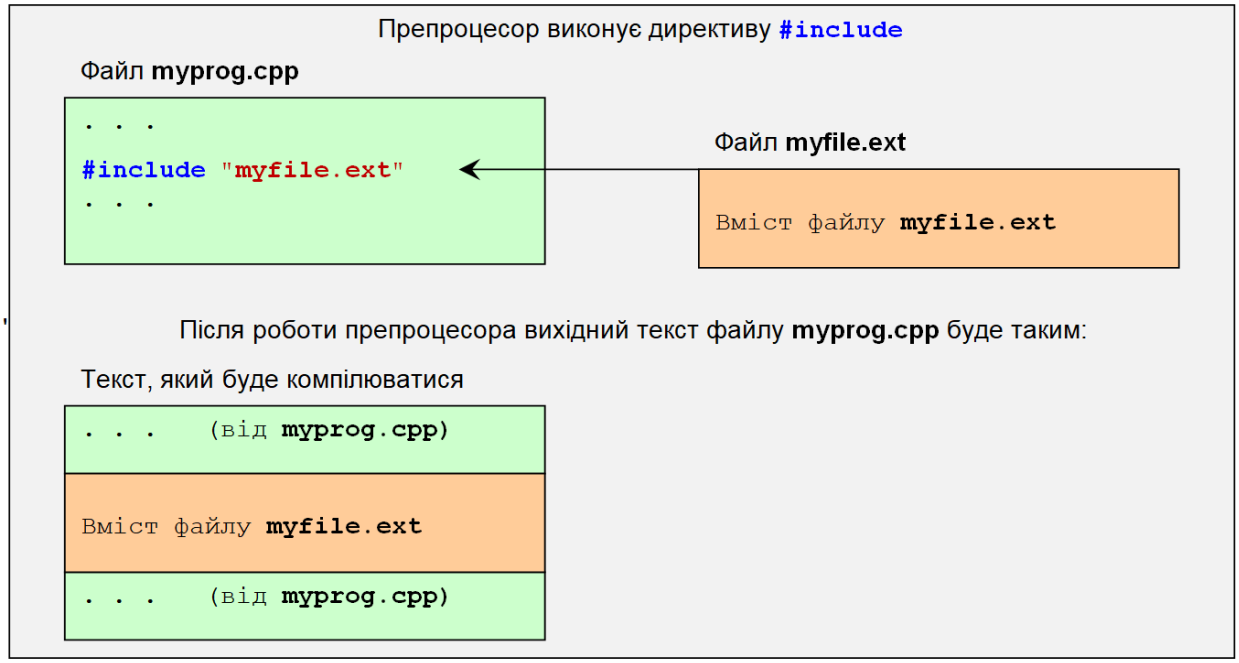


Рис. 2.1. Виконання директиви #include

Замкнене коло з ланцюжків #include

Препроцесор не може виконати такі директиви **#include**

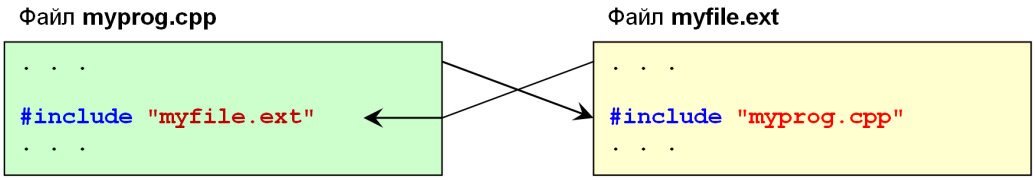


Рис. 2.2. Замкненн коло з #include

Компілятор C++ на такі "взаємовключення" реагує повідомленням про помилку. Звісно, це помилка програміста, який записав такі #include.

Вкладеність директив #include

Вкладеність означає, що у тексті файлу, який вставляється директивою #include, є також своя #include. Наприклад, у файлі **prog.cpp** є рядок:

```
#include "myfile.inc"
```

а у файлі **myfile.inc** є також рядок з #include, наприклад

```
#include "otherfile.h"
```

і так далі. Таку ланцюжкову вкладеність файлів можна відобразити так:

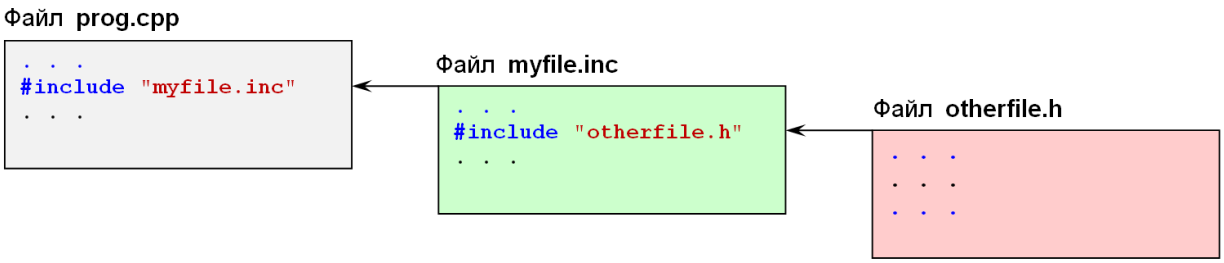


Рис. 2.3. Залежність файлів по #include

На схемі залежності файлів відповідно запису та виконанню директив #inclde можна констатувати ієрархію файлів-учасників.

Схема #include-ієрархії файлів

Можна схематично відобразити відношення включення файлів директивами #include. У такій схемі вказуються файли, які містять директиви #include, та файли, які включаються цими директивами.

На найнижчому рівні ієрархії розташовуються файли, які не містять #include. Наступний рівень посідають файли, у текстах яких записані #include файлів найнижчого рівня. І так далі.

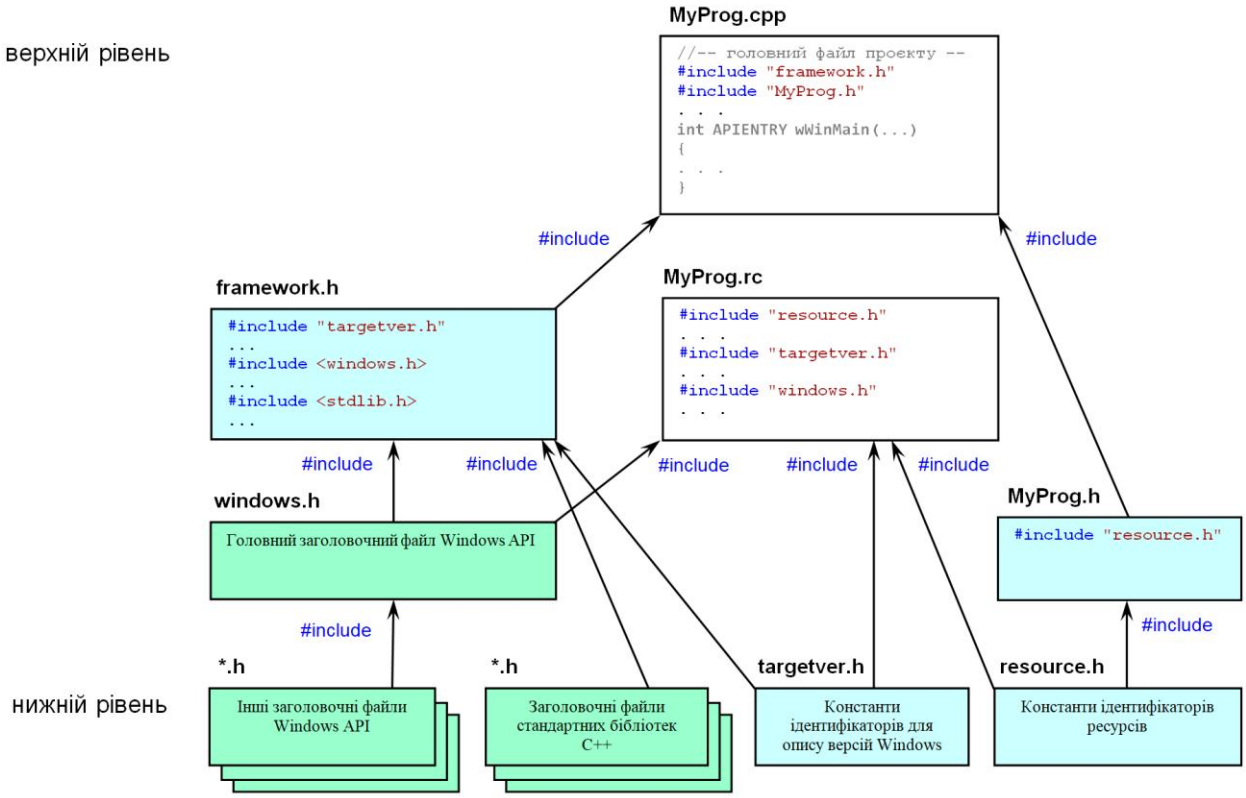


Рис.2.4. Приклад схеми #include-ієрархії для файлів найпростішого проекту Visual C++ Windows Desktop Application

Як можна помітити з вищенаведеної схеми, директиви #include записуються переважно для файлів з розширенням .h – так званих заголовкових файлів, хедерів (*headers*). Ще раз підкреслимо, що директива #include вставляє увесь вміст вказаного файлу і після цього робиться компіляція.

Модульність програм C++

Загалом, весь вихідний текст програми може бути записаний в один файл *.cpp (за винятком ще окремого файлу *.rc для проєктів для Windows). Таке можна уявити тільки для невеличких простих програм.

При вдосконаленні, додаванні нових можливостей вихідний текст, як правило, зростатиме. Починаючи з деякої кількості рядків, у вихідному тексті буде важко розбиратися, аналізувати, робити виправлення.

Модульність є ефективним шляхом розробки **складних** програм. Вихідний текст розподіляється по модулям, наприклад, відповідно основним функціям, або відповідно певним структурам даних. Засоби розробки

програм на C++, такі як Visual Studio, дозволяють легко створювати програми з модулів, які можуть окремо компілюватися. Перевагою модульності є зменшення складності об'єктів компіляції, можливість прискорити розробку шляхом **командної розробки** – кожний модуль може розроблятися та налагоджуватися окремо різними програмістами.

Структура вихідного тексту модуля C++

Вихідний текст кожного модуля складається, як мінімум, з двох файлів:

1. Основний файл *.cpp, наприклад, з ім'ям **module1.cpp**
2. Файл заголовку *.h, наприклад, **module1.h**

Також можуть бути інші файли, наприклад, для проектів для Windows:

3. Файл ресурсів *.rc, наприклад, **module1.rc** (це необов'язково, тільки у випадку, коли модуль окремо репрезентує якийсь елемент інтерфейсу користувача, наприклад, вікно діалогу тощо)
4. Інші файли

1. Основний файл вихідного тексту (module1.cpp) може містити оголошення-визначення перемінних, констант, визначення окремих функцій, визначення функцій-членів класів тощо. Приклад вмісту файлу **module1.cpp**:

```
#include "framework.h"
#include <math.h>
#include "module1.h"

//--оголошення-визначення глобальної перемінної--
int iResult_MOD1  = 0;

//--оголошення внутрішніх перемінних модуля--
static int i,n,size,counter;

//--оголошення внутрішніх функцій модуля–
static INT_PTR CALLBACK DlgProc_MOD1(HWND,UINT,WPARAM,LPARAM);
static void SomeFunc_MOD1(void);

//--визначення функції, яка експортується--
void MyFunc_MOD1(void)
{
. . .
}

//--визначення внутрішньої функції – функції діалогового вікна--
INT_PTR CALLBACK DlgProc_MOD1(HWND hDlg,UINT message,WPARAM wParam,LPARAM)
{
. . .
return 0;
}

//--визначення внутрішньої службової функції--
void SomeFunc_MOD1(void)
{
. . .
}
```

У наведеному вище прикладі перемінна **iResult_MOD1** та функція **MyFunc_MOD1()** будуть використовуватися в інших модулях. Це буде **інтерфейс модуля**. Крім того, у модулі є члени – перемінні та функції, які будуть використовуватися тільки для внутрішніх потреб модуля. Вони зовні невидимі, їхні імена недоступні для інших модулів програми – для цього вони оголошенні зі словом **static** попереду.

Файл module1.cpp потрібно включити у проект – він повинен бути у списку файлів Source Files вікна Solution Explorer. Тільки тоді цей файл буде компілюватися.

Примітка. У проектах на основі Windows API на початку тексту файлів *.cpp повинен бути рядок

```
#include "framework.h"
```

(у проектах Microsoft Visual Studio C++ до 2017 року замість **framework.h** був **stdafx.h**).

2. Файл заголовку модуля (module1.h) містить оголошення елементів, які будуть загальнодоступними – для використання у інших модулях програми. Іншими словами, описують інтерфейс модуля. Такі елементи позначаються словом **extern**.

Приклад вмісту файлу **module1.h**

```
#ifndef _MODULE1_H_
#define _MODULE1_H_

#define STATUS_ERROR_MOD1 -1
#define STATUS_GOOD_MOD1 1

extern int iResult_MOD1;
extern void MyFunc_MOD1(void);

#endif
```

або

```
#pragma once


#define STATUS_ERROR_MOD1 -1
#define STATUS_GOOD_MOD1 1

extern int iResult_MOD1;
extern void MyFunc_MOD1(void);
```

Для того, щоб при включенні заголовочного файлу директивою #include уникнути помилок компіляції повторного визначення, зокрема символічних констант STATUS_ERROR_MOD1 та STATUS_GOOD_MOD1, у файлі записано рядки:

```
#ifndef _MODULE1_H_
#define _MODULE1_H_

. . .

#endif
```

або

```
#pragma once


. . .
```

Рекомендується подібним чином оформлювати усі файли заголовків *.h.

Примітка. Директива **#pragma once** – специфічна для проектів Visual C++, може бути несумісною з іншими компіляторами.

Файл module1.h включати у проект не обов'язково (він буде автоматично завантажуватися директивою **#include**), проте бажано – тоді він буде у списку файлів Header Files вікна Solution Explorer і його буде легко викликати для огляду.

Рекомендація. Щоб не заплутатися у іменах ідентифікаторів і не використовувати щось на кшалт **namespace**, рекомендується імена **extern**-членів модуля **module1** робити унікальним. Наприклад, до кожного імені додавати наприкінці "_MOD1".

3. Файл ресурсів (module1.rc) містить опис елементів графічного інтерфейсу, наприклад, меню, діалогових вікон, та інших ресурсів. Цей файл потрібно включити у проект – він повинен бути у списку файлів Resource Files вікна Solution Explorer.

Поняття #include-ієрархії модулів програми

Незважаючи на те, що модуль може складатися з декількох файлів, ми його можемо розглядати як єдине ціле, вказуючи тільки зовнішни зв'язки (залежності). Цим відрізнятиметься #include-ієрархія модулів від #include-ієрархії файлів

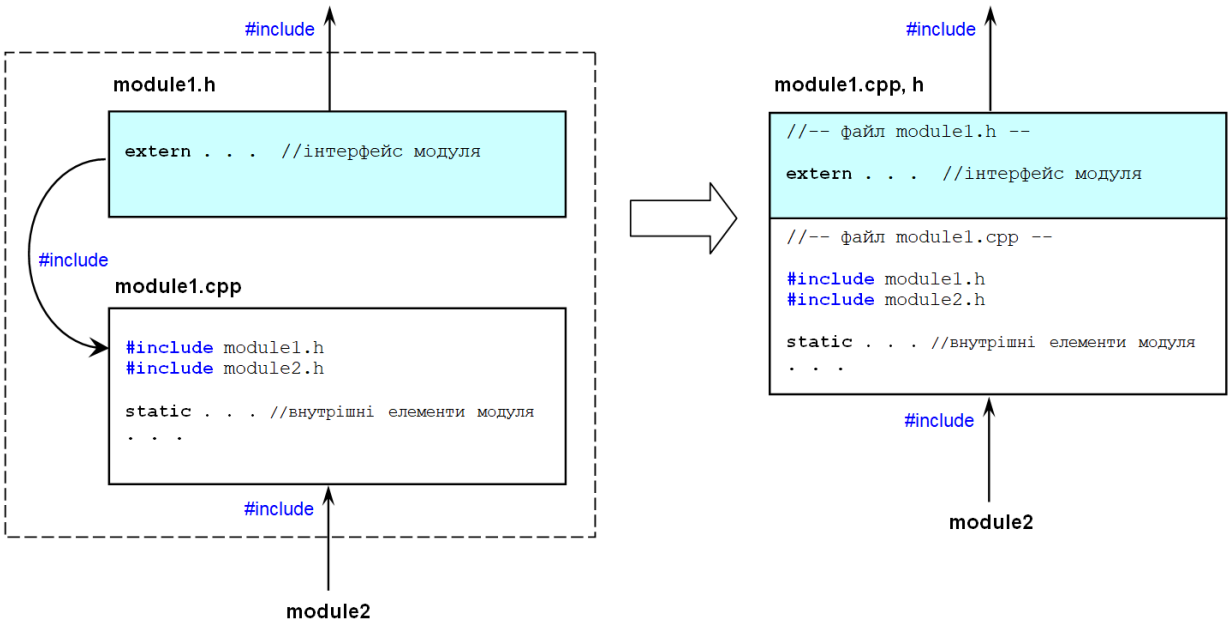


Рис. 2.5. Модуль з двох файлів

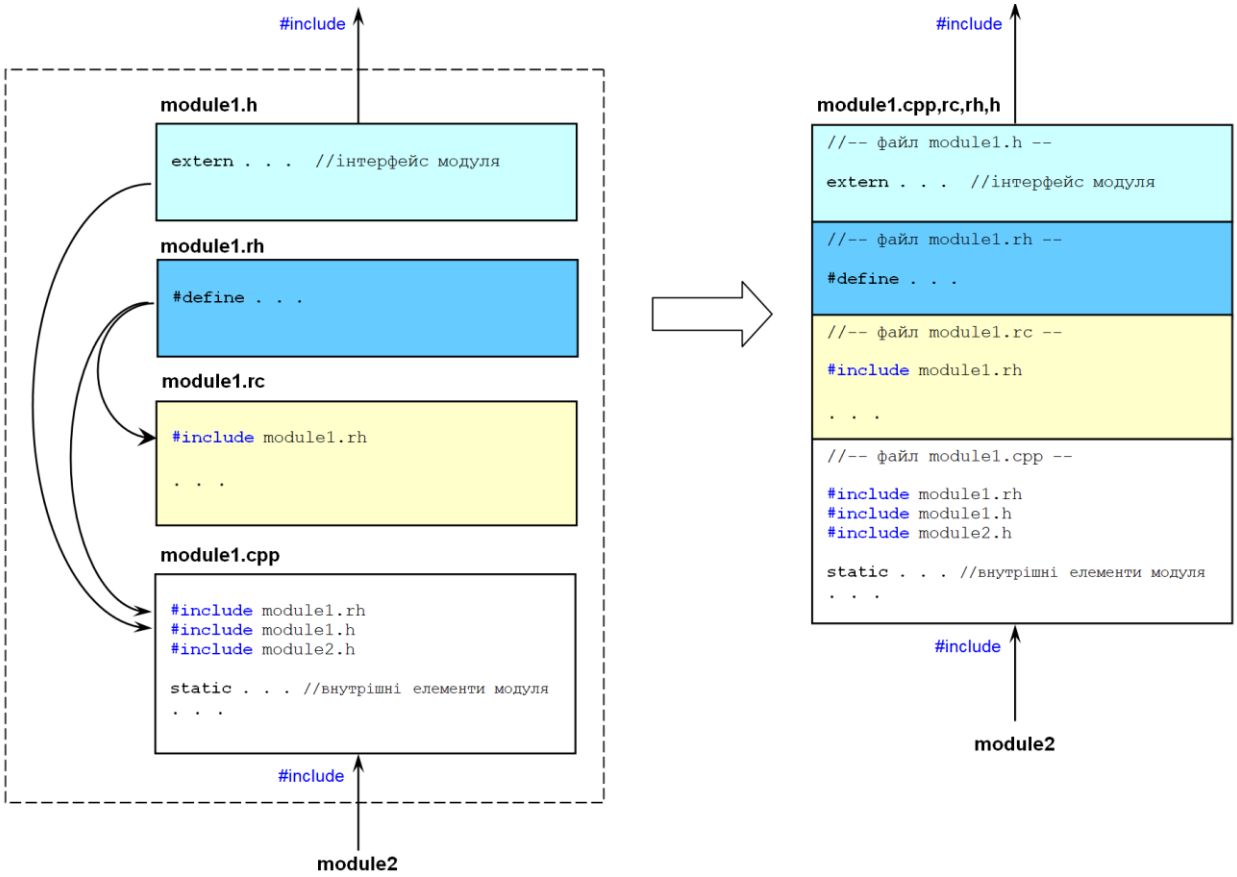


Рис. 2.6. Файли одного модуля програми на C++ Windows API

Стратегія модульності

Програма-моноліт – коли весь програмний код міститься у одному великому файлі. Це погано. Чим більше функцій виконує програма, тим більше розмір файлу. Чим більше розмір, чим складніше – тим гірше.

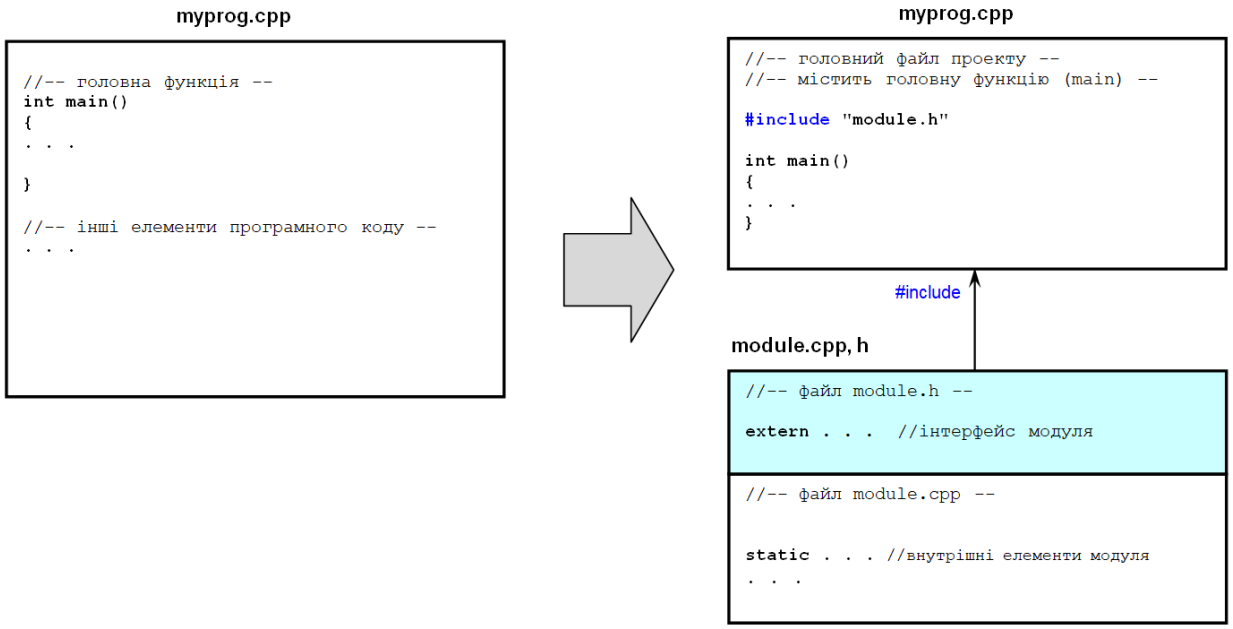


Рис. 2.7. Розділення коду програми-моноліту. Виділення модуля

Виділення модуля по функціональним ознакам, або з точки зору виокремлення певної структури даних, бажано зробити так, щоб цей модуль був самодостатнім і міг бути придатним й для інших проектів. Це буде означати повторне використання коду і є розумною стратегію роботи.

Можна вважати ідеальним випадок, коли усі модулі незалежні один від одного і посідають нижній рівень #include-ієрархії (за винятком головного файлу).

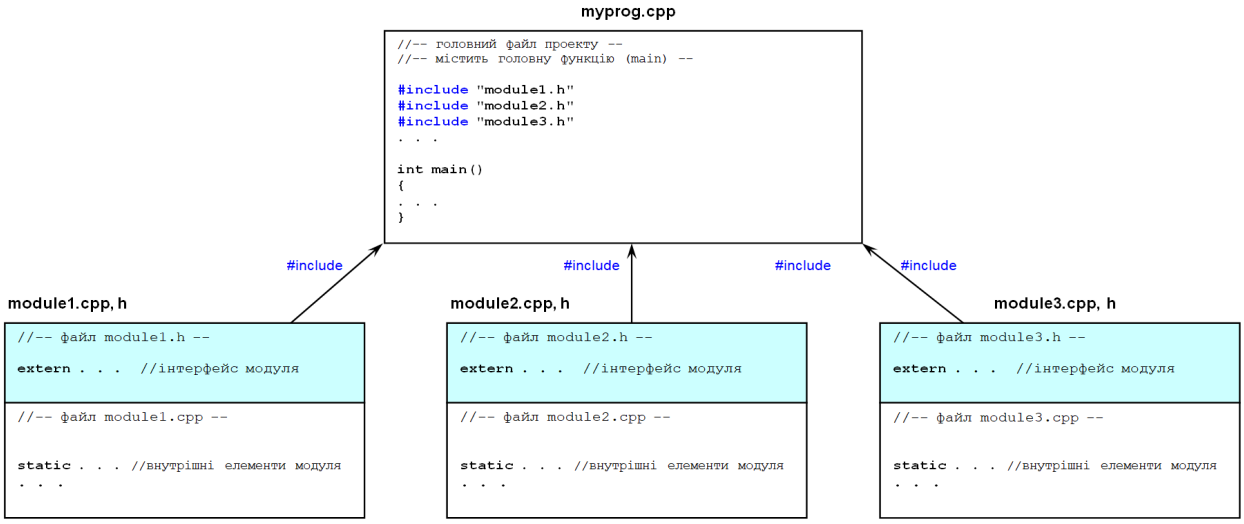


Рис. 2.8. Ідеальнй варіант модульності проєкта C++

Оскільки на практиці для написання модулів проекту зазвичай використовуються якісь бібліотеки, фреймворки, то це можна відобразити наступним чином

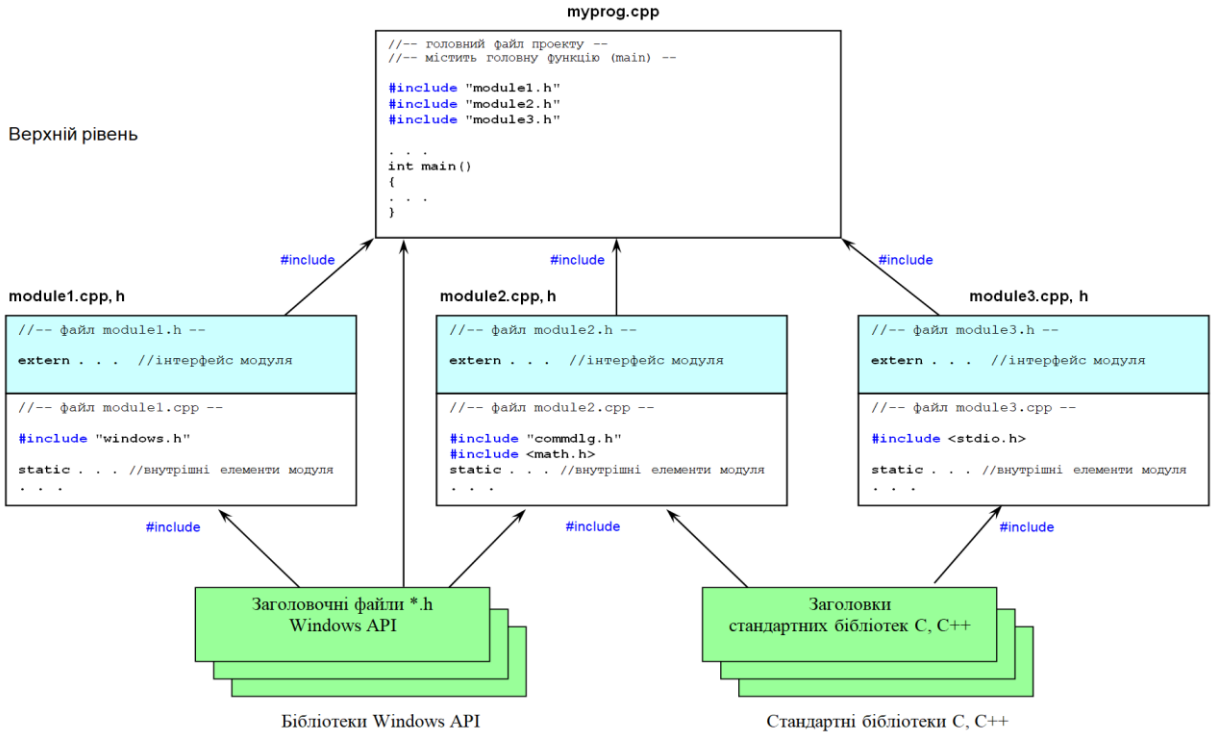


Рис. 2.9. Більш реалістичний варіант модульності

Якщо використовується якісь бібліотеки, фреймворки, то вони посідатимуть нижній рівень схеми залежностей (#include-єрархії) і модулі розробника будуть залежними від них. Але модулі розробника, в ідеалі, мають бути незалежні один від одного. Це стосується як вже наявних модулів розробників, так і модулів, які можуть ще розроблятися і додаватися у майбутньому при функціональному розширенні, нарощуванні можливостей системи.

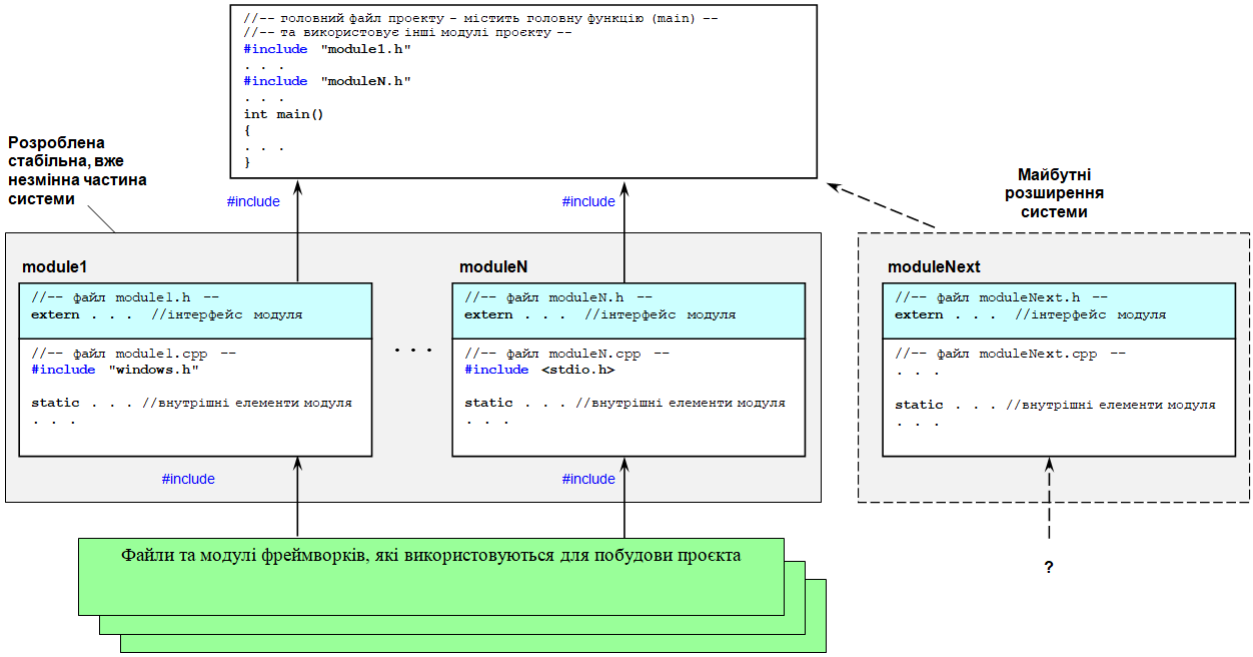


Рис. 2.10. Модульність проєкту C++

Можна зформулювати ознаки розумної стратегії модульності наступним чином:

- розділення програмного коду на **невеличкі** модулі, які **мінімально зв'язані між собою** (в ідеалі – зовсім незалежні один від одного)

- залежності між модулями повинні бути такими, щоб при розвитку системи, наприклад, при нарощуванні функціональних можливостей, нові можливості містилися б у нових модулях, і **при додавання нових модулів старі модулі й їхні залежності не повинні змінюватися**

- забезпечення можливостей **використання розроблених модулів в інших проєктах**

Контрольні запитання

1. Що таке препроцесор C++?
2. Назвіть директиви препроцесора
3. Що робить директива #define?
4. Як запрограмувати макрос з параметрами?
5. Що робить директива #include?
6. Що відображає схема #include-ієрархії?
7. Яка мета модульності програм?
8. Що містить вихідний текст модуля програми на C++?
9. Навіщо потрібен файл заголовку?
10. Чи може модуль складатися тільки з файлу заголовку?
11. Які файли модулів необхідно включати у список файлів проєкту у середовищі Visual Studio C++?
12. Що означає слово "моноліт" стосовно програмного коду?
13. Якою може бути стратегія модульності проєктів C++?
14. Яким чином можна розділювати програмний код на модулі?

Лекція 3

Тема 2.3. Класи C++. Інкапсуляція. Спадкування. Поліморфізм

Поняття класу

Клас - це опис властивостей об’єктів певного типу. Клас може об’єднувати:

- дані-члени
- функції-члени
- конструктори
- деструктор.

Оголошення класу

Для оголошення використовується ключове слово class

```
class Ім’я
{
private:
. . .           //список членів
protected:
. . .           //список членів
public:
. . .           //список членів
};
```

Оголошення вводить новий тип із вказаним ім’ям. Цей тип створюється на основі вже відомих типів даних – елементи класу можуть мати стандартні типи, або типи, які створені програмістом до цього. Крім того, в оголошенні визначається обмеженість доступу до членів класу:

- **private** означає, що члени класу доступні тільки усередині цього класу.
- **protected**: члени класу доступні у цьому класі та у похідних класах.
- **public**: члени класу доступні будь-де в області видимості даного класу. Такі члени зазвичай використовуються як інтерфейс до об’єктів.

За замовчуванням в C++ члени класів є private.

Дані-члени

Це набір елементів, можливо різних типів. Це можуть бути, наприклад, перемінні.

Функції-члени

Це функції, що маніпулюють даними-членами класу.

- Мають доступ до усіх даних-членів або функцій-членів, оголошеним усередині класу.
- Можуть бути визначені усередині або поза оголошенням класу
- Можуть бути віртуальними.
- Функції-члени часто звуть **методами**.

Конструктор

- Конструктор автоматично викликається при створенні об’єкту типу клас. Якщо створюється динамічний (тимчасовий) об’єкт за допомогою оператора n**ew**, то конструктор викликається при виконанні цього оператора
- Конструктор має те саме ім'я, що і клас, для якого він визначений.
- Конструктор без параметрів – конструктор за замовчуванням (*default constructor*).
- Конструктор може мати параметри (аргументи). Конструктор не має типу – його не можна оголошувати навіть як **void**
- Конструктор може бути **private**, **protected** або **public**
- Може бути визначений усередині або поза оголошенням класу (аналогічно функціям-членам)
- Конструкторів може бути декілька. Якщо в оголошенні класу не записано жодного конструктора, то автоматично створюється порожній конструктор за замовчуванням.
- Зазвичай конструктор використовується для початкової ініціалізації даних при створенні нового об'єкта.

Деструктор

- Деструктор автоматично викликається при знищенні (або виході з області видимості) цього об’єкта. Якщо об’єкт був створений оператором **new**, то деструктор викликається при виконанні операції **delete** при знищенні цього динамічного об’єкта
- Має те ж ім'я, що і клас, але зі знаком ~ (тильда) попереду
- Може бути віртуальним
- Деструктор може бути тільки один

- Деструктор зазвичай використовується для дій, що супроводжують знищення об'єкта.

Приклад оголошення класу

Поставимо за мету запрограмувати сховище даних – стек, вимоги до якого такі:

- для запису елемента потрібно викликати функцію **Push**(значення)
- для читання передбачити функцію **Get()**, причому дані зчитуються у зворотному порядку – першим зчитується останній записаний елемент
- користувачі стека байдужі до подробиць його реалізації

```
class IntStack
{
private:
   int *p;              //дані-члени
   int size, num;
public:
   IntStack (int);      //конструктор
  ~IntStack(void);      //деструктор

   void Push(int);      //функції-члени інтерфейсу стеку
   int Get(void);
};
```

Далі потрібно написати визначення для усіх функцій-членів, конструктора та деструктора

Визначення функцій-членів, конструкторів і деструктора класу

Для функцій-членів, конструкторів і деструктора їхнє визначення аналогічно визначенню функцій. При визначенні поза оголошенням використовується операція (`::`).

Визначення членів розглянемо на прикладі класу **`IntStack`**

```
IntStack::IntStack(int s)
{
p = new int[s];           //створюється робочий масив стека p[]
size = s;
num = 0;
}

IntStack::~IntStack(void)     //деструктор знищує масив p[]
{
delete []p;
}
```

```
void IntStack::Push(int elem)  //функція запису елемента у стек
{
if (num < size)            //контроль переповнювання стека
   {
   p[num] = elem;
   num++;    //лічильник кількості збережених елементів у стеку
   }
}

int IntStack::Get(void)   //ця функція витягає елемент зі стека
{
if (num < 1) return 0;
num--;
return p[num];
}
```

Створення екземплярів об’єктів

Після того, як оголошенням класу та визначенням його функцій-членів описано новий тип, цей тип можна використати для створення перемінних. Кожна з таких перемінних є об’єктом, який розташовується у пам'яті, Як і звичайні перемінні, перемінні типу клас можуть бути глобальними, локальними та динамічними.

Глобальні об’єкти

Глобальні об’єкти створюються шляхом оголошення поза будь-якою функцією. Спочатку вказується тип а потім ім’я об’єкта, наприклад

```
IntStack Q(1000);     //створюється об’єкт-стек з ім’ям Q
```

Таке оголошення не просто вводить нове ім’я Q. Таке оголошення є інструкцією дії – автоматично викликається конструктор. А у конструктора є аргумент, у який записується 1000.

Після того, як об’єкт створений, можна викликати його **public** функції-члени, При виклику спочатку ім’я об’єкту, потім крапка і далі ім’я члену, наприклад

```
Q.Push(99);
Q.Push(7);

val = Q.Get();
val = Q.Get();
```

Глобальний об’єкт існує весь час роботи програми.

Локальні об’єкти

Локальні об’єкти створюються усередині функцій, наприклад

```
void MyFunc(void)
{
int val;
IntStack Q(1000);      //створюється локальний об’єкт з ім’ям Q

Q.Push(99);
Q.Push(7);

val = Q.Get();
val = Q.Get();
}
```

Локальний об’єкт знищується при виході з зони його видимості – тіла функції MyFunc

Зоною існування-видимості може бути і блок { }

```
void MyFunc(int m)
{
int val;
if (m > 0)
   {
   IntStack Q(1000);     //створюється локальний об’єкт з ім’ям Q
   Q.Push(99);
   Q.Push(7);
   . . .
   val = Q.Get();
   val = Q.Get();
   }
val = Q.Get();     //помилка компіляції - об'єкт Q тут не видимий
}
```

Динамічні об’єкти

Можливо створювати та знищувати об’єкти тоді, коли у цьому виникає потреба. Спочатку необхідно оголосити перемінну-вказівник:

```
IntStack *p;    //оголошуємо вказівник на об’єкт типу IntStack
```

Таке оголошення об’єкт не створює. Створюється тільки перемінна-вказівник, яка буде колись зберігати адресу майбутнього об’єкта. Сам об’єкт створюється за допомогою операції **new**.

```
p = new IntStack(1000);    //створення динамічного об’єкта
if (p)    //якщо об’єкт успішно створений – з ним можна працювати
   {
   p->Push(99);            //виклик функції-члена Push
   p->Push(7);

   val = p->Get();
   val = p->Get();

   delete p;               //знищення динамічного об’єкта
   }
```

Існує вірогідність, що “new” відмовиться створювати об'єкт – тоді результатом буде нульова адреса і значення вказівника p = 0. Відповідно, не можна отримати доступ до членів неїснуючого об'єкта. Тому варто завжди перевіряти значення отриманого вказівника на об'єкт.

Успадкування класів

Від класів C++ можна створювати класи-спадкоємці (інша назва – похідні класи). Можна сказати, що об’єкт A, властивості якого описуються класом ClassA, може мати спадкоємця – об’єкт B, який має властивості об’єкта A, плюс деякі інші, індивідуальні властивості, які описуються класом-спадкоємцем ClassB

```
class ClassA
{
. . .    //члени класу – властивості об’єкта A
};

//--оголошення класа-спадкоємця (похідного класа)--
class ClassB : public ClassA
{
. . .    //члени цього класу – індивідуальні властивості об’єкта
B
};
```

У класі ClassB будуть міститися усі члени класу ClassA плюс індивідуальні члени, оголошені у класі ClassB.

Клас ClassA зветься **базовим** класом. Якщо кожний клас-спадкоємець має безпосередньо тільки один базовий клас, то будемо називати таке успадкування простим, одиничним. Навіть таким простим успадкуванням може бути побудована велика ієрархія з багатьох класів-спадкоємців у вигляді дерева

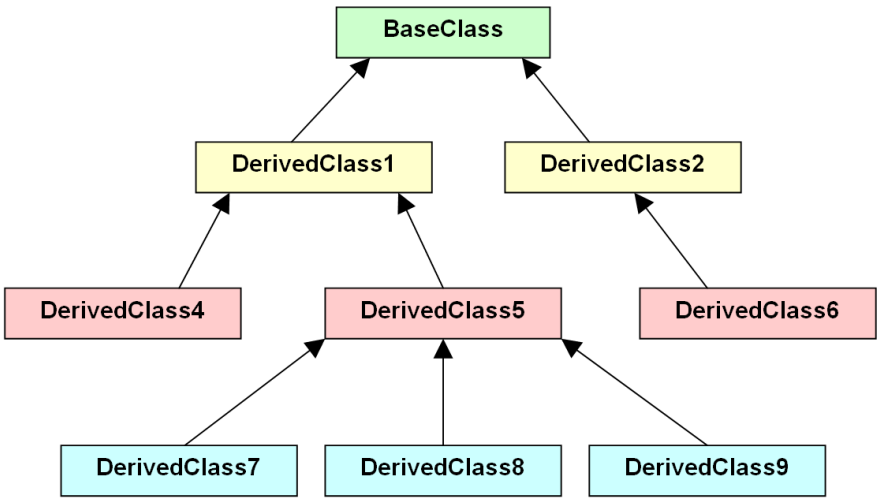


Рис. 3.1. Приклад дерева успадкування класів

Для відображення взаємин класів часто використовують так звані ***діаграми класів***. Згодом ми їх розглянемо докладніше, а поки що скористаємося правилом – стрілка вказуватиме на базовий клас.

Приклад

```
class SpacePoint          //оголошення базового класу
{
private:
   double x, y, z;
public:
   void Move(double, double, double);
   double DistanceFromCenter(void);
};

//---оголошення похідних класів----
class SpaceTimePoint : public SpacePoint
{
private:
   long time;
public:
   void SetTime(long);
};

class SpaceCircle : public SpacePoint
{
private:
   double radius;
public:
   void SetRadius(double);
};
```

```
//--визначення функцій-членів класів--
void SpacePoint::Move(double xx,double yy,double zz)
{
  x = xx;
  y = yy;
  z = zz;
}

double SpacePoint::DistanceFromCenter(void)
{
  return sqrt(x*x + y*y + z*z);
}

void SpaceTimePoint:: SetTime(long t)
{
  time = t;
}

void SpaceCircle::SetRadius(double r)
{
  radius = r;
}
```

```
//--робота з об’єктами--
SpaceTimePoint QT;        //створення об’єктів
SpaceCircle QC;

QT.Move(3.7, 55.1, 9);
QT.SetTime(2022);
. . .
QC.Move(0, 2.1, -7.5);
QC.SetRadius(1.6);
. . .
distance = QC.DistanceFromCenter();
. . .
distance = QT.DistanceFromCenter();
```

Як бачимо, у об’єктів похідних класів ми викликаємо не тільки їхні індивідуальні методи – **SetTime** та **SetRadius**, а й методи базового класу – **Move** та **DistanceFromCenter**.

Щодо конструкторів та деструкторів при успадкуванні класів. При створенні об'єкта похідного класу автоматично спочатку викликається конструктор базового класу, а потом конструктор похідного класу. При знищенні об'єкта автоматично спочатку викликається деструктор похідного класу а потім деструктор базового класу. У розглянутому вище прикладі немає явно оголошених контструкторів та деструкторів.

Класи та модульність

Якщо потрібно зробити модульний проект, то виникає запитання: як розподілити вихідний текст щодо класів по модулям і файлам? В C++ жорстких обмежень немає: в одному файлі модуля може міститися програмний код декількох класів, або, навпаки, опис одного класа може бути розподілений по декільком файлам.

Для ілюстрації візьмемо розглянутий вище приклад. Запишемо оголошення базового класу **SpacePoint** у файлі заголовку **SpacePoint.h,** а визначення функцій-членів цього класу – у файлі **SpacePoint.cpp.** Отримаємо модуль **SpacePpoint** з двох файлів. Так само опис похідних класів будемо записувати у окремих модулях, кожний з цих модулів представлятиметься заголовком *.h і файлом *.cpp. Код використання цих класів – створення екземплярів об'єктів та виклику їхніх методів буде міститися у головному файлі. Відобразимо це наступним чином

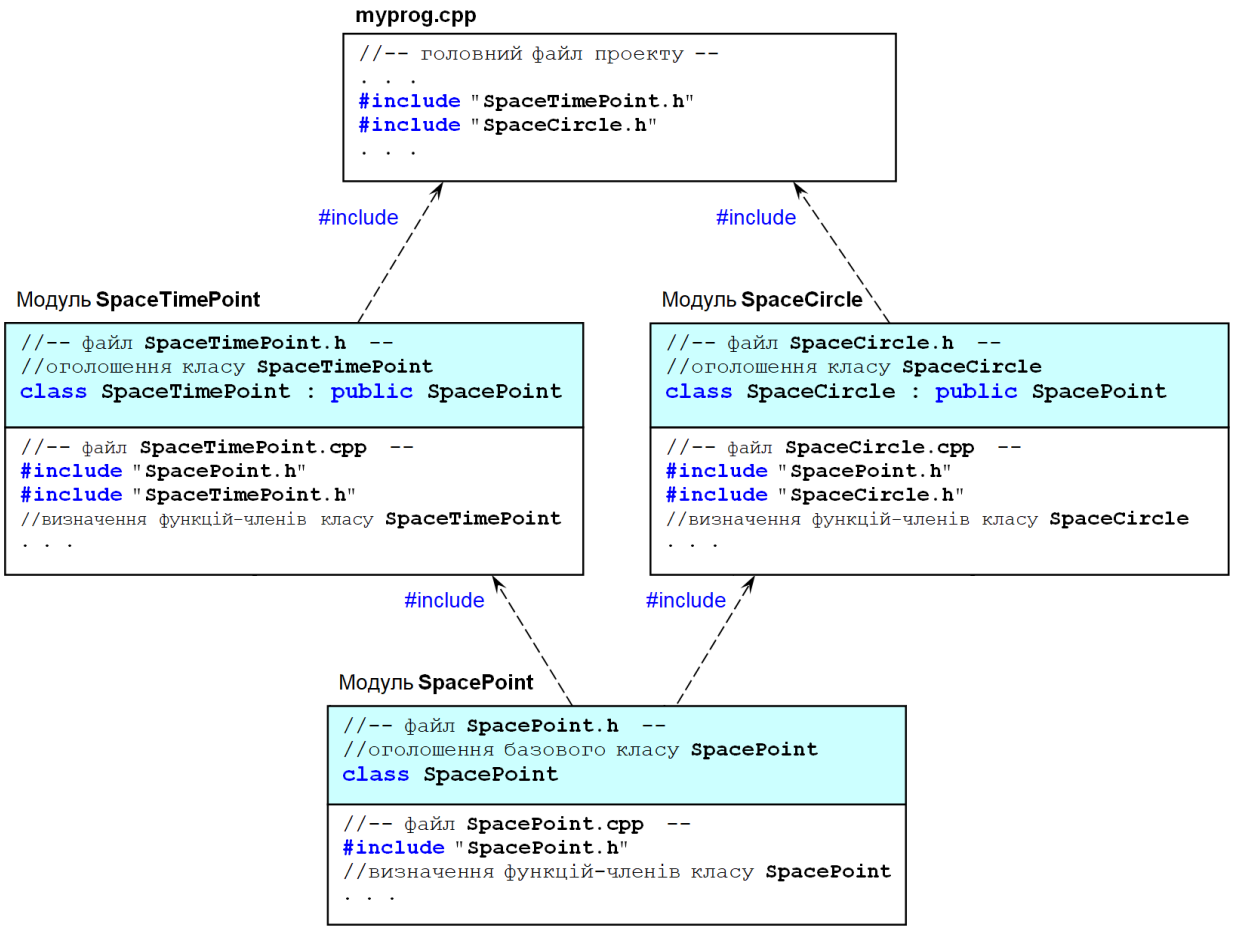


Рис. 3.2. Класи, модулі та файли

На такій схемі залежностей (#include-ієрархії) модуль базового класу опинився на нижньому рівні на відміну від діаграм класів, де зазвичай базовий клас розташовується зверху.

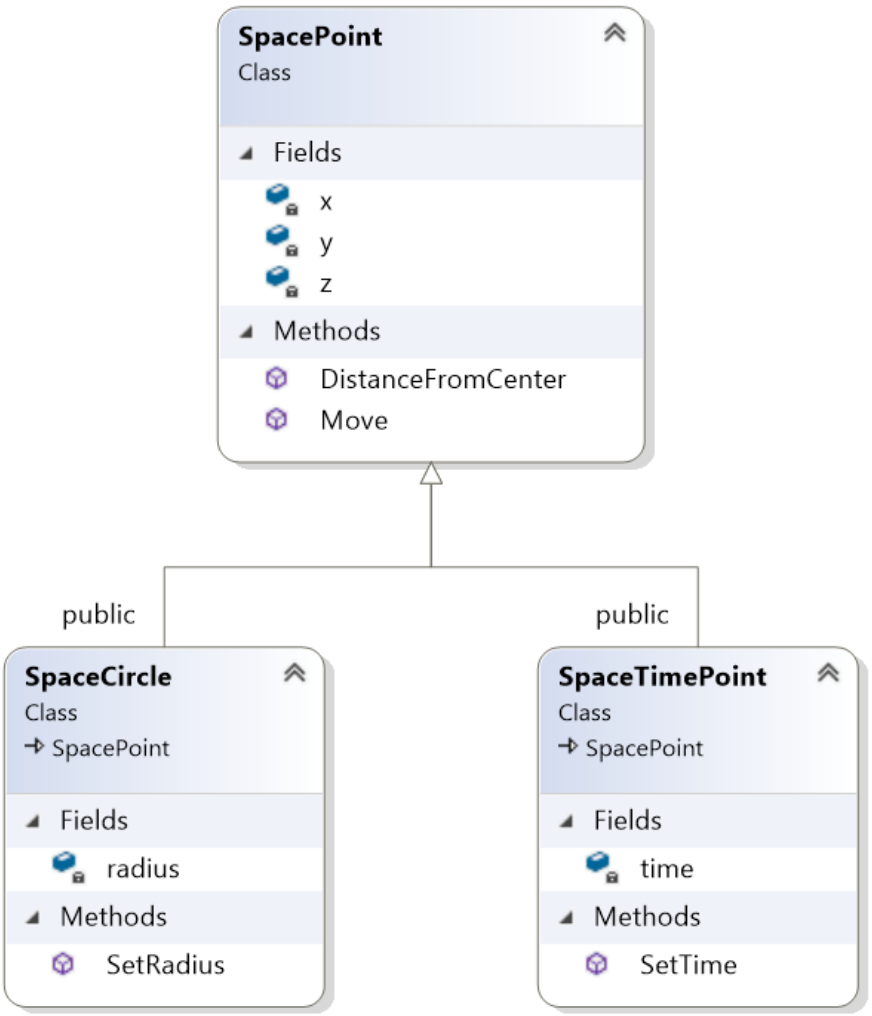


Рис. 3.3. Діаграма класів (зроблена у середовищі Visual Sudio C++)

Таким чином, можна зробити висновок, що клас може надавати деякі можливості, як мінімум у двох аспектах:

1. Клас – це засіб створення нового типу даних, відповідно опису та реалізації деякої абстракції для вирішення завдань створення певної програмної системи на основі об'єктно-орієнтованого підходу.

2. Клас – це “модулеутворюючийї” архітектурний елемент побудови програмної системи. Правильно спроектовані класи дозволяють отримати зручну модульність.

Іншими словами, можна казати про дві ролі класів у дизайні програмних систем.

Контрольні запитання

1. Що таке клас? Що містить клас C++?
2. Як можна описувати властивості об'єктів класами?
3. Що вказується в оголошенні класу?
4. Чим відрізняються protected члени від public?
5. Коли та як викликаються конструктор та деструктор?
6. Що таке конструктор за замовчуванням?
7. Скільки може бути деструкторів?
8. Чи може визначення функцій-членів бути записано в оголошенні класу?
9. Як створюється динамічний об'єкт?
10. Що таке локальний об'єкт?
11. Коли спрацьовує деструктор локального об'єкта?
12. Що таке клас-спадкоємець?
13. Яка послідовність роботи конструкторів та деструкторів базового та похідного класів?
14. Що відображається на діаграмах класів?
15. Що таке ієрархія успадкування?
16. Стосовно класів та модульності: що означає дві ролі класів в побудові систем?

Лекція 4

Тема 2.3. (Продовження)

Віртуальні функції

Перед тим, як дати визначення предмету обговорення, розглянемо наступний приклад. Нехай буде базовий класс (Base) та похідні (Derived, Derived2). Нижче код оголошення-визначення для цих класів

```
#include <iostream>
using namespace std;

class Base
{
protected:
    virtual void fun(void) {
        cout << "fun of Base class" << endl;
    }
public:
    void method(void) {
        cout << "method calls a ";
        fun();
     }
};

class Derived : public Base
{
protected:
    void fun(void) {
        cout << "fun of Derived class" << endl;
    }
};

class Derived2 : public Base
{
protected:
    void fun(void) {
        cout << "fun of Derived2 class" << endl;
    }
};
```

Як бачимо, у похідних класах перевизначається функція **`fun()`**.

Функцію **`fun()`** викликає функція **`method()`**, яка оголошена та визначена у базовому класі. Ми хочемо, щоб у похідних класах функція **`method()`** працювала би з відповідно перевизначеною версією **`fun()`**.

Зробимо тестову програму

```
//--test--
int main()
{
    Base B;
    B.method();

    Derived D;
    D.method();

    Derived2 D2;
    D2.method();

    return _getwch();       //щоб вікно консолі автоматично не зникло
}
```

Результат виконання тестової програми наданий нижче

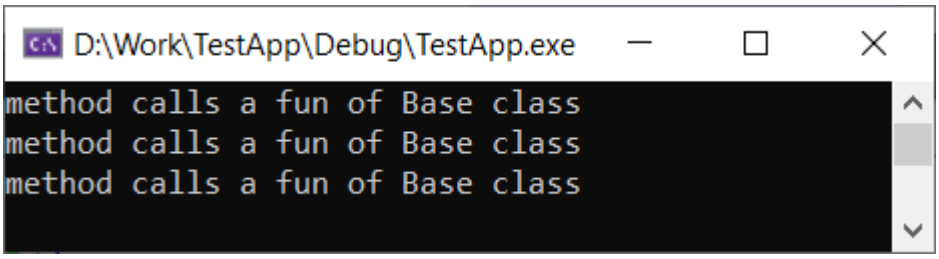


Рис. 3.3. Вікно консольної тестової програми

Чому програма демонструє не те, що від неї очікувалося? Замість перевизначеної **fun()** у кожному похідному класі викликається версія цієї функції базового класу. Як вказати, щоб об'єкт похідного класу у тілі функції **method()** викликав би функцію **fun()** саме від його класу, а не базового?

Рішення цієї проблеми виявляється дуже простим. Достатньо в базовому класі оголосити функцію **fun() віртуальною**:

```
class Base
{
protected:
    virtual void fun(void) {                    //віртуальна функція
        cout << "fun of Base class" << endl;
    }
public:
    void method(void) {
        cout << "method calls a ";
        fun();
     }
};
```

Решта коду (похідних класів та тестової програми) залишається без будь-яких змін. Тестова програма у вікні вже демонструє те, що треба:

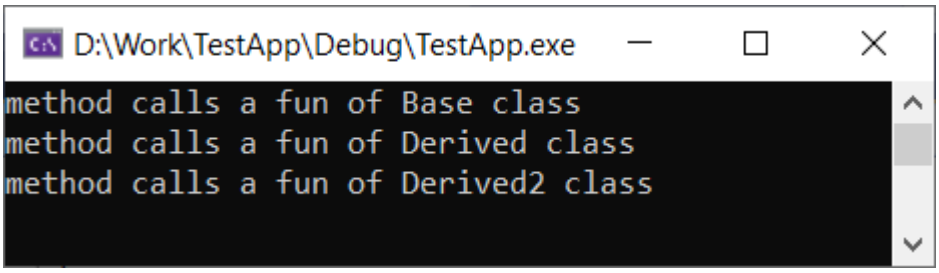


Рис. 3.4. Тест виконаний успішно

Таким чином, віртуальна функція – це така функція-член класу, яка коректно викликається у об'єктів тих класів, для яких вона перевизначена.

Може виникнути питання: а кому тоді потрібні некоректно працюючі функції? Чому б в реалізації мови C++ усі функції не зробити як віртуальні, щоб взагалі не виникало би непорозумінь? В деяких інших об'єктно-орієнтованих мовах програмування так і зроблено – наприклад, в Java усі методи класів працюють як віртуальні функції C++ [10]. Відповідь може бути такою. Для віртуальних функцій компілятор C++ створює спеціальну таблицю, яка вбудовується в машинний код і аналізується в ході виконання програми для забезпечення виклику саме тої версії функції, яка належіть класу поточного створенного об'єкта. Кожна така таблиця потребує ресурсів, а крім цього, виклик віртуальних функцій відбувається дешо повільніше порівняно зі звичайними функціями.

Чисто віртуальні функції і абстрактні класи

Такий різновид віртуальних функцій є засобом підвищення рівня абстракції. Змінимо оголошення базового класу попереднього прикладу наступним чином – зробимо функцію **`fun()`** чисто віртуальною

```
class Base
{
protected:
    virtual void fun(void) = 0;        //чисто віртуальна функція
public:
    void method(void) {
        cout << "method calls a ";
        fun();
     }
};
```

Треба відзначити, що таке позначення, прийняте в мові C++ для чисто віртуальних функцій не є інтуїтивно зрозумілим (як свідчить досвід спілкування, у деяких початківців може скластися враження, що така функція немов би дорівнює нулю, що вочевидь, некоректно).

Основні положення щодо чисто віртуальних функцій

- якщо чисто віртуальна функція функція оголошена в певному класі, то в цьому класі вона не може бути визначена. В базовому класі така функція може бути тільки у вигляді оголошення
- якщо в класі є хоча б одна чисто віртуальна функція, тоді цей клас зветься *абстрактним*
- не може бути створений об'єкт абстрактного класу (хоча вказівник на абстрактний клас може бути оголошений, але ініціалізований такий вказівник може бути тільки адресою об’єкта похідного неабстрактного класу)
- визначення чісто віртуальних функцій може бути тільки у похідному класі (хочк це не означає, що якщо клас похідний, то у ньому обов’язково повинні бути визначення усіх чисто віртуальних функцій базового класу). Замість слова «визначення» для чисто віртуальних функцій в ООП часто вживається слово «реалізація»
- чисто віртуальні функції дають можливість у програмному коді маніпулювати абстракціями з відкладеною реалізацію.

Продовжимо приклад. Оголосимо похідні класи від розглянутого вище абстрактного класу Base. У похідних класах буде визначена (реалізована) функція **`fun()`**

```
class Derived : public Base
{
protected:
    void fun(void) {
        cout << "fun of Derived class" << endl;
    }
};

class Derived2 : public Base
{
protected:
    void fun(void) {
        cout << "fun of Derived2 class" << endl;
    }
};
```

Примітка. Оголошеня цих класів повністю повторює код попереднього прикладу.

Далі зробимо тестову программу, яка буде створювати екземпляри об'єктів тільки похідних класів (об'єкт класу Base створений бути не може, оскільки це вже абстрактний класс).

```
//--test--
int main()
{
    Derived D;
    D.method();

    Derived2 D2;
    D2.method();

    return _getwch();
}
```

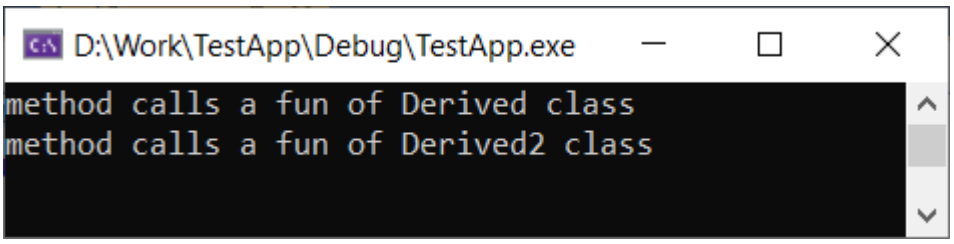


Рис. 3.5. Реультат тесту

Якщо в програмній системі деяка ієрархія класів розпочинається з базового класу, у якого немає чисто віртуальних функцій і, відповідно, усі функції визначені, то це звуть визначенням (реалізацією) за замовчуванням. Якщо базовий клас є абстрактним – тоді кажуть про відкладену реалізацію.

Поліморфізм

Розглянемо приклад програми. У вихідному тексті буде базовий абстрактний класс Base та похідні класи – Derived, Derived2.

```
class Base
{
public:
    virtual void method(void) = 0;
};

class Derived : public Base
{
public:
    void method(void) { cout << "method of Derived class" << endl; }
};

class Derived2 : public Base
{
public:
    void method(void) { cout << "method of Derived2 class" << endl; }
};
```

Далі визначимо функцію, яка буде використовувати об’єкти класів.

```
void clientFunc(Base* p) {
    cout << "calling a ";
    p->method();
}
```

У цій функції аргумент є вказівником на базовий клас. Але, оскільки клас Base є абстрактним, то об'єкт не може бути створений, тобто, можливо викликати функцію **method()** тільки похідних класів.

У тестовій програмі створюються об'єкти і їхні адреси передаються у функцію-клієнт цих класів.

```
//--test--
int main()
{
    Base* p = new Derived;
    clientFunc(p);
    delete p;

    p = new Derived2;
    clientFunc(p);
    delete p;

    return _getwch();
}
```

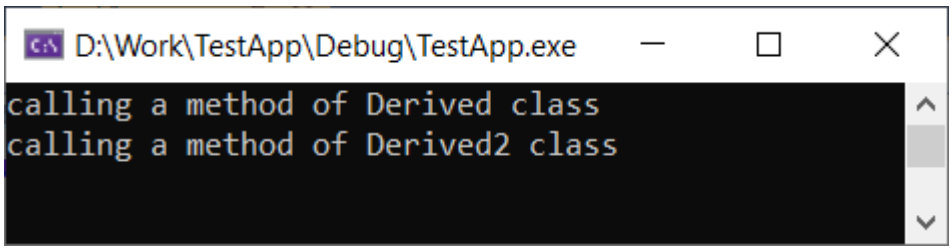


Рис. 3.6. Результат тесту

Незважаючи на те, що оголошується вказівник на базовий клас, по цьому вказівнику робиться виклик метода того класу, чий об'єкт був створений в ході виконання програми. Можна сказати, що це поліморфний виклик метода – той самий програмний код, але викликає однойменний метод будь-якого класу з ієрархії спадкування від Base. Вибір метода потрібного класу робиться на етапі виконання програми. Селектором є значення вказівника.

```
Base* p = new Derived;
p->method();          //поліморфний виклик
```

Примітки:

- для поліморфного виклику функція обов'язково має бути віртуальною. Але не вимагається, щоб вона була чисто віртуальною
- зовсім не обов'язково, щоб базовий клас був абстрактним. Головне – щоб у ньому була оголошена віртуальна функція, ім'я якої треба вказати у поліморфному виклику.

Щойно розглянутий варіант поліморфізму на основі вказівника на базовий клас не вважається єдино можливим варіантом організації поліморфізму. У мові C++ є також інші засоби впровадження ідеї поліморфізму.

В загальному сенсі, основна ідея поліморфізму – "*один інтерфейс, багато методів*".

Віртуальний деструктор

Виконаємо невеличке дослідження. Поставимо за мету з'ясувати, як поводять себе конструктори та деструктори при організації поліморфізму через вказівник на базовий клас. Розглянемо це на прикладі

```
#include <iostream>
using namespace std;

class Base
{
public:
    Base(void) {
        cout << "constructor of Base class" << endl;
    }
   ~Base(void) {
        cout << "destructor of Base class" << endl << endl;
    }
    virtual void method(void) = 0;
};

class Derived : public Base
{
public:
    Derived(void) {
        cout << "constructor of Derived class" << endl;
    }
   ~Derived(void) {
        cout << "destructor of Derived class" << endl;
    }
    void method(void) {
        cout << "method of Derived class" << endl;
    }
};
```

```
class Derived2 : public Base
{
public:
    Derived2(void) {
        cout << "constructor of Derived2 class" << endl;
    }
   ~Derived2(void) {
        cout << "destructor of Derived class" << endl;
    }
    void method(void) {
        cout << "method of Derived2 class" << endl;
    }
};
```

В оголошенні цих класів явно записані конструктори та деструктори. Далі визначимо функцію, яка буде робити поліморфний виклик метода по вказівнику на базовий клас і потім знищувати динамічний екземпляр об’єкта.

```
void clientFunc(Base* p) {
    cout << "calling a ";
    p->method();
    delete p;          //об'єкт більше не потрібен - знищуємо його
}
```

Завдяки знищенню об'єктів у **clientFunc** вихідний текст тестової програми стає лаконічнішим

```
//--test--
int main()
{
    clientFunc(new Derived);

    clientFunc(new Derived2);

    return _getwch();
}
```

Результат роботи – у вікні консолі

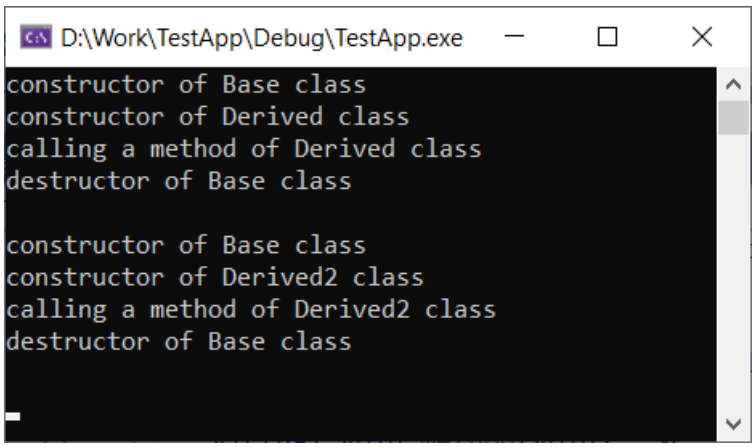


Рис. 3.7. Виконання тестової програми

При роботі такой тестової програми з конструкторами усе гаразд – спочатку, як і повинно бути, викликається конструктор базового класу а потім конструктор похідного класу. А з деструкторами щось не так – взагалі не викликаються деструктори похідних клаів. Для даної прграми це не дуже критично – у деструкторі лише має виводитися рядок тексту, але уявимо собі, що у деструкторах мала б звільнятися пам'ять від якихось робочих масивів – тоді будуть накопичуватися витоки пам’яті або, взагалі, програма може аварійно припинити роботу.

Вирішення такої проблеми є досить простим – достатньо оголосити деструктор базового класу віртуальним

```
class Base
{
public:
    Base(void) {
        cout << "constructor of Base class" << endl;
    }
    virtual ~Base(void) {       //віртуальний деструктор
        cout << "destructor of Base class" << endl << endl;
    }
    virtual void method(void) = 0;
};
```

Решта вихідного тексту програми – без будь-яких змін.

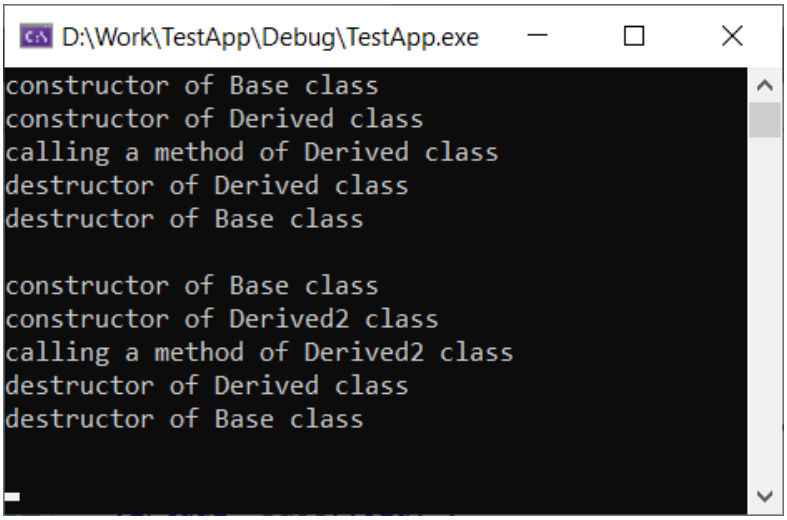


Рис. 3.8. Робота виправленої програми

Тепер деструктори похідних класів викликаються.

Висновок. Для забезпечення коректної роботи поліморфізму через вказівник на базовий клас, потрібно деструктор базового класу оголошувати віртуальним.

Контрольні запитання

1. Який сенс робити деякі функції-члени класів віртуальними?
2. Чому в C++ функції-члени класів не є віртуальними за замовчуванням (автоматично)?
3. Що таке чисто віртуальна функція і як вона позначається в C++?
4. Що таке абстрактний клас?
5. Що таке поліморфізм та що таке поліморфний виклик метода?
6. Що означає “один інтерфейс – багато методів”?
7. Як можна реалізувати поліморфізм?
8. Навіщо потрібен віртуальний деструктор?

Тема 2.4. Класи C++. Множинне спадкування

Поняття множинного спадкування

Множинне спадкування – це утворення похідного класу одразу від декількох базових класів, наприклад

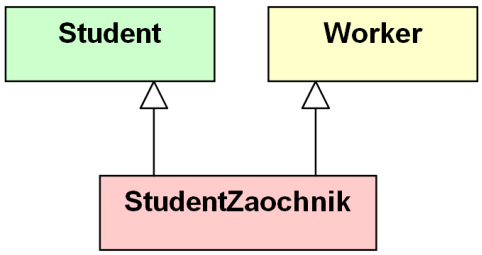


Рис. 4.1. Ілюстрація множинного спадкування

Оголошення класу-спадкоємця двох базових класів має наступний вигляд

```
class StudentZaochnik : public Student, public Worker
{
. . .
};
```

У похідному класі об’єднуються властивості та функції об’єктів усіх базових класів. Якщо у базових класах немає членів з однаковими іменами, то це є найпростішим випадком для множинного спадкування. Розглянемо приклад

```
//--оголошення класів--
#include <iostream>
#include <string>
using namespace std;

class Student
{
private:
    string studyplace;
public:
    Student(void) {
        cout << "Student ";
    }
    void setStudy(const char* s) {
        studyplace.assign(s);
    }
    void askStudyPlace(void) {
        cout << studyplace << " ";
    }
};
```

```
class Worker
{
private:
    string workplace;
public:
    Worker(void) {
        cout << "Worker ";
    }
    void setWork(const char* s) {
        workplace.assign(s);
    }
    void askWorkPlace(void) {
        cout << workplace << " ";
    }
};
```

```
class StudentZaochnik : public Student, public Worker
{

};
```

Тестова програма

```
//--test--
int main()
{
    Student S;
    S.setStudy("NAU");
    S.askStudyPlace();
    cout << endl;

    Worker W;
    W.setWork("Bank");
    W.askWorkPlace();
    cout << endl;

    StudentZaochnik SZ;
    SZ.setStudy("KPI");
    SZ.setWork("Google");
    SZ.askStudyPlace();
    SZ.askWorkPlace();

    return _getwch();
}
```

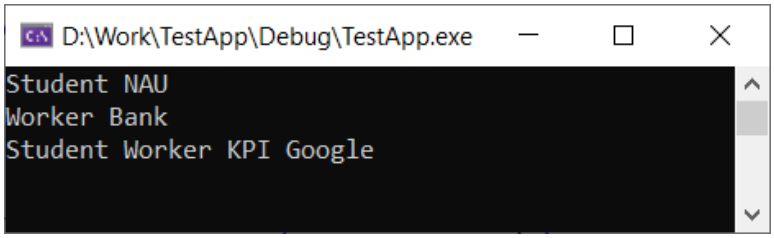


Рис. 4.2. Результат виконання тестової програми

Зверніть увагу, що для об'єкта класу **StudentZaochnik** спрацювали два конструктора базових класів у послідовності відповідно порядку запису імен цих класів в оголошенні похідного класу.

Особливості множинного спадкування

Наявність членів з однаковими іменами у базових класах породжує деякі незручності для множинного спадкування. Розглянемо це на прикладі.

```
#include <iostream>
#include <string>
using namespace std;

class Student
{
private:
    string studyplace;
public:
    void setStudy(const char* s) {
        studyplace.assign(s);
    }
    void askAbout(void) {
        cout << "I'm a student of " << studyplace << endl;
    }
};
```

```
class Worker
{
private:
    string workplace;
public:
    void setWork(const char* s) {
        workplace.assign(s);
    }
    void askAbout(void) {
        cout << "I'm a worker of " << workplace << endl;
    }
};
```

```
class StudentZaochnik : public Student, public Worker
{

};
```

```
//--test--
int main()
{
    Student S;
    S.setStudy("NAU");
    S.askAbout();
    cout << endl;

    Worker W;
    W.setWork("Bank");
```

```
    W.askAbout();
    cout << endl;

    StudentZaochnik SZ;
    SZ.setStudy("KPI");
    SZ.setWork("Google");
    SZ.Student::askAbout();      //буде: I'm a student of KPI
    SZ.Worker::askAbout();       //буде: I'm a worker of Google

    return _getwch();
}
```

У обох базових класах – Student та Worker є функція-член з іменем **askAbout**. Оскільки у класі StudentZaochnik вона не перевизначена, то для того, щоб її викликати у об'єкта цього класу, необхідно явно вказувати, від якого базового класу її взяти. Якщо це незручно для клієнтів цього класу, то можна перевизначити функцію **askAbout**, наприклад, наступним чином

```
class StudentZaochnik : public Student, public Worker
{
public:
    void askAbout(void) {
        Worker::askAbout();
        cout << "and" << endl;
        Student::askAbout();
    }
};
```

```
//--test--
int main()
{
    Student S;
    S.setStudy("NAU");
    S.askAbout();
    cout << endl;

    Worker W;
    W.setWork("Bank");
    W.askAbout();
    cout << endl;

    StudentZaochnik SZ;
    SZ.setStudy("KPI");
    SZ.setWork("Google");
    SZ.askAbout();

    return _getwch();
}
```

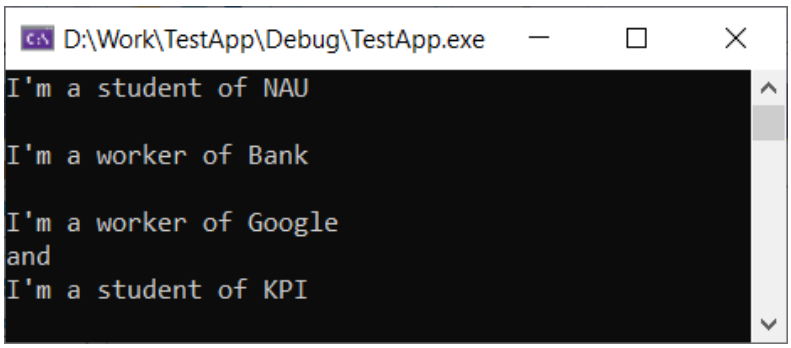


Рис. 4.3. Результат виконання тесту

Поки що можна констатувати, що особливих проблем з множинним спадкування ми не побачили.

Ромбічне успадкування

Розглянемо наступний приклад. Уявимо, що є базовий класс **Person**. Похідними від нього будуть класи **Student** та **Worker**. Клас **StudentZaochnik** буде похідним від **Student** та **Worker**. Відобразимо це діаграмою класів

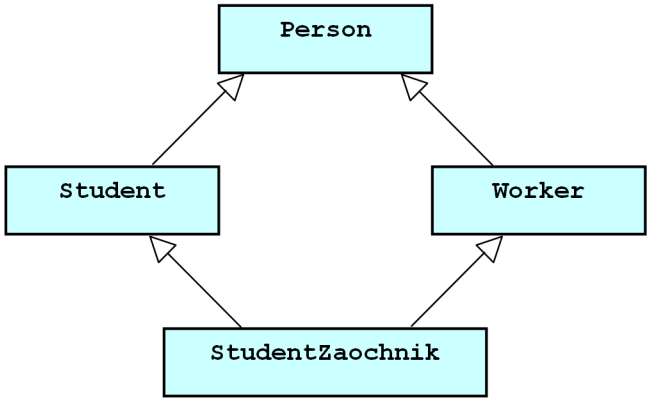


Рис. 4.4. Приклад ромбічного успадкування

Оголошення класів

```
#include <iostream>
#include <string>
using namespace std;

class Person
{
private:
    string name;
public:
    void setName(const char* s) {
        name.assign(s);
    }
```

```
    void getName(void) {
        cout << "My name is " << name << endl;
    }
};

class Student : public Person
{
private:
    string studyplace;
public:
    void setStudy(const char* s) {
        studyplace.assign(s);
    }
    void askAbout(void) {
        cout << "I'm a student of " << studyplace << endl;
    }
};

class Worker : public Person
{
private:
    string workplace;
public:
    void setWork(const char* s) {
        workplace.assign(s);
    }
    void askAbout(void) {
        cout << "I'm a worker of " << workplace << endl;
    }
};

class StudentZaochnik : public Student, public Worker
{
public:
    void askAbout(void) {
        Worker::askAbout();
        cout << "and" << endl;
        Student::askAbout();
    }
};
```

Далі у тесті має створюватися екземпляр об'єкта класу **StudentZaochnik**

```
//--test--
int main()
{
    StudentZaochnik SZ;
    SZ.setName("Oleg");
    SZ.setStudy("KPI");
    SZ.setWork("Google");

    SZ.getName();
    SZ.askAbout();
    return _getwch();
}
```

Але до виконання такої програми не дійде – компілятор видає помилки “**ambiguous access of 'setName'**”, та “**ambiguous access of 'getName'**”.

Класи **Student** та **Worker** є спадкоємцями класу **Person**, і кожен з них успадковує члени **setName** та **getName**. Тоді компілятору стає незрозуміло, як включати ці члени з базових класів у похідний клас множинного успадкування **StudentZaochnik**.

Віртуальні базові класи

Вірогідно, саме через проблему реалізації ромбічного успадкування в деяких інших мовах ООП, наприклад, Java, взагалі відмовилися від множинного успадкування класів [10]. А в C++ ця проблема вирішується зовні, як здається, просто – достатньо дописати слово **virtual** в оголошенні класів **Student** та **Worker.** Решта коду не змінюється.

```
class Student : public virtual Person
{
private:
    string studyplace;
public:
    void setStudy(const char* s) {
        studyplace.assign(s);
    }
    void askAbout(void) {
        cout << "I'm a student of " << studyplace << endl;
    }
};
```

```
class Worker : public virtual Person
{
private:
    string workplace;
public:
    void setWork(const char* s) {
        workplace.assign(s);
    }
    void askAbout(void) {
        cout << "I'm a worker of " << workplace << endl;
    }
};
```

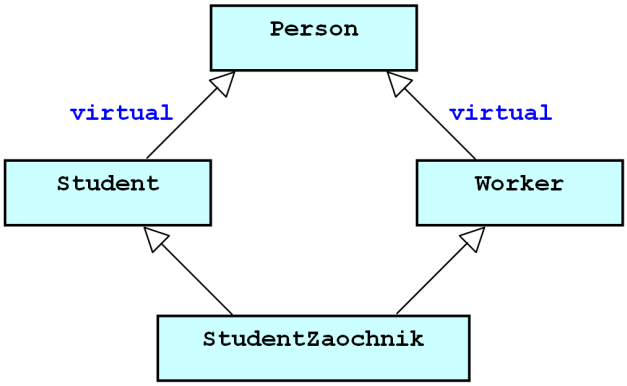


Рис. 4,5. Оголошення базового класу віртуальним для забезпечення коректності ромбічного успадкування в C++

Тепер тестова програма успішно компілюється і у вікні консолі показує очікуваний результат

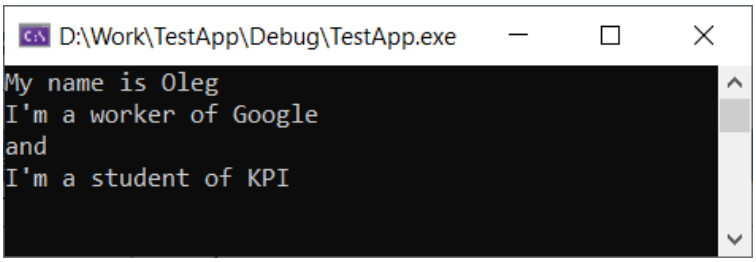


Рис. 4.6. Результат тесту ромбічного успадкування

Необхідно зазначити, що схема ромбічного успадкування виникає достатньо часто при використанні множинного успадкування в побудові ієрархій типів.

Розглянемо наступний приклад. Нехай потрібно створити класи для графічного відображення наступних форм

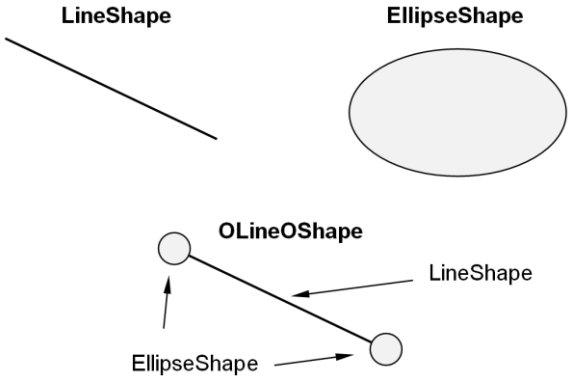


Рис. 4.7. Три типи форм

Нехай класи LineShape та EllipseShape будуть похідними від базового абстрактного класу Shape. Для OLineOShape множинне спадкування – від класів LineShape та EllipseShape. Для відображення форм будемо поліморфно викликати метод **`show`**, який оголошено у базовому класі Shape.

```
class Shape
{
protected:
   long xs1,ys1,xs2,ys2;
public:
   virtual ~Shape() { }
   void Set(long x1,long y1,long x2,long y2) {
      xs1 = x1; ys1 = y1; xs2 = x2; ys2 = y2;
   }
   virtual void show(HDC) = 0;
};
```

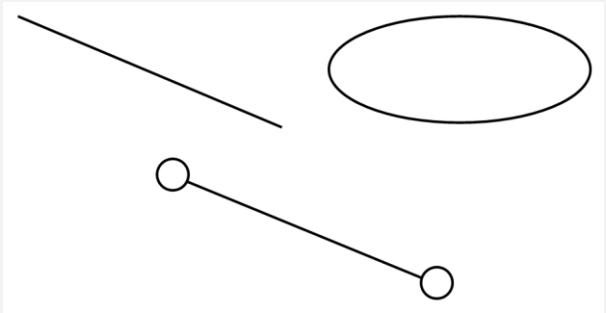
Оголошення похідних класів

```
class LineShape : public virtual Shape
{
public:
   virtual void show(HDC);
};

class EllipseShape : public virtual Shape
{
public:
   virtual void show(HDC);
};

class OLineOShape : public LineShape, public EllipseShape
{
public:
   virtual void show(HDC);
};
```

Задля скорочення викладення не будемо тут розглядати програмний код визначення методів **show**. Далі приклад тесту

```
Shape *p;
p = new LineShape;
if (p)
   {
   p->Set(30,30,200,100);
   p->Show(hdc);
   delete p;
   }
p = new EllipseShape;
if (p)
   {
   p->Set(230,30,400,100);
   p->Show(hdc);
   delete p;
   }
p = new OLineOShape;
if (p)
   {
   p->Set(130,130,300,200);
   p->Show(hdc);
   delete p;
   }
```

Результат



Підведемо підсумки щодо множинного успадкування.

Для забезпечення поліморфізму при наявності множинного спадкування повинен бути один базовий клас, щоб створити вказівник на нього. В загальному випадку у цьому базовому класі повинен бути віртуальний деструктор.

Для вирішення проблеми неоднозначності базовий клас повинен успадковуватися як віртуальний базовий клас.

Вказані проблеми реалізації множинного спадкування класів призвели до відмови від нього у деяких інших мовах програмування, зокрема, у Java, C# [10].

Контрольні запитання

1. Для чого призначене множинне успадкування?
2. Як оголошується множинне спадкування?
3. Які проблеми може створити наявність членів з однаковими іменами у різних базових класах?
4. Що таке ромбічне успадування і які проблеми воно спричиняє?
5. Що таке віртуальні базові класи і навіщо вони потрібні?
6. Чи є множинне спадкування класів в інших мовах програмування?

Лекція 5

Тема 2.5. Особливості програмування для операційної платформи Windows. Об'єктна орієнтованість системи

Побудова застосунків для Windows

Прикладні програми, які мають працювати в середовищі Windows, є програмними застосунками (додатками), керованими повідомленнями (“message driven“).

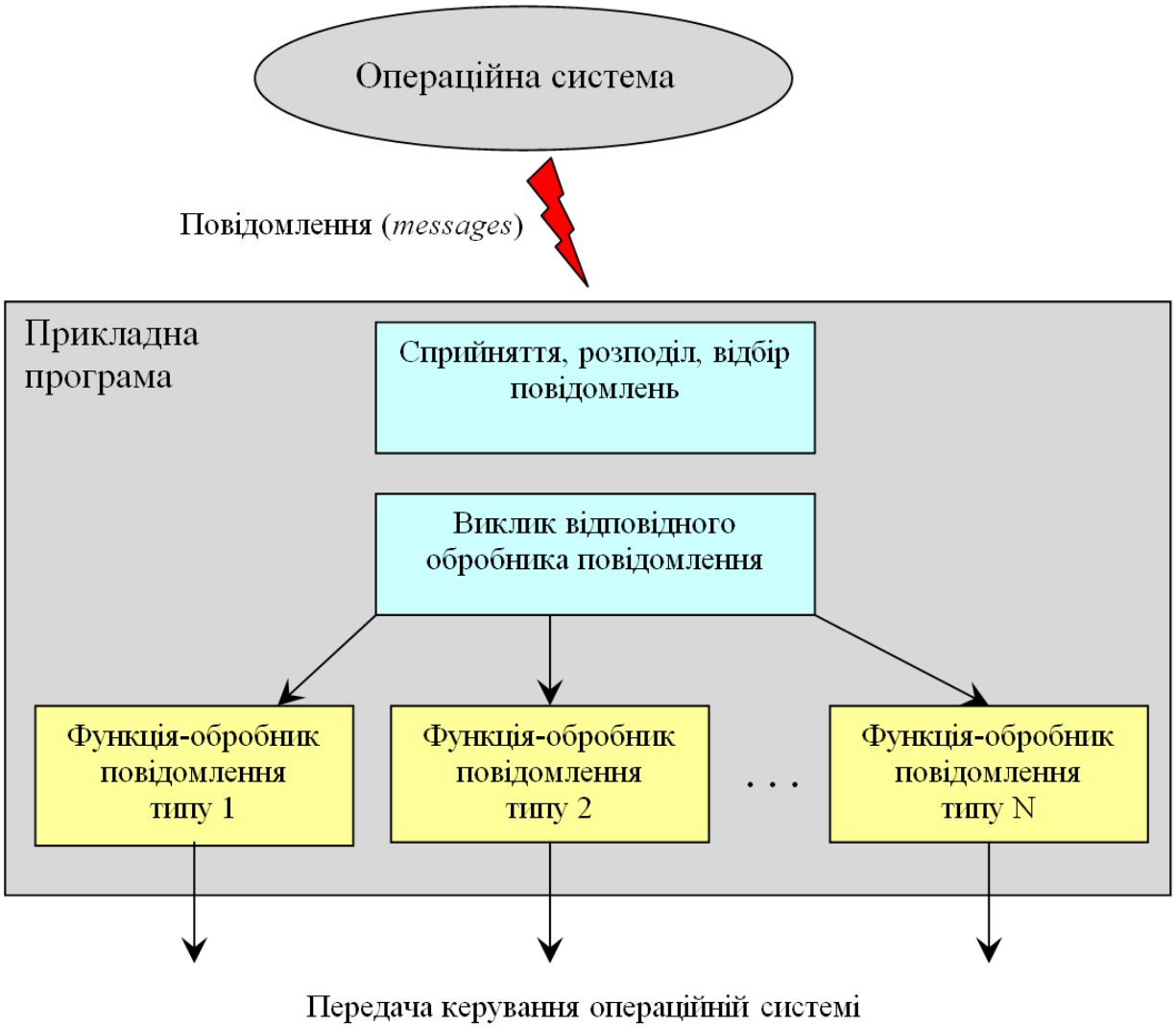


Рис. 5.1. Робота прикладної програми в середовищі ОС Windows

Після запуску прикладної програми на виконання, Windows починає надсилати їй повідомлення (”windows messages”). Це повідомлення

- про дії користувача
- про події у системі
- про завершення

Що таке повідомлення? В ОС Windows це безпосередній, або взятий з черги повідомлень виклик операційною системою функції вікна прикладної програми (callback-функції)

Функціональні можливості прикладної програми визначаються програмним кодом відповідних функцій-обробників повідомлень.

Розробник звичайної прикладної програми-застосунку для Windows (”Windows desktop application”) маж реалізувати у програмному коді типову послідовність кроків:

1. Зареєструвати клас вікна. При цьому, зокрема, вказати адресу callback-функції фікна, яка буде обробляти повідомлення.
2. Створити головне вікно програми.
3. Розпочати цикл очікування, прийняття та обробки повідомлень від Windows
4. Завершення роботи

Скелет типової програми проєкту C++ для Windows

Точкою входу у програму є функція C/C++ main. З неї починається виконання програми. В проєктах C++ для Windows вона зветься WinMain. В тілі цієї функції зосереджується програмний код щодо виконання усіх вказаних вище кроків за винятком обробників повідомлень.

```
int APIENTRY _tWinMain(HINSTANCE hInstance,
                       HINSTANCE hPrevInstance,
                       LPTSTR    lpCmdLine,
                       int       nCmdShow)
{
. . .   //тіло функції
return (int) msg.wParam;
}
```

Параметр **hInstance** є ідентифікатором екземпляру програми (*instance handle*). Це унікальне число, яке є ідентифікатором процесу

Параметр **hPrevInstance** мав би означати ідентифікатор попередньої копії програми, проте це вже не використовується і зараз завжди **hPrevInstance = 0.**

Параметр **lpCmdLine** вказує на рядок тексту з нулем наприкінці. Цей параметр означає командний рядок виклику програми і через нього можна передавати, наприклад, ім’я файлу, який програма повинна обробляти одразу після запуску.

Параметр **nCmdShow** – це число, яке вказує, у якому вигляді треба показувати вікно програми одразу після запуску (нормально, згорнуто, невидиме тощо).

Реєстрація класу головного вікна

```
WNDCLASSEX wcex;
wcex.cbSize = sizeof(WNDCLASSEX);
wcex.style      = CS_HREDRAW | CS_VREDRAW;
wcex.lpfnWndProc= WndProc;    //адреса callback-функції вікна
wcex.cbClsExtra = 0;
wcex.cbWndExtra = 0;
wcex.hInstance       = hInstance;
wcex.hIcon      = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_SIMPLE));
wcex.hCursor        = LoadCursor(NULL, IDC_ARROW);
wcex.hbrBackground  = (HBRUSH)(COLOR_WINDOW+1);
wcex.lpszMenuName    = MAKEINTRESOURCE(IDC_SIMPLE);
wcex.lpszClassName  = szWindowClass;
wcex.hIconSm        = LoadIcon(wcex.hInstance,
MAKEINTRESOURCE(IDI_SMALL));
RegisterClassEx(&wcex);
```

Створення головного вікна

```
hWnd = CreateWindow(szWindowClass,              //ім'я класу вікна
                szTitle,                        //текст у заголовку
                WS_OVERLAPPEDWINDOW,            //стиль вікна
                CW_USEDEFAULT, CW_USEDEFAULT,   //розташування
                CW_USEDEFAULT, CW_USEDEFAULT,   //розміри
                NULL, NULL, hInstance, NULL);

if (!hWnd) return FALSE;        //вікно створити не вдалося
ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);     //запит надіслати повідомлення WM_PAINT
```

Стиль вікна може визначатися комбінуванням бітових прапорців – символічних констант з префіксом WS_
Наприклад, для того, щоб створити вікно із повзунами скролінгу, достатньо у поле стилю записати

```
hWnd = CreateWindow(szWindowClass,
                    szTitle,
                    WS_OVERLAPPEDWINDOW | WS_HSCROLL | WS_VSCROLL,
                    . . .
```

Цикл повідомлень

Після того, як вікно створене, потрібно організувати цикл повідомлень для цього вікна.Зазвичай цикл повідомлення програмується на основі функції **GetMessage**

Найпростіший варіант циклу повідомлень:

```
MSG msg;
while (GetMessage(&msg, NULL, 0, 0))
  {
  TranslateMessage(&msg);
  DispatchMessage(&msg);
  }
```

Функція **GetMessage** бере повідомлення з черги повідомлень програми і записує його у структуру **msg**

Трохи складнішим є цикл повідомлень, у якому враховуються так звані акселератори – активні комбінації клавіш, натискування яких призводить до вибору відповідних пунктів меню

```
MSG msg;
HACCEL hAccelTable;
while (GetMessage(&msg, NULL, 0, 0))
  {
  if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
     {
     TranslateMessage(&msg);
     DispatchMessage(&msg);
     }
  }
```

Функція головного вікна

Після того як клас вікна зареєстрований, вікно створене, показане на екрані, програма увійшла у цикл повідомлень. Далі розпочинається реальна робота програми згідно її призначенню. Функція вікна WndProc обробляє усі повідомлення і фактично визначає те, що буде виконувати програма у відповідь на дії користувача.

Мінімальний код віконної функції. У такому коді явно записана обробка повідомлення WM_DESTROY, решта повідомлень обробляються функцією DefWindowProc

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
                         WPARAM wParam, LPARAM lParam)
{
switch (message)
   {
   case WM_DESTROY:
     PostQuitMessage(0);
     break;
   default: return DefWindowProc(hWnd, message, wParam, lParam);
  }
return 0;
}
```

Тип функції головного вікна – **LRESULT CALLBACK.** Що це означає? Для Win32 тип **LRESULT** – це тип **long**.

А слово **CALLBACK** – це перейменоване **__stdcall.** Слово **__stdcall** формально означає спосіб виклику та завершення процедур, прийнятий у Windows. Навіщо таке перейменування? У такий спосіб позначається, що функція вікна є функцією зворотного виклику (callback-функція).

Звичайний вигляд функції головного вікна

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
                         WPARAM wParam, LPARAM lParam)
{
int wmId, wmEvent;
PAINTSTRUCT ps;
HDC hdc;
switch (message)
   {
   case WM_LBUTTONDOWN:
      . . .
      break;
   case WM_LBUTTONUP:
      . . .
      break;
   case WM_MOUSEMOVE:
      . . .
      break;
   case WM_PAINT:
      hdc = BeginPaint(hWnd, &ps);
      . . .
      EndPaint(hWnd, &ps);
      break;
   case WM_DESTROY:
      PostQuitMessage(0);
      break;
   case WM_COMMAND:
      wmId    = LOWORD(wParam);
      wmEvent = HIWORD(wParam);
      switch (wmId)
         {
         case IDM_ABOUT:
           DialogBox(hInst, MAKEINTRESOURCE(IDD_ABOUTBOX), hWnd, About);
           break;
         case IDM_EXIT:
           DestroyWindow(hWnd);
           break;
         default: return DefWindowProc(hWnd, message, wParam, lParam);
         }
      break;
   default: return DefWindowProc(hWnd, message, wParam, lParam);
   }
return 0;
}
```

Програмни код обробників повідомлень потрібно вставляти між відповідних case та break. Але, зазвичай, обробники містять багато рядків вихідного тексту, тому краще оформлювати такі обробники власними функціями з іменами, наприклад onMessageName. Такі функції можуть бути окремими глобальними функціями, а можуть бути методами якогось класу.

Структурований код для функції вікна може мати наступний вигляд

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
                         WPARAM wParam, LPARAM lParam)
{
switch (message)
   {
   case WM_LBUTTONDOWN:
      OnLBdown(hWnd);
      break;
   case WM_LBUTTONUP:
      OnLBup(hWnd);
      break;
   case WM_MOUSEMOVE:
      OnMouseMove(hWnd);
      break;
   case WM_PAINT:
      OnPaint(hWnd);
      break;
   case WM_DESTROY:
      PostQuitMessage(0);
      break;
   case WM_COMMAND:
      return OnCommand(hWnd, message, wParam, lParam);

   default: return DefWindowProc(hWnd, message, wParam, lParam);
   }
return 0;
}
```

Дочірні вікна керування

Дочірнє вікно керування (*child window control*) обробляє повідомлення миші та клавіатури і повідомляє батьківське вікно про те, що стан дочірнього вікна змінилося. У цьому випадку дочірнє вікно буде для батьківського вікна пристроєм вводу.

Можна створювати власні дочірні вікна керування, проте вже існують вже визначені, стандартні класи дочірніх вікон.

Такими стандартними дочірніми вікнами керування є:

- кнопка (button)
- прапорець опції (checkbox)
- вікно редагування (edit)
- вікно списку (listbox)
- вікно комбінованого списку (combobox)
- повзун скролінгу (scrollbar)
- інші

Взаємодія з такими елементами на основі повідомлень. Віконна функція батьківського вікна надсилає повідомлення дочірнім вікнам керування, а ті надсилають повідомлення батьківському вікну. Для

стандартних дочірніх вікон є власні віконні функції – програмісту немає потреби їх створювати.

Дочірні вікна керування найчастіше використовують у вікнах діалогу. Розташування та розміри дочірніх вікон керування записують у файлах ресурсів.

Дочірні вікна керування можна також розташувати на поверхні головного вікна програми. Це робиться викликом функції CreateWindow. Створення дочірніх вікон керування на головному вікні зручно робити при обробці повідомлення WM_CREATE, яке обробляється на початку роботи новоствореного вікна. Розглянемо це докладніше на прикладі.

Нехай потрібно на поверхню головного вікна розташувати наступні дочірні вікна керування: 2 кнопки (button), елемент вводу тексту (edit) та список (listbox). Оголосимо ідентифікатори цих елементів

```
//--ідентифікатори дочірніх вікон головного вікна--
HWND hWndMain  = 0,
     hWndButn1 = 0,
     hWndButn2 = 0,
     hWndEdit  = 0,
     hWndList  = 0;
```

В коді функції головного вікна запишемо створення дочірніх вікон та обробники їхніх повідомлень

```
//--функція головного вікна програми--
LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
                         WPARAM wParam, LPARAM lParam)
{
. . .
switch (message)
   {
   case WM_CREATE:
      CreateMyControlsMainWindow(hWnd);      //створення дочірніх вікон
      break;

   case WM_COMMAND:
      wmId    = LOWORD(wParam);
      wmEvent = HIWORD(wParam);
      switch (wmId)
         {
         case 1:     //обробка повідомлення натискування 1-ї кнопки
            MessageBox(hWnd,"Натиснуто першу кнопку...","Мої вітання!",MB_OK);
            break;
. . .
```

Створення дочірніх вікон запрограмуємо у вигляді власної функції **CreateMyControlsMainWindow**

```
void CreateMyControlsMainWindow(HWND hWnd)
{
    HINSTANCE hi;
    RECT rc;
    hi = (HINSTANCE)GetWindowLong(hWnd, GWL_HINSTANCE);
```

```
    GetClientRect(hWnd, &rc);
    hWndButn1 = CreateWindowA("button", "Перша кнопка",
        WS_CHILD | WS_VISIBLE | WS_BORDER | BS_PUSHBUTTON,
        0, 0, 150, 30,
        hWnd, (HMENU)1, hi, NULL);
    hWndButn2 = CreateWindowA("button", "Друга кнопка",
        WS_CHILD | WS_VISIBLE | WS_BORDER | BS_PUSHBUTTON,
        150, 0, 150, 30,
        hWnd, (HMENU)2, hi, NULL);
    hWndEdit = CreateWindowA("edit", "<введіть текст>",
        WS_CHILD | WS_VISIBLE | WS_BORDER | BS_PUSHBUTTON,
        0, 30, 300, 25,
        hWnd, (HMENU)3, hi, NULL);
    hWndList = CreateWindowA("listbox", NULL,
        WS_CHILD | WS_VISIBLE |
        WS_BORDER | WS_VSCROLL |
        WS_HSCROLL | LBS_NOTIFY |
        LBS_NOINTEGRALHEIGHT,
        0, 55, 300, rc.bottom - rc.top - 55,
        hWnd, (HMENU)4, hi, NULL);
    PostMessage(hWndList, LB_SETHORIZONTALEXTENT, (WPARAM)1500, 0L);
}
```

Для перевірки натискуємо першу кнопку – спрацьовує обробник повідомлення WM_COMMAND з параметром wParam = 1. Ідентифікатор цієї кнопки визначено в CreateWindowA як (HMENU)1. Так само, треба написати обробники повідомлень для інших елементів.

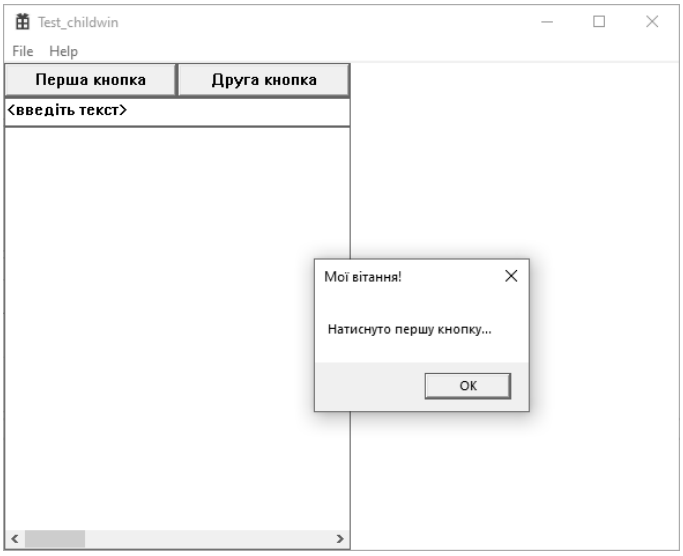


Рис. 5.2. Результат розташування елементів у головному вікні

Дочірнє вікно Toolbar

Доволі часто у програмах можна побачити стрічку з кнопками, які розташовуються на головному вікні нижче головного меню. Ці кнопки складають набір інструментів. Стрічка з такими кнопками є дочірнім вікном Toolbar. Для створення вікна Toolbar зручно використати бібліотеку

COMCTL32 елементів керування загального користування (*Common Control Library*).

Для того, щоб використати якійсь елемент з цієї бібліотеки, потрібно включити заголовочний файл

```
#include <commctrl.h>
```

Крім того, лінкеру потрібно посилання на статичну бібліотеку **COMCTL32.** Це можна зробити так:

```
#pragma comment(lib, "comctl32.lib")
```

При роботі програми потрібно спочатку викликати функцію **InitCommonControls,** а потім вже можна створювати Toolbar потрібного вигляду. Розглянемо це на прикладі. Нехай потрібно створити Toolbar з 4 стандартними кнопками: File new, File open, File save, File print

```
void OnCreate(HWND hWnd)
{
    TBBUTTON tbb[4];

    ZeroMemory(tbb, sizeof(tbb));
    tbb[0].iBitmap = STD_FILENEW;            //стандартне зображення
    tbb[0].fsState = TBSTATE_ENABLED;
    tbb[0].fsStyle = TBSTYLE_BUTTON;         //тип елементу - кнопка
    tbb[0].idCommand = ID_TOOL_FILE_NEW;  //цей ID буде у повідомленні
                                          //WM_COMMAND
    tbb[1].iBitmap = STD_FILEOPEN;
    tbb[1].fsState = TBSTATE_ENABLED;
    tbb[1].fsStyle = TBSTYLE_BUTTON;
    tbb[1].idCommand = ID_TOOL_FILE_OPEN;

    tbb[2].iBitmap = STD_FILESAVE;
    tbb[2].fsState = TBSTATE_ENABLED;
    tbb[2].fsStyle = TBSTYLE_BUTTON;
    tbb[2].idCommand = ID_TOOL_FILE_SAVEAS;

    tbb[3].iBitmap = STD_PRINT;
    tbb[3].fsState = TBSTATE_ENABLED;
    tbb[3].fsStyle = TBSTYLE_BUTTON;
    tbb[3].idCommand = ID_TOOL_FILE_PRINT;

    hwndToolBar = CreateToolbarEx(hWnd,          //батьківське вікно
        WS_CHILD | WS_VISIBLE | WS_BORDER | WS_CLIPSIBLINGS | CCS_TOP,
        IDC_MY_TOOLBAR,   //ID дочірнього вікна Toolbar
        1, HINST_COMMCTRL, IDB_STD_SMALL_COLOR,
        tbb,                //масив опису кнопок
        4,                  //кількість кнопок
        0, 0, 0, 0,         //розташування та розміри
        sizeof(TBBUTTON));
}
```

Створення Toolbar зручно виконувати при обробці повідомлення WM_CREATE у функції головного вікна. Далі розглянемо скелет функції головного вікна

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
                         WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
    case WM_CREATE:
        OnCreate(hWnd);
        break;
    case WM_COMMAND:
        {
            int wmId = LOWORD(wParam);
            switch (wmId)
            {
            case ID_FILE_NEW:                //ID пункта меню
            case ID_TOOL_FILE_NEW:           //ID кнопки Toolbar
                OnFileNew(hWnd);
                break;
            case ID_FILE_OPEN:
            case ID_TOOL_FILE_OPEN:
                OnFileOpen(hWnd);
                break;
            case ID_FILE_SAVEAS:
            case ID_TOOL_FILE_SAVEAS:
                OnFileSaveAs(hWnd);
                break;
            case ID_FILE_PRINT:
            case ID_TOOL_FILE_PRINT:
                OnFilePrint(hWnd);
                break;
. . .
```

Реакцію на натискування кнопок забезпечують відповідні обробники. В якості теста, наприклад, обробник OnFileNew запишемо наступним чином

```
void OnFileNew(HWND hWnd)
{
    SendMessage(hwndToolBar, TB_PRESSBUTTON, ID_TOOL_FILE_NEW, 1);
    MessageBoxA(hWnd, "User selects File New", "Mode selection test",
                MB_OK);
    SendMessage(hwndToolBar, TB_PRESSBUTTON, ID_TOOL_FILE_NEW, 0);
}
```

Результат такого тесту наведено нижче

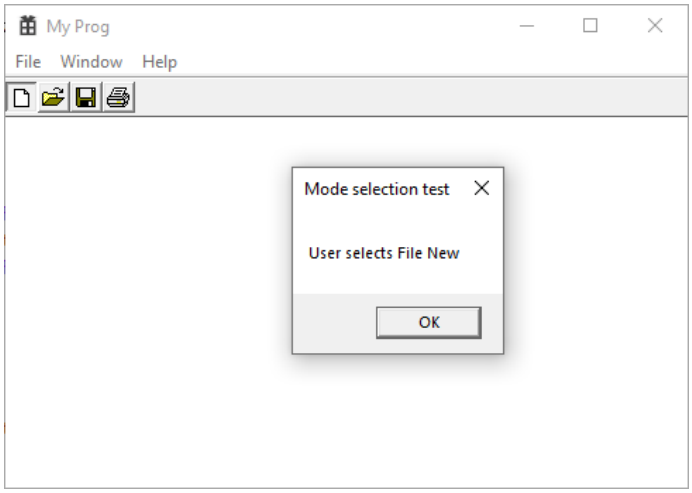


Рис. 5.3. Результат натискування кнопки File new у тулбарі

Модальні вікна діалогу

Модальні вікна зазвичай використовуються у програмах для того, щоб користувач вводив якісь параметри або для інформування користувача про якісь події. Вони звуться модальними (*modal*), тому що після своєї появи таке вікно зосереджує фокус вводу на собі і не дає можливості переключитися на інші вікна даної програми аж доти не закриється це вікно діалогу.

Стандартом CUA (Common User Access) вимагається, зокрема, що кожне таке вікно повинне містити кнопки Ok та Cancel22]. Обидві ці кнопки мають закривати вікно діалогу, але Ok підтверджує сприйняття введених користувачем даних, а Cancel означає відміну.

Створення модального вікна діалогу можна зробити викликом функції DialogBox.

```
result = DialogBox(hInst,MAKEINTRESOURCE(IDD_DIALOG),hWnd, DlgProc);
```

де `hInst` – ідентифікатор програми, `IDD_DIALOG` – ідентифікатор ресурсу вікна, `hWnd` – хендл батьківського вікна (наприклад, головного вікна), `DlgProc` – ім'я callback-функції вікна. Функція **`DialogBox`** повертає цілочисельне значення результату, яке можна використати для аналізу дій користувача, наприклад, якщо result = 0, то це означатиме, що користувач натиснув Cancel, а значення result = 1 спонукає сприйняття введених даних у вікні діалогу.

Нижче наведена типова структура callback-функції вікна діалогу

```
BOOL CALLBACK DlgProc(HWND hDlg,UINT iMessage,
                      WPARAM wParam, LPARAM lParam)
{
switch (iMessage)
   {
   case WM_INITDIALOG:
      //...     початкова ініціалізація елементів вікна
      return (INT_PTR)TRUE;
   case WM_COMMAND:
      if (LOWORD(wParam) == IDOK)  //якщо натиснуто кнопку Ok
          {
          //... сприйняття введеної користувачем інформації
          EndDialog(hDlg, 1);
          break;
          }
      if (LOWORD(wParam) == IDCANCEL) EndDialog(hDlg, 0);
      break;
   default : break;
   }
return FALSE;
}
```

Можна побачити багато спільного цієї функції з функцією головного вікна програми (WndProc) – також чотири параметри та розбір повідомлень оператором switch. Головні відмінності: функція завершує роботу після виклику EndDialog, повертає значення типу BOOL, обробляє повідомлення WM_INITDIALOG.

Розглянемо приклад простого вікна діалогу зі стрічкою вводу тексту Edit Control. Нехай у редакторі ресурсів ми створили вікно діалогу зі стрічкою вводу тексту. Ця стрічка – елемент Edit Control, може мати ідентифікатор, наприклад, IDC_EDIT1.

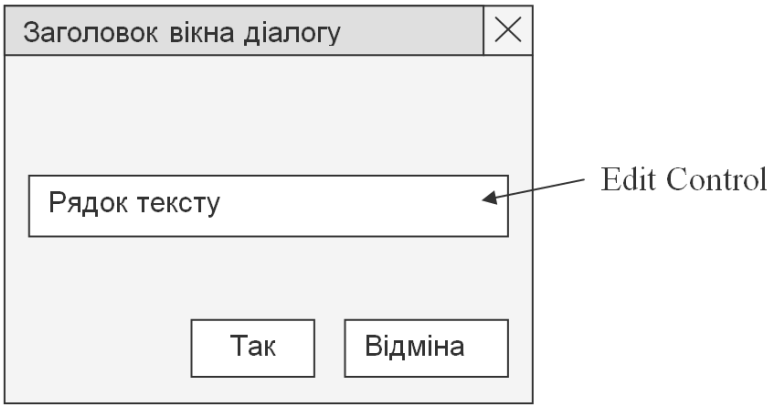


Рис. 5.4. Вікно діалогу зі стрічкою вводу тексту Edit Control

Потрібно запрограмувати, щоб після натискування кнопки Так (Ok) програма вихоплювала текст з цієї стрічки і записувала в якийсь буфер програми. Щоб вихопити текст з Edit Control можна скористатися функцією GetDlgItemText. Програмний код для callback-функції вікна діалогу нижче

```
BOOL CALLBACK DlgProc(HWND hDlg,UINT iMessage,WPARAM wParam, LPARAM)
{
switch (iMessage)
  {
  case WM_COMMAND:
    if (LOWORD(wParam) == IDOK)      //якщо натиснуто кнопку Ok
       {
       GetDlgItemText(hDlg,IDC_EDIT1, буфер, кількість символів);
       EndDialog(hDlg, 1);
       break;
       }
    if (LOWORD(wParam) == IDCANCEL) EndDialog(hDlg, 0);
    break;
  default : break;
  }
return FALSE;
}
```

Немодальні вікна діалогу

Немодальне (*modeless*) вікно діалогу не вимагає, щоб його закрили для переключення фокусу вводу на інше вікно програми. Воно може містити різноманітні елементи для вводу (керування) а також кнопку Close. Такі вікна можуть достатньо самостійно існувати упродовж роботи програми, не блокуючи роботу інших вікон. Немодальні вікна варто робити у випадках, коли потрібно локалізувати у окремому вікні якісь поточні параметри вводу, наприклад, у такому вікні може бути палітра для вибіру кольорів малювання об’єктів у графічному редакторі. Інший приклад – вікно пошуку тексту в стандартному блокноті Windows або програмі Word.

Як створити немодальне вікно діалогу? Для створення немодального вікна діалогу можна скористатися функцією CreateDialog, яка належіть до функцій Windows API. Нижче приклад створення немодального вікна у власній функції ActivateDlgWindow

```
HWND hWndDlg = NULL;      //хендл немодального вікна діалогу
. . .

void ActivateModeless(HWND hWnd)
{
if (hWndDlg)   //якщо це вікно діалогу вже є, то закриваємо його
   {
   PostMessage(hWndDlg, WM_SYSCOMMAND, SC_CLOSE, 0);
   return;
   }
hWndDlg = CreateDialog(hInst, MAKEINTRESOURCE(IDD_DIALOG1), hWnd, DlgProc);
}
```

При створенні вікна можна передати функції CreateDialog у якості другого параметру ідентифікатор ресурсу вікна діалогу. Якщо на поверхні такого вікна мають бути якісь елементи керування – кнопки, вікна списків,

перемикачі, радіокнопки тощо, то спочатку потрібно у редакторі ресурсів внести ці елементи у файл ресурсів так само, як і для модального вікна. У якості прикладу зробимо вікно з двома кнопками і елементом EditText.

```
IDD_DIALOG1 DIALOGEX 200, 150, 233, 57
STYLE DS_SETFONT | DS_MODALFRAME | DS_FIXEDSYS | WS_POPUP | WS_VISIBLE |
WS_CAPTION | WS_SYSMENU
CAPTION "Modeless dialog"
FONT 10, "MS Shell Dlg", 400, 0, 0xCC
BEGIN
    DEFPUSHBUTTON   "Apply",IDOK,133,40,43,12
    PUSHBUTTON      "Close",IDCANCEL,183,40,43,12
    EDITTEXT        IDC_EDIT1,17,15,197,16,ES_AUTOHSCROLL
END
```

Примітка: в описі вікна треба поставити опцію **WS_VISIBLE**, інакше вікно не буде показуватися.

Оскільки повідомлення, які надсилаються для немодального вікна діалогу проходять через чергу повідомлень програми, то для коректної роботи програми треба вдосконалити головний цикл отримання та обробки повідомлень наступним чином

```
HACCEL hAccelTable = LoadAccelerators(hInstance, ...);
MSG msg;
//головний цикл повідомлень програми
while (GetMessage(&msg, nullptr, 0, 0))
   {
   if ((hWndDlg == 0) || (!IsDialogMessage(hWndDlg, &msg)))

      //це виконується, якщо повідомлення не для немодального вікна
      if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
         {
         TranslateMessage(&msg);
         DispatchMessage(&msg);
         }
   }
```

Приклад callback-функції немодального вікна діалогу

```
BOOL CALLBACK DlgProc(HWND hDlg, UINT message,
                      WPARAM wParam, LPARAM lParam)
{
switch (message)
   {
   case WM_SYSCOMMAND:
      if (wParam == SC_CLOSE) DestroyWindow(hDlg);
      break;
   case WM_COMMAND:
      if (LOWORD(wParam) == IDOK)   //натиснуто кнопку Apply
         {
         Apply(hDlg);     //щось робимо
         break;
         }
```

```
      if (LOWORD(wParam) == IDCANCEL)     //натиснуто кнопку Close
         DestroyWindow(hDlg);
      break;
   case WM_DESTROY:          //знищення вікна
      hWndDlg = NULL;
      break;
   default: break;
   }
return FALSE;
}
```

Для закриття такого вікна діалогу будемо використовувати окрему кнопку Close а також кнопку-хрестик у правому верхньому куті вікна – там розташовується системне меню. Ми запрограмували, щоб при натискуванні вказаних кнопок робився виклик функції DestroyWindow.

Також передбачена кнопка Apply, при натискуванні якої виконуються якісь дії, наприклад, береться текст з елемента EditText вікна діалогу і відображається у головному вікні.

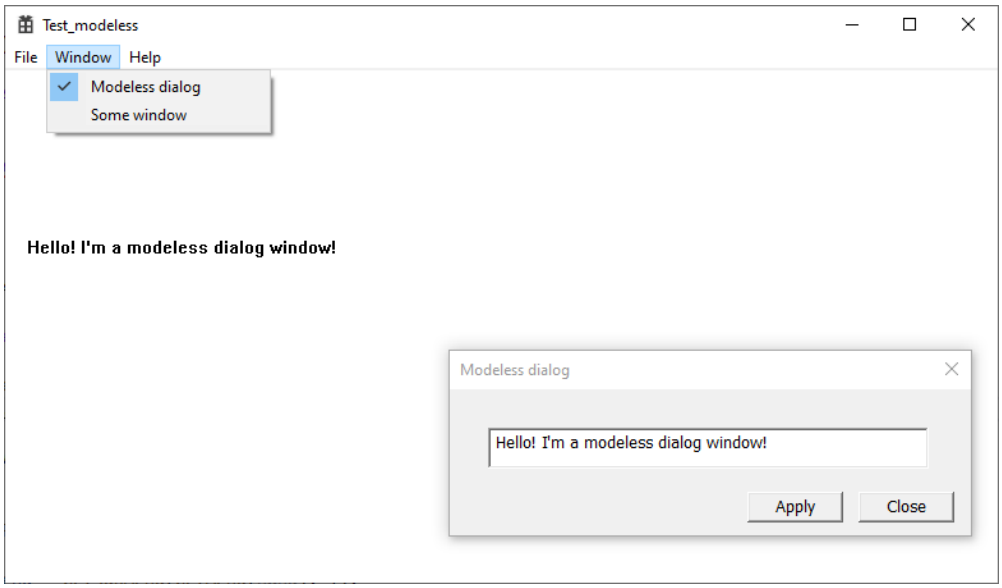


Рис. 5.5. Приклад програми з немодальним вікном діалогу

Примітка. При розробці програм з декількома вікнами для керування активізацією та закриттям додаткових вікон зазвичай передбачають меню Window яке, згідно стандарту CUA, має розташовуватися передостаннім праворуч – ліворуч від меню Help. Якщо деяке вікно активне, тоді автоматично позначається відповідний пункт меню Window. Такі позначки можна робити викликом функції CheckMenuItem [1].

Щодо об'єктної орієнтованості Windows

В середовищі ОС Windows функціонують об’єкти, які є екземплярами класу вікна і керуються та взаємодіють шляхом обміну повідомлень. Кожний тип вікна – головне вікно програми, діалогові вікна, дочірні елементи керування мають спільні риси. Відповідно, програмісту потрібно реалізувати вибір класу вікна, визначення його властивостей та записати приймання, обробку та надсилання повідомлень для об'єктів програми.

Незважаючи на об'єктну орієнтованість побудови самої ОС, інтерфейс програмування – API Windows зроблений у вигляді множини функцій. Але це не означає відмову від об'єктно-орієнтованого стилю програмування для розробників програмних додатків. Кожен з них обиратиме таку парадигму розробки власного проєкта, яка може забезпечити йому успіх. Використання ООП для створення програмних додатків для Windows є вельми доречним.

Контрольні запитання

1. Що таке повідомлення?
2. Що таке застосунок "message driven"?
3. Які особливості побудови програмного забезпечення для Windows?
4. Які складові програмного коду програмного застосунку для Windows?
5. Навіщо потрібна реєстрація класу головного вікна?
6. Що робить функція вікна?
7. Що таке дочірнє вікно керування і хто є батьком для нього?
8. Як створити дочірнє вікно на головному вікні?
9. Що таке модальне вікно діалогу?
10. Чим відрізняється немодальне вікно діалогу від модального?
11. Як створити дочірнє вікно на вікні діалогу?

Лекція 6

Тема 3.1. Діаграми UML

Уніфікована мова моделювання UML (Unified Modeling Language) – це стандарт опису структури, поведінки та інших аспектів програмних (і не тільки) систем у графічному вигляді. Містить множину типів діаграм. Деякі з цих діаграм призначені для відображення реалізації об'єктно-орієнтовнаого підходу до проектування систем. Остання версія стандарту UML – редакція 2.5.1 була оприлюднена на сайті OMG у грудні 2017 року [18].

Основні різновиди UML–моделей

Опис системи мовою UML з п’яти взаємопов’язаних точок зору можна представити наступним чином

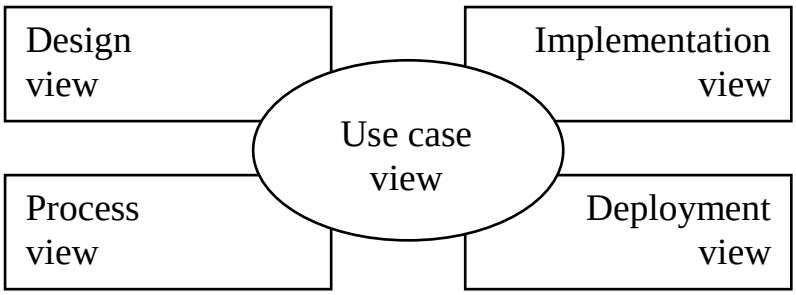


Рис. 6.1. Основні різновиди UML–моделей

Представлення з точки зору прецедентів (Use case view) описують систему з погляду кінцевого користувача, аналітика, тестера. Такі моделі не відображабть структуру програмного забезпечення, а описують загальне ставлення до системі. Дане подання насамперед декларує функціональні вимоги до системи, значення сервісів, які система має надати кінцевому користувачеві.У UML це відображається за допомогою діаграм прецедентів (Use case diagram), динамічний аспект представлений у діаграмах взаємодій (Interaction diagram), станів (Statechart diagram), активності (Activity diagram).

Представлення з точки зору дизайну (Design view) включає класи, інтерфейси та кооперації. У UML це відображається за допомогою діаграм класів (Class diagram) та об'єктів (Object diagram), динамічний аспект відображається у діаграмах взаємодій, станів, активності.

Представлення з точки зору процесів (Process view) включає потоки та процеси, які формують паралельну обробку та синхронізацію в системі. Дане уявлення в першу чергу відноситься до продуктивності, масштабованості та пропускної спроможності системи. У UML статичний та

динамічний аспекти відображаються тими ж діаграмами, що й у Design view, але увага акцентується на активних класах, що представляють процеси та потоки.

Представлення з точки зору реалізації (Implementation view) включає компоненти та файли, що використовуються при складанні системи. Подібне уявлення в першу чергу стосується управління конфігураціями (Configuration management) релізів продукту. Статичний аспект у UML відображено діаграмою компонентів (Component diagram), а динамічний – діаграмами взаємодій, станів, активності.

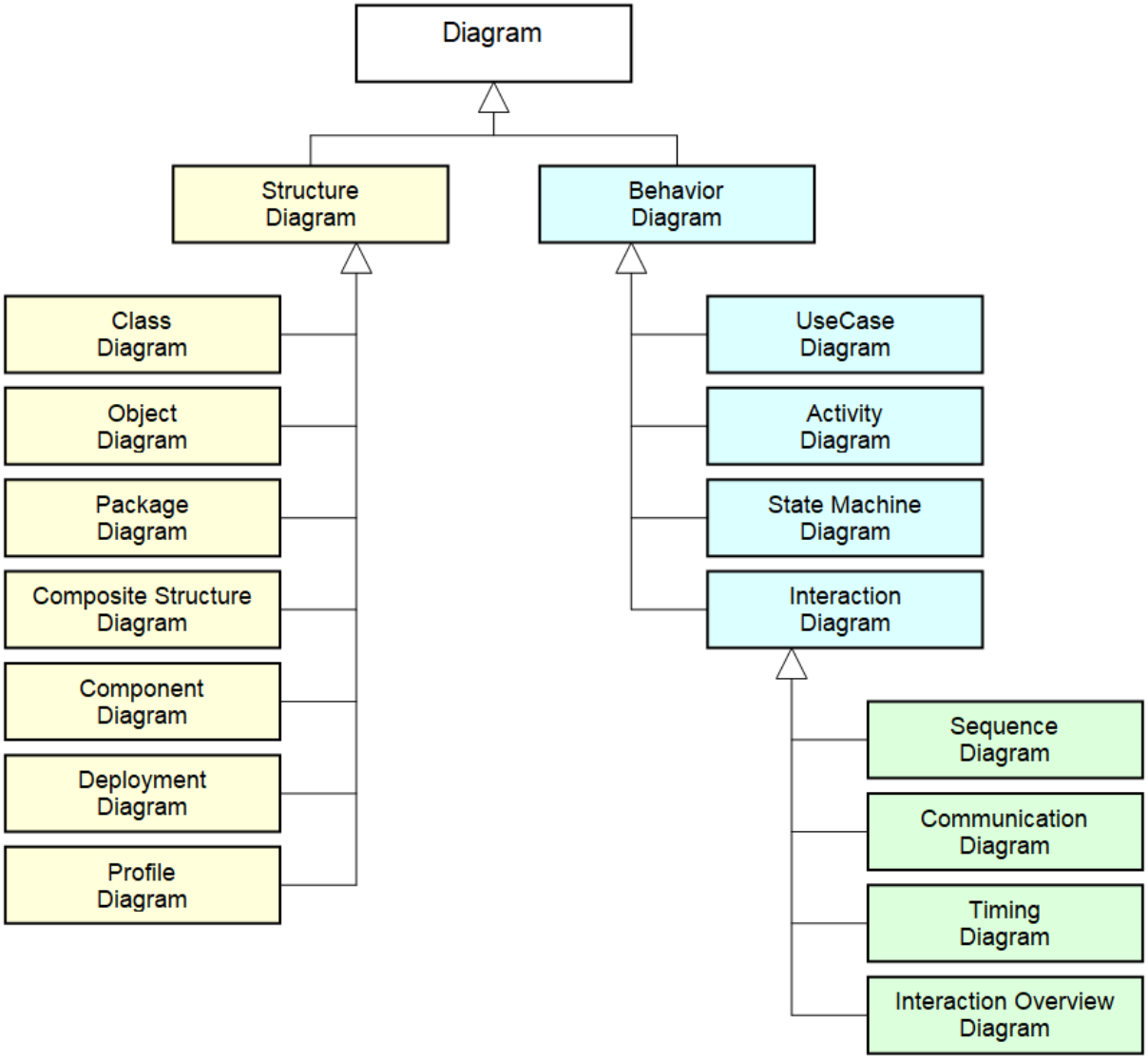
Представлення з точки зору впровадження (Deployment view) включає вузли та їхню взаємодію — вони визначають апаратну топологію, де виконується програмне забезпечення. Це уявлення насамперед відноситься до поширення, доставки, встановлення компонентів, з яких будується фізична система. Статичний аспект у UML відображається діаграмою впровадження (Deployment diagram), а динамічний діаграмами взаємодій, станів, активності.

Щодо різноманіття UML-моделей.

Складну систему можна і потрібно представляти у вигляді набору невеликих і майже незалежних моделей-діаграм, причому жодна з них не є достатньою для опису системи та отримання повного уявлення про неї, оскільки кожна з них фокусується на певному аспекті функціонування системи і висловлює різний рівень абстракції. Іншими словами, кожна окрема модель відповідає певній, приватній точці зору на проектовану систему.

Кількість типів діаграм для моделювання конкретної програми не є фіксованою. Для простих програм немає необхідності будувати все без винятку діаграми. Наприклад, для локальної програми не обов'язково будувати діаграму розгортання. Перелік діаграм залежить від специфіки проекту, що розробляється, і визначається, у першу чергу, його розробником.

Огляд діаграм UML



Структурні діаграми (Structure Diagrams)

- Класів (Class diagram)
- Об'єктів (Object diagram)
- Пакетів (Package diagram)
- Композитної структури (Composite structure diagram)
- Компонентів (Component diagram)
- Розгортання (Deployment diagram)
- Профілів (Profile diagram)

Діаграми поведінки (Behavior Diagrams)

- Прецедентів (Use case diagram)
- Діяльності (Activity diagram)
- Станів (State Machine diagram)
- Взаємодії (Interaction diagram)
- Послідовності (Sequence diagram)
- Комунікації (Communication diagram)
- Синхронізації (Timing diagram)
- Огляду Взаємодії (Interaction Overview diagram)

Рис. 6.1. Різновиди діаграм UML

Діаграми прецедентів

Головними елементами таких діаграм є **актори** та **прецеденти**.

Актори

Актор (від англ. *action*) – це хтось або щось, що взаємодіє із даною системою як зовнішній чинник. На діаграмах прецедентів актор позначається такою фігуркою

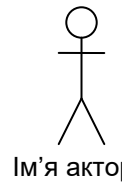


Рис. 6.2. Умовна позначка для актора

Актором може бути, наприклад, клієнт платіжної системи. А може бути якійсь зовнішній пристрій, під’єднаний до системи, або якась програма, яка робить запит на отримання деяких даних тощо. Таким чином, незважаючі на «людське» позначення, актором не обов’язково має бути людська істота. Головне – актор знаходиться за межами системи.

Прецеденти

Прецедент – це позначення якихось дій, процесів, які відбуваються в системі, і які важливі у певному контексті розгляду. Іншими словами, прецеденти можуть описувати якісь варіанти використання (use-case) чогось у системі. Прецеденти описують, що саме виконується у системі при взаємодії з акторами – що актор хоче зробити, що актор повинен зробити, яких дій чекати від системи у відповідь на звернення актора тощо. Зазвичай у якості прецедентів виступають події пов’язані з акторами, хоча можуть бути прецеденти, які безпосередньо з акторами і не пов’язані.

Графічне позначення прецедента – еліпс з ім’ям прецеденту або описом дій

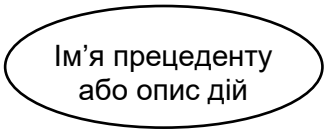


Рис. 6.2. Умовна позначка прецедента

Актор зв’язується з прецедентом *асоціацією*. На діаграмі асоціація позначається лінією між актором та прецедентом.

Нижче приклад простої діаграми прецедентів, яка описує взаємодію основних користувачів з деякою системою підтримки освітнього процесу

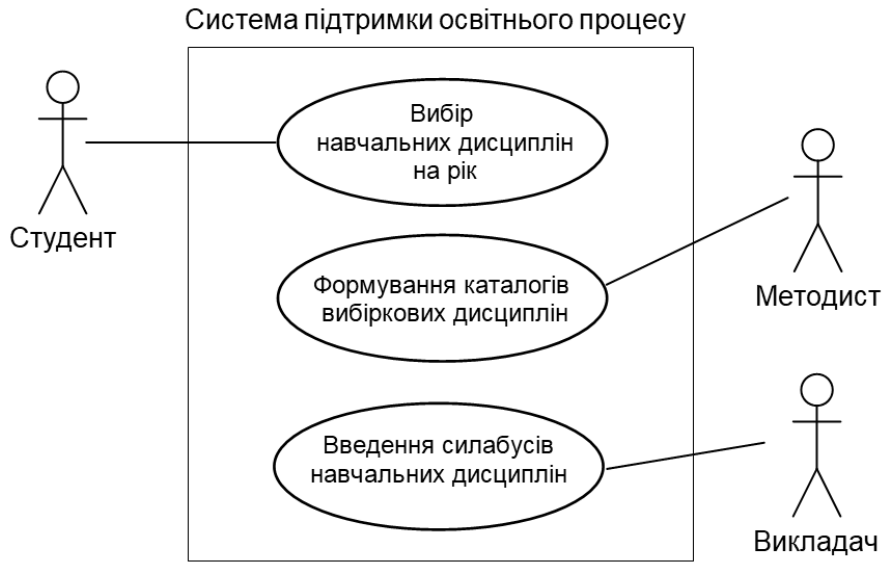


Рис. 6.3. Приклад простої діаграми прецедентів

Діаграми прецедентів є зручним та потужним засобом для наочного, з потрібною деталізацією, опису функціоналу системи та взаємодії акторів з системою. Наочність моделі сприяє аналізу системи.

Розгляд навіть такого простого прикладу може призвести до багатьох запитань. Так, наприклад, яка роль методиста у системі і які у нього функції? Інше запитання: наскільки автоматизованим може бути формування каталогу вибіркових дисциплін?

Важливим питанням є таке: що знаходиться у межах системи, а що зовні? Діаграми прецедентів є одним із засобів допомоги у вирішенні цього часто нетривіального завдання. На діаграмі прецедентів можна явно показати межу системи, але часто її не малють – тоді вважається, що усе, що зображено на діаграмі прецедентів, окрім акторів, належіть системі.

Відношення між прецедентами

Між прецедентами можуть бути наступні відношення:

- включення (include)
- розширення (extend)
- узагальнення (generalization)

Відношення між прецедентами можуть бути використані для забезпечення структурованості та потрібної деталізації моделі, для спрощення аналізу системи, для зручності її реалізації.

Включення прецедентів

Якщо у деякому прецеденті А можна виділити якусь частку, яку завжди необхідно робити, то цю підмножину дій можна назвати прецедентом В і відобразити усе це наступним чином

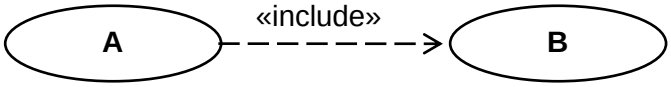


Рис. 6.3. Включення прецедентів

Необхідно зазначити, що прецедент А став вже неповним сам по собі і для свого виконання потребує завершення усіх дій прецедента В.

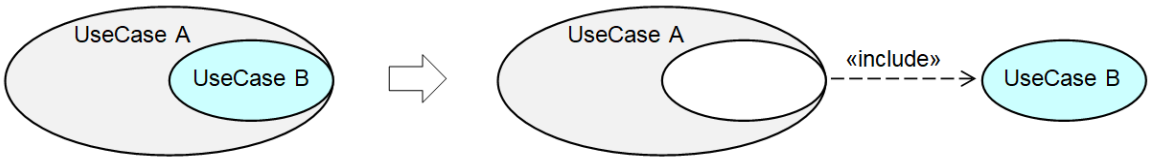


Рис. 6.4. Ілюстрація відношення включення прецедентів. Використане джерело https://www.uml-diagrams.org/use-case-include.html

Включення прецедентів подібно до того, як в ході виконання якоїсь функції викликається одна або декілька інших функцій

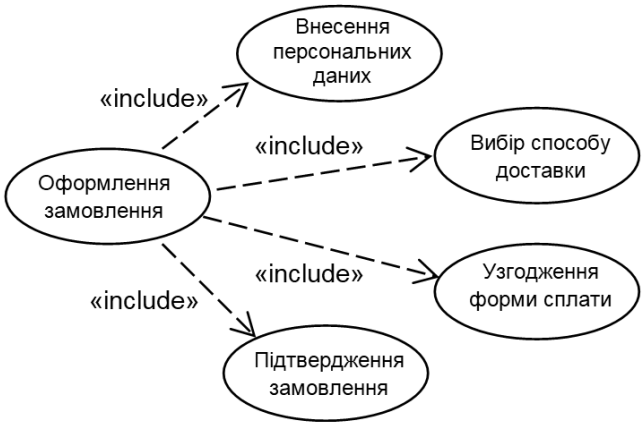


Рис. 6.5. Приклад включення прецедентів

Якщо два або більше прецедента мають деяку спільну поведінку, тоді з них можна виділити таки дії і оформити їх як спільний прецедент

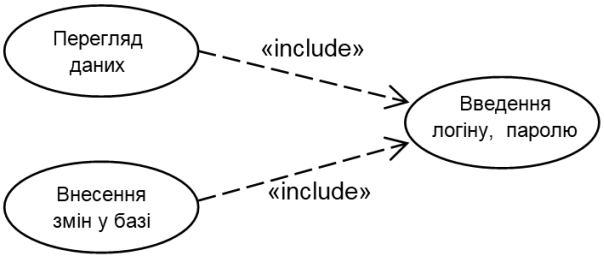


Рис. 6.6. Приклад виділення спільного прецеденту

Знаходження подібностей поведінки і, відповідно, виділення спільних прецедентів є важливою частиною аналізу функціоналу системи. Загалом, подрібнення прецедентів із вказуванням відношень включення між ними означає, що опис системи стає більш деталізованим та структурованим.

Можна поставити наступне запитання. Як бути, коли, як здається, потрібно показати, що прецедент А включає прецедент В, але дії прецедента В виконуються у прецеденті А не завжди? Тобто має бути зроблений якійсь опис умови виконання прецеденту В. Для відношення включення прецедентів таке зробити не можна.

Розширення прецедентів

Розширення прецендентів – це відношення прецедентів, яке описує як та за яких умов поведінка прецедента-розширювача може бути впроваджена у (або повністю замінити) поведінку прецедента, який розширюється. Відношення позначається словом «extend» та стрілкою, спрямованою від прецедента-розширювача на прецедент, який розширюється. Так, наприклад, нехай є прецедент **Реєстрація**. Можна його розширити, наприклад, прецедентом **Отримати допомогу при реєстрації**. Скорочений формат позначення такого розширення може мати вигляд

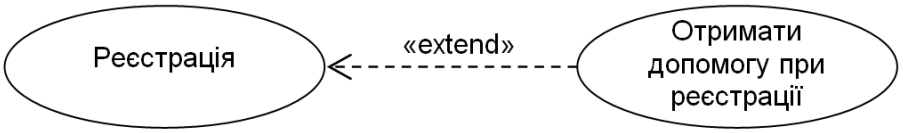


Рис. 6.7. Приклад розширення прецедентів

Розширення може бути позначено докладніше – із вказуванням точки розширення та відповідної умови

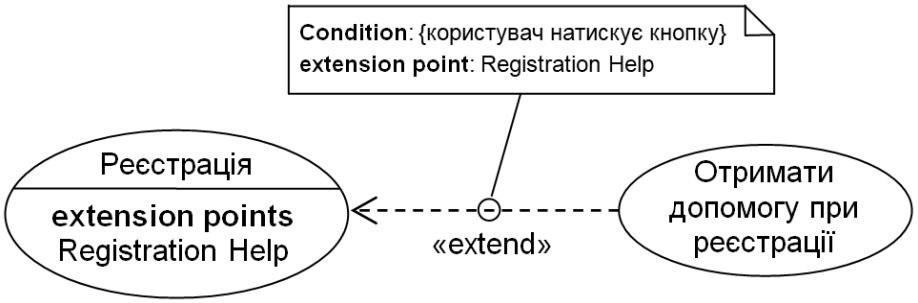


Рис. 6.8. Приклад розширення прецедентів із вказуванням умови.
Використане джерело: https://www.uml-diagrams.org/use-case-extend.html

Нижче наведений приклад діаграми прецедентів, у якій використані розширення та інші відношення прецедентів.

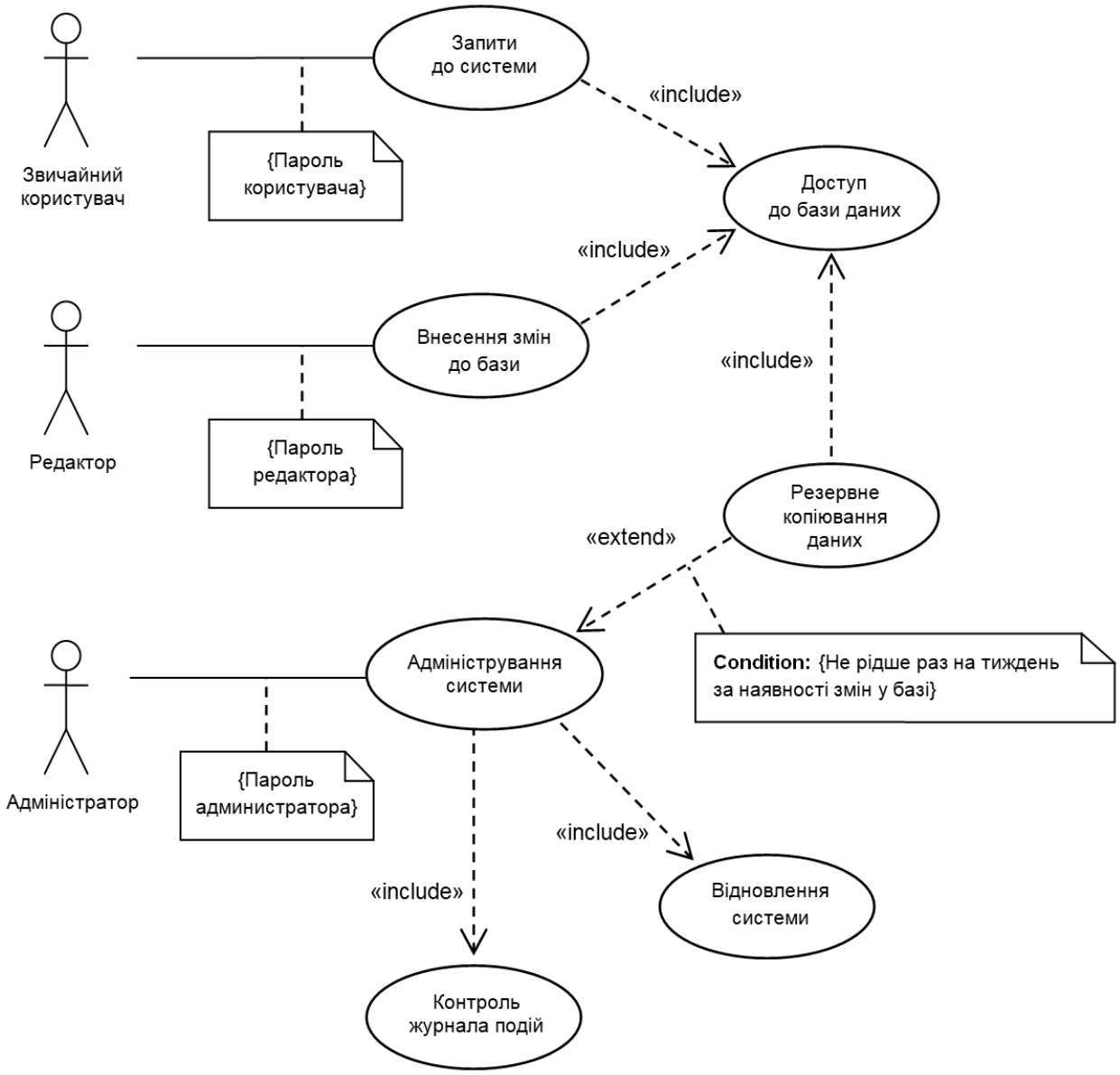


Рис. 6.9. Приклад діаграми прецедентів

Узагальнення прецедентів

Узагальнення прецедентів дещо подібне узагальненню класів, хоча тут мова йде не про узагальнення властивостей об’єктів, а про узагальнення поведінки, якихось дій. Дочірній прецедент конкретизує властивості та поведінку батьківського, узагальненого прецеденту. Узагальнення показано лінією з трикутною стрілкою, спрямовану до узагальненого прецеденту.

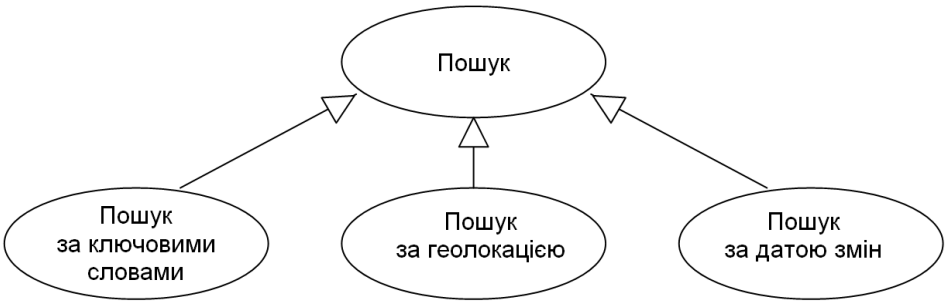


Рис. 6.10. Приклад узагальнення прецедентів

Підсумки щодо діаграм прецедентів

У підсумку, можна вказати призначення діаграм прецедентів:

- Визначення межі системи та її зв'язків із користувачами та іншими системами
- Формування загальних вимог щодо функціоналу системи. Прецеденти описують склад функцій – тобто, ЩО виконується, але не описують ЯК виконується та у якій послідовності
- Розробка моделі системи задля її наступної деталізації
- Основа для розробки тестів
- Підготовка документів для взаємодії з замовниками та користувачами системи. Діаграма прецедентів описує поведінку системи з точки зору її користувача

Діаграми прецедентів зазвичай створюються на первісних етапах розробки системи.

Діаграми класів

Такі діаграми показують структуру системи, підсистеми або компонента як пов'язані класи та інтерфейси з їх атрибутами, операціями та зв'язками. Клас на діаграмі показується прямокутником, розділеним на три частини

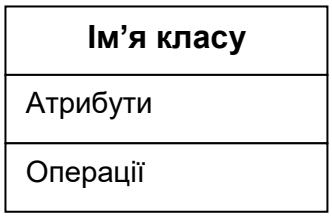


Рис. 6.11. Формат позначення класу

Ім’я класу – одне, максимум два слова, записується напівжирним шрифтом з вирівнюванням від центру. Ім’я абстрактного классу записується курсивом, а перед іменем інтерфейсу – ключове слово «interface».

Атрибути – показують властивості об’єктів – це поля (або дані-члени класів). Записуються з вирівнювання з лівого краю. Назва атрибуту починається з маленької букви. Спочатку йде назва, потім через двокрапку тип.

Операції (методи, або функції-члени класів) – позначають поведінку. Синтаксис операції подібний атрибутам.

Для атрибутів та операцій вказуються також обмеження доступу

\+ (public)

– (private)

\# (protected)

~ (package)

Рис. 6.12. Приклади позначення класу та інтерфейсу

Основні види відношень між классами та об'єктами

На діаграмах UML відображаються, зокрема, таки відношення

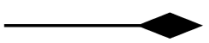
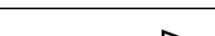
Відношення	Позначення	Примітка
Залежність		
Асоціація		
Агрегація		*"has - a"*
Композиція		*"part - of"*
Узагальнення (успадкування)		*"is - a"*
Реалізація		

Рис. 6.13. Основні відношення класів, об'єктів

Залежність – це таке відношення, коли зміна специфікації класу B призводить до зміни роботи залежного від нього класу A, но не навпаки. Стрілка спрямовується від класу-клієнта (A) до класу-постачальника (B).

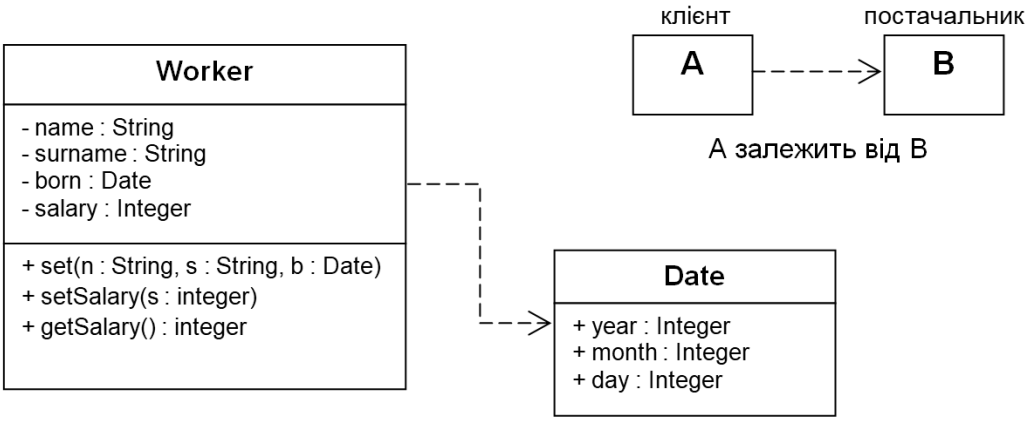


Рис. 6.14. Залежність

Асоціація – це просто якийсь зв'язок між об'єктами. Асоціація може мати ім'я, яке вказує тип відношень, та стрілку напрямку

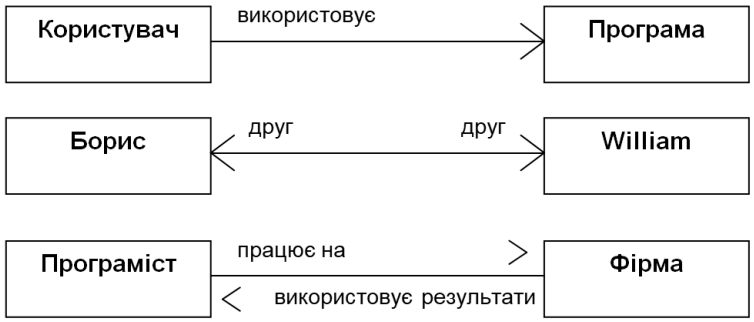


Рис. 6.15. Асоціації

Може вказуватися кратність асоціаці. Кратність (мультиплікатор) означає число зв'язків між кожним об'єктом на початку лінії із об'єктом кінця лінії.

У відношеннях між класами асоціація може мати деякі властивості, описані навіть деяким окремим класом

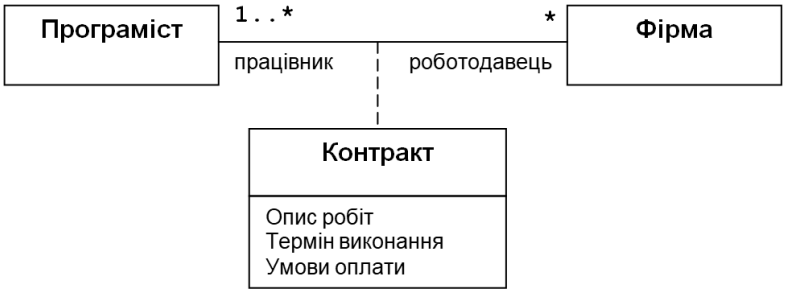


Рис. 6.16. Приклад опису асоціації

Тлумачити наведену вище діаграму можно наступним чином: фірмі потрібен хоча б один програміст (1..*), а програміст може взагалі не працювати, працювати в одній або декількох фірмах (*). Умови роботи програміста на фірмі описуються контрактом.

Ще один випадок складної асоціації – асоціація з вибором альтернативи (*xor – association англ.*)

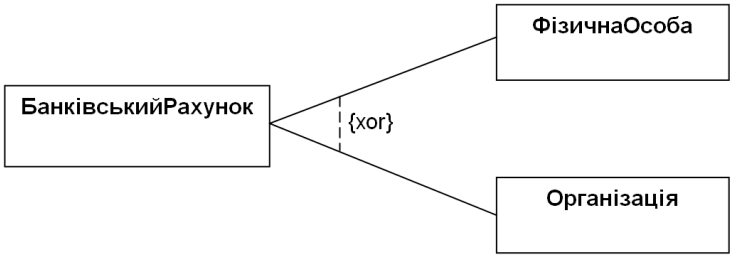


Рис. 6.17. Приклад xor-асоціації

Така xor-асоціація означатиме, що банківський рахунок може бути або для фізичної особи, або для організації.

Узагальнення – це відношення *"є – чимось" (* ***"is – a"*** *англ.*) означає, що деякий тип є різновидом якогось узагальненого типу. Іншими словами, деякий похідний клас успадковує властивості якогось базового класу.

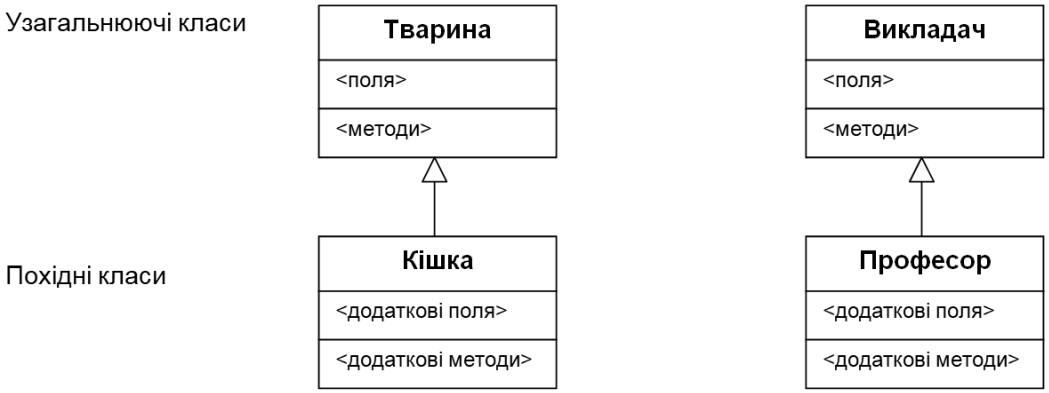


Рис. 6.18. Приклади узагальнення

Таким чином: "Кішка є твариною", "Професор є викладачем".

Реалізація – це спеціалізований абстрактний зв’язок між двома наборами елементів моделі, один з яких представляє специфікацію (постачальник), а інший — реалізацію останнього (клієнт).

Різновидом реалізації, який досить часто використовується в ООП, є **реалізація інтерфейсу**. Реалізація інтерфейсу – це спеціалізований зв’язок

реалізації між класифікатором та інтерфейсом. Цей зв’язок означає, що реалізуючий класифікатор відповідає контракту, визначеному інтерфейсом.

Відношення реалізації інтерфейсу між класифікатором та інтерфейсом означає, що класифікатор підтримує набір функцій, якими володіє інтерфейс, і будь-який з його батьківських інтерфейсів.

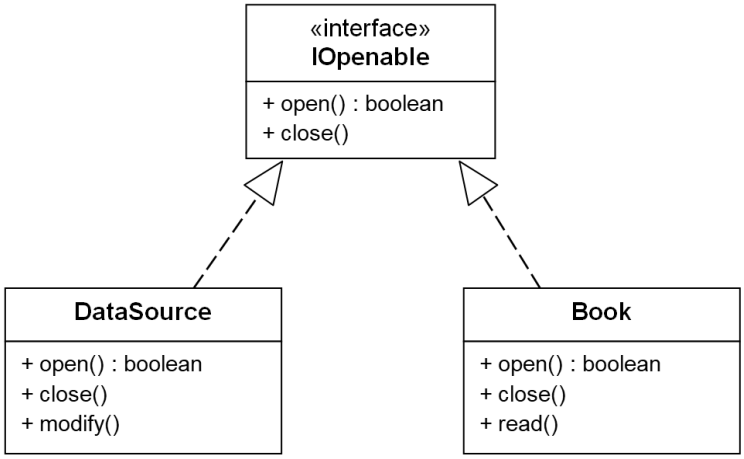


Рис. 6.19. Реалізація інтерфейсу

Агрегація – це відношення *"має – щось" (* "***has – a***" *англ.*) означає, що деяке ціле може складатися з окремих частин, причому кожна з цих частин має право на окреме життя. На діаграмі класів агрегація позначається лінією з ромбом. Ромб вказує на ціле (контейнер)

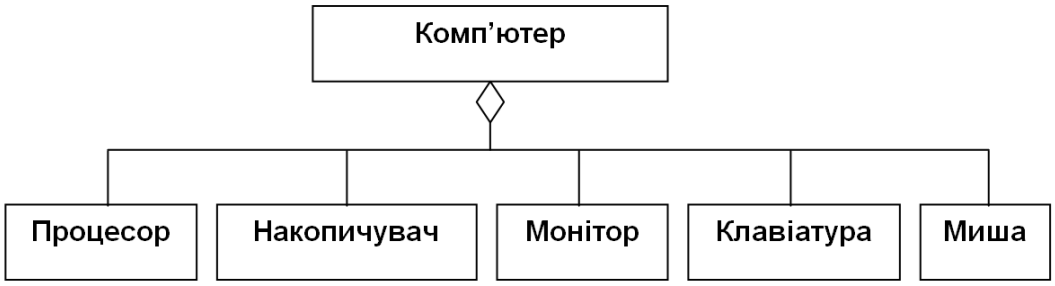


Рис. 6.20. Агрегація

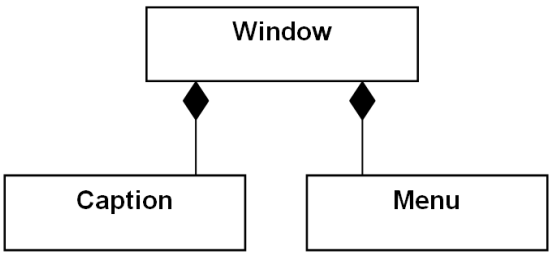
Кожна окрема частина, яка агрегується, може продовжувати своє існування, наприклад, мишу можна відімкнути від комп’ютера і з'єднати зі смартфоном. Час життя частини може бути більше часу життя контейнера.

Композиція – це різновид агрегації з таким обмеженням: якщо складене ціле (контейнер) видаляється, усі його складові частини видаляються разом із ним. Композиція – це відношення *"**part – of**"*. Відмінність композиції від

агрегації полягає у співвідношенні життєвого циклу об’єкту-цілого та об’єктів-частин.

Композиція позначається чорним ромбом зі сторони контейнера.

В ООП доволі частим є випадок, коли клас А у якості поля містить у собі об’єкт іншого класу В. Якщо знищується об’єкт-контейнер (типу А) , то це автоматично знищує також об’єкт типу В. Так, наприклад, об’єкт Вікно може містити у собі об’єкти Заголовок та Меню




```
class Caption
{
. . .
};

class Menu
{
. . .
};

class Window
{
private:
   Caption cob;
   Menu mob;
public:
   . . .
};
```

або

```
class Caption
{
. . .
};

class Menu
{
. . .
};

class Window
{
private:
   Caption* pc;
   Menu* pm;
public:
   Window() {pc = new Caption; pm = new Menu; }
  ~Window() {delete pc; delete pm; }
   . . .
};
```

Рис. 6. 21. Приклад композиції на С++

Композиція vs Успадкування

Розглянемо наступний приклад. Нехай клієнту потрібні два класи: Base з методом Method1 та клас Derived з методами Method1, Method2. Зробимо дві спроби запрограмувати клас Derived.

Перший раз Derived буде похідним від Base – код праворуч.

Другий раз клас Derived зробимо не спадкуванням від Base, а композицією. Але для метода Method1 у класі Derived використаємо об'єкт класу Base, який живе тільки у класі Derived і від нього виликаємо Method1 – код ліворуч.

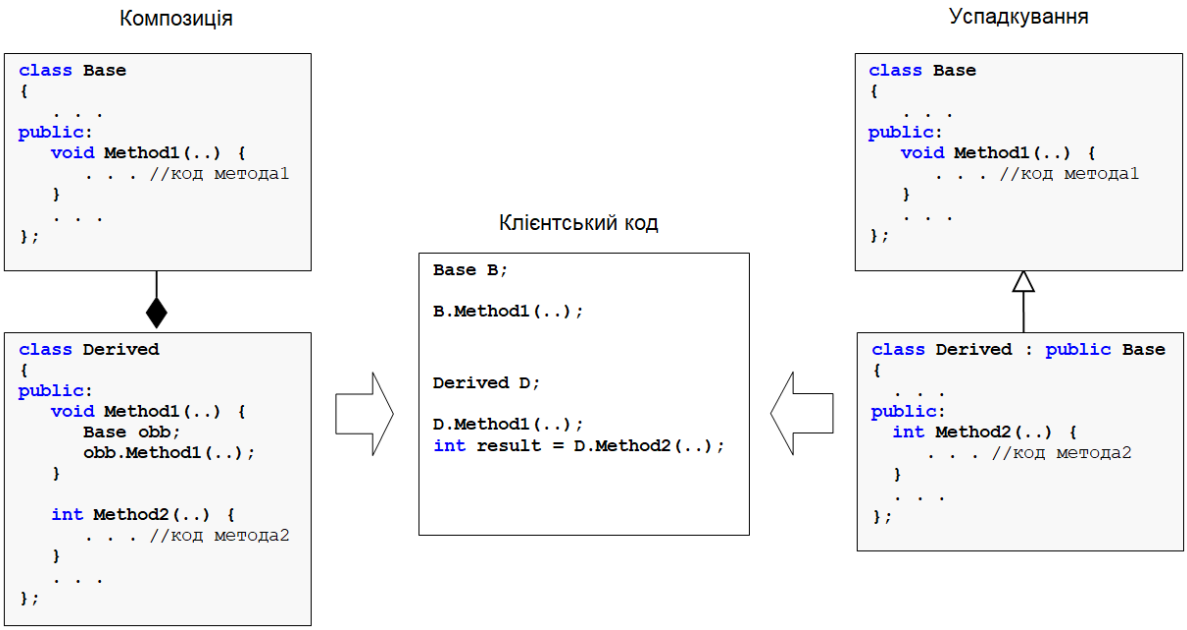


Рис. 6.22. Композиція vs Успадкування

Клієнт цих класів створює екземпляри об'єктів Base і Derived і викликає їхні методи – Method1, Method2. Як здається, клієнт не відчуває зовні відмінностей у роботі об'єкта класу Derived – чи то у випадку реалізації цього класу спадкуванням, чи то композицією.

Таким чином, можна стверджувати, що у деяких випадках одне й те саме можна реалізувати композицію або успадкуванням. Зокрема, можливо замінювати композицією успадкування, у тому числі множинне.

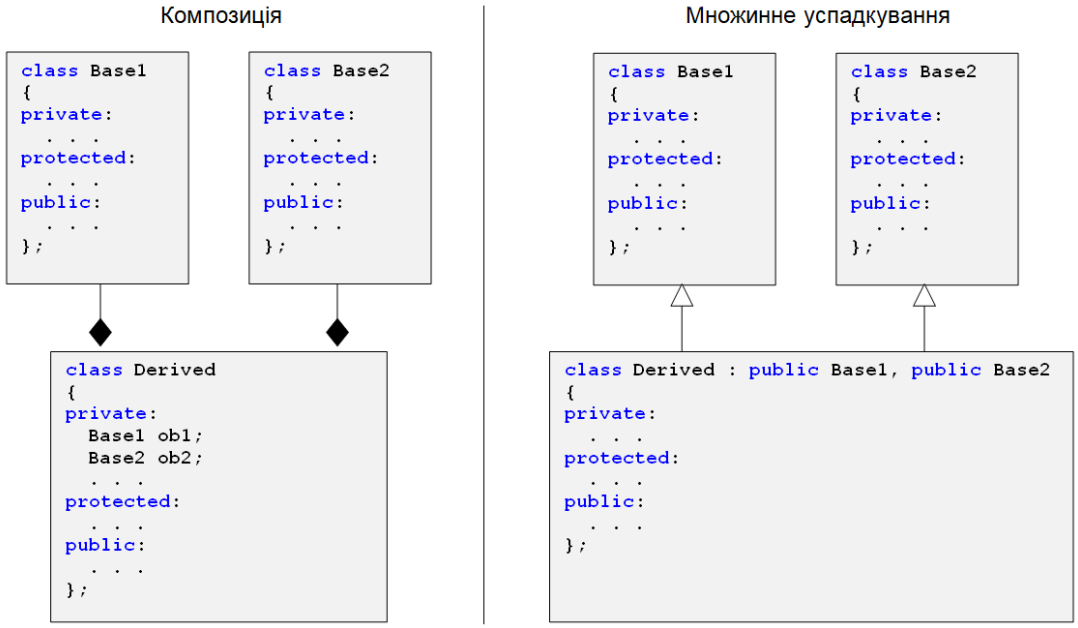


Рис. 6.23. Два варіанти реалізації класу Derived

Принципова можливість заміни множиного спадкування може виправдати відмову від нього у деяких мовах програмування.

Створення діаграм класів у проекті C++ засобами Visual Studio

Для цього у вікні Solution Explorer виберіть рядок з іменем проекту і далі, клацнувши правою кнопкою «миші» увійдіть у вікно додавання нового елемента у проект.

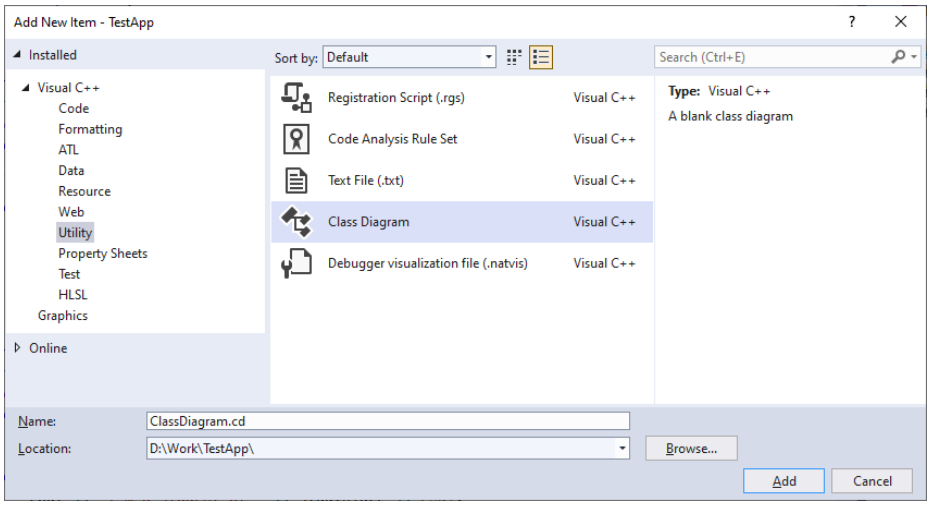


Рис. 6.24. Додавання діаграми класів у проект

Після вибору імені файла діаграми класів і натискування кнопки “Add” буде відображене порожнє вікно. Треба додати у вікно діаграми потрібні класи. Можна автоматично додати усі класи проєкта – у вікні Solution Explorer вибрати рядок з іменем проекту і далі вибрати View->View Class Diagram.

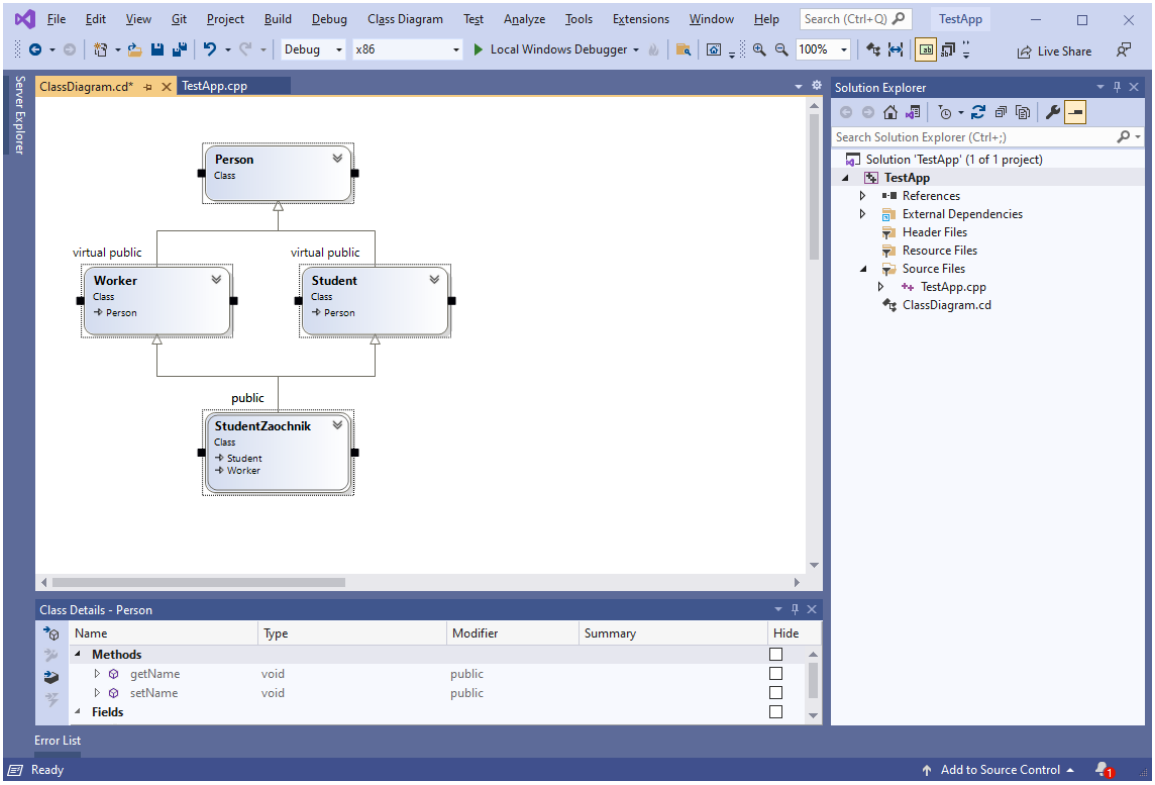


Рис. 6.25. Вигляд вікна Visual Studio для діаграми класів

Діаграму класів можна відредагувати. Розкриємо показ усіх членів класів – це може призвести до потреби скорегувати форму ліній зв’язків.

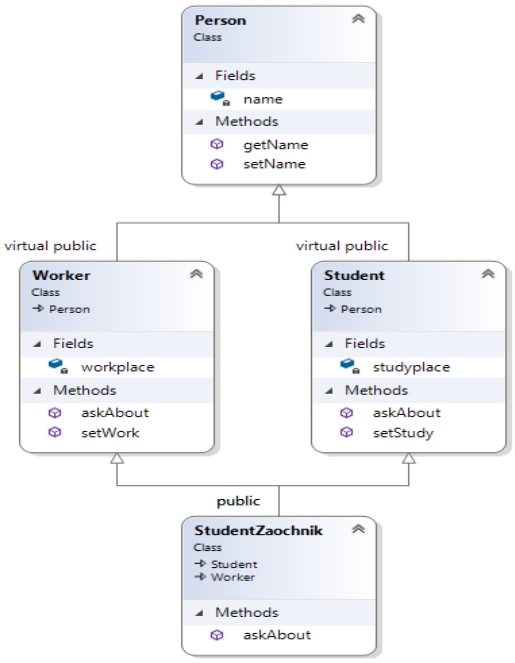


Рис. 6.26. Вигляд діаграми класів із деталізацією показу усіх членів класів

Для аналізу частини класів треба вибрати їх з відповідних модулів

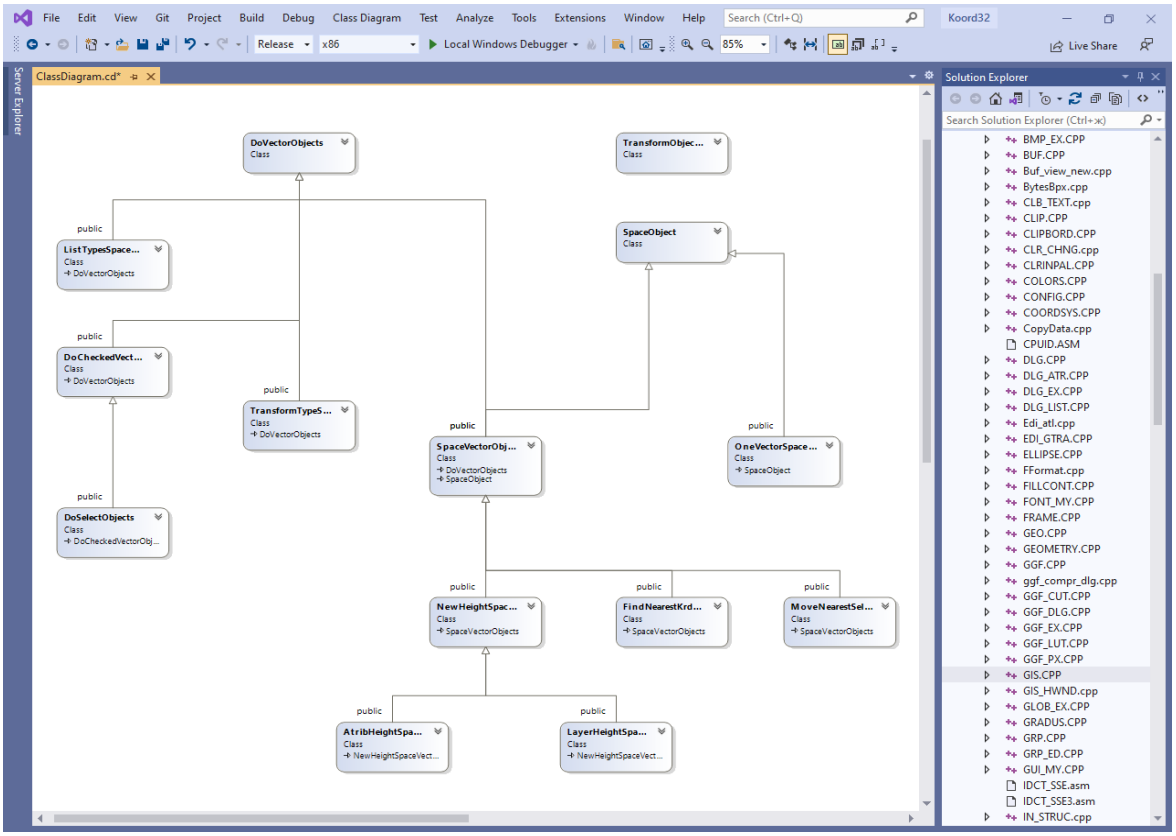


Рис. 6.27. Показ частини класів при огляді складного проекту

Контрольні запитання

1. Що таке UML?
2. Що означає "моделювання" засобами мови UML?
3. Які є основні різновиди представлення систем у вигляді UML-моделей?
4. Які є різновиди діаграм UML?
5. Якими сутностями оперують діаграми прецедентів?
6. Що визначається поняттям "актор" на діаграмі прецедентів?
7. Що таке прецедент?
8. Які відношення можуть бути між прецедентами?
9. Чим відрізняється розширення прецедентів від включення?
10. Як позначається клас на діаграмах класів?
11. У чому відмінності позначення класів-інтерфейсів від інших класів?
12. Як позначаються атрибути об'єктів?
13. Куди спрямовується стрілка на позначенні залежностей?
14. Чим композиція відрізняється від агрегації?
15. Що таке xor-асоціація?
16. Як позначити спадкування класів на діаграмі UML?
17. Як спадкування замінити композицією?
18. Як створити діаграму класів у проєкті C++ в середовищі Visual Studio?

Лекція 7

Тема 3.2. Вкладені та локальні класи

Вкладені класи

У тілі оголошення одного класу можна оголосити інший клас. Наприклад, у класі Outer оголошуємо вкладений клас з ім’ям Nested. Вкладений клас буде членом класу Outer. Для доступу до членів класів потрібно створювати відповідні об'єкти

```
#include <iostream>
#include <string>
using namespace std;

class Outer    //охоплюючий клас
{
public:
    class Nested     //вкладений клас
    {
    public:
        void methodnested(string s)
        {
            cout << "Calling the methodnested" << s << endl;
        }
    };

    void method(void)
    {
        Nested n;
        n.methodnested(" from a method");
    }
};
```

```
//--test--
int main()
{
    Outer o;
    o.method();

    Outer::Nested n;              //об'єкт вкладеного класу
    n.methodnested(" directly");

    return _getwch();
}
```

Тестова програма виведе на консоль два рядка

```
Calling the methodnested from a method
Calling the methodnested directly
```

Якщо потрібно заборонити зовні мати доступ до вкладеного класу та його членів, то клас Nested треба оголосити як **private**. Тоді об'єкт класу Nested можна буде створити тільки всередині класу Outer.

```
class Outer
{
private:
    class Nested    //вкладений клас
    {
    public:
        void methodnested(string s)
        {
            cout << "Calling the methodnested" << s << endl;
        }
    };
public:
    void method(void)
    {
        Nested n;
        n.methodnested(" from a method");
    }
};
```

```
//--test--
int main()
{
    Outer o;
    o.method();

    return _getwch();
}
```

Тестова програма виведе на консоль рядок

```
Calling the methodnested from a method
```

Можна записати визначення функцій за межами оголошення класів

```
//--оголошення класів--
class Outer   //охоплюючий клас
{
private:
    class Nested    //вкладений клас
    {
    public:
        void methodnested(string s);
    };
public:
    void method(void);
};
```

```
//--визначення функцій класів--
void Outer::method(void)
{
    Nested n;
    n.methodnested(" from a method");
}

void Outer::Nested::methodnested(string s)
{
    cout << "Calling the methodnested" << s << endl;
}
```

Права доступу у вкладених класах

У деяких інформаційних джерелах по C++ зазначається, що “вкладення класу в інший клас не дає спеціальних привілеїв доступу до функцій-членів вкладеного класу. Так само функції-члени охоплюючого класу не мають спеціального доступу до членів вкладеного класу” [16]. Таким чином, може скластися враження, що вкладені класи немає сенсу робити – буцімто, вони поводять себе як зовсім окремі класи, хіба що один якось використовує інший. Це не зовсім так. Зробимо невеличке дослідження. Запишемо в охоплюючосу класі перемінну x, і будемо намагатися доступитися до неї у вкладеному класі.

```
class Outer   //охоплюючий клас
{
public:
    int x = 0;

    class Nested    //вкладений клас
    {
    public:
        void methodnested(void)
        {
            x = 15;    //помилка, доступ до члена охопллюючого класа
        }
    };

    void method(void)
    {
        cout << "x = " << x << endl;
        Nested n;
        n.methodnested();
        cout << "x = " << x << endl;
    }
};
```

Компілятор видає помилку. Далі перепишемо метод вкладеного класу наступним чином: будемо передавати йому вказівник на охоплюючий клас. Тоді метод вкладеного класу може працювати з членом x охоплюючого классу навіть у випадку, якщо x буде оголошене як private.

```
class Outer   //охоплюючий клас
{
private:
    int x = 0;
    class Nested    //вкладений клас
    {
    public:
        void methodnested(Outer *po)
        {
            po->x = 15;    //а так - працює
        }
    };
```

```
public:
    void method(void)
    {
        cout << "x = " << x << endl;
        Nested n;
        n.methodnested(this);
        cout << "x = " << x << endl;
    }
};
```

```
//--test--
int main()
{
    Outer o;
    o.method();

    return _getwch();
}
```

Тестова програма виводить на консоль два рядка

```
x = 0
x = 15
```

А тепер перевіримо, чи можливо взагалі по вказівнику на клас отримати доступ до private члена іншого окремого класу? Зробимо клас Nested окремим класом і спростимо код метода Outer, щоб не робити попереднє оглошення класів. Перевіримо наступний код. Як і очікувалося – компілятор видає помилку щодо доступу зовні до private члена x.

```
class Outer
{
private:
    int x = 0;
public:
    void method(void)
    {
        x = 23;
    }
};

class Nested    //окремий клас
{
public:
    void methodnested(Outer* po)
    {
        po->x = 15;    //помилка - спроба доступу до private члена
    }
};
```

Примітка. А якщо x зробити public, то усе компілюється успішно.

Висновок. У вкладеного класу є привілеї доступу до членів охоплюючого класу по вказівнику на нього.

Коли має сенс робити вкладені класи? Уявимо, що існують два окремі класа. Один з них (класс **Outer**) якось використовує інший (**Someclass**)

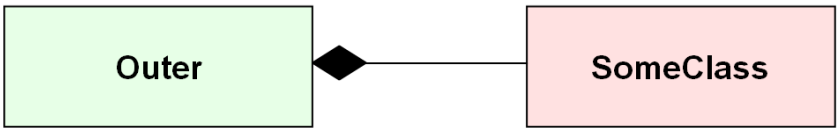


Рис. 7.1. Пара окремих класів

Якщо клас **Someclass** нікому, окрім класу **Outer**, не потрібен, то доцільно **Someclass** інкапсулювати у клас **Outer** у вигляді вкладеного класу

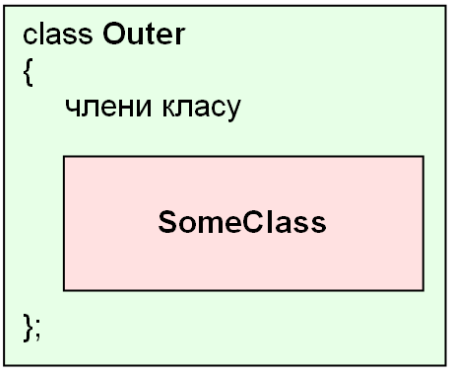


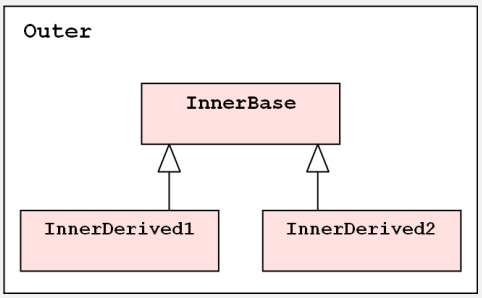
Рис. 7.2. Клас **Someclass** став вкладеним

Одним з призначень вкладеного класу буде те, що він може ховати деякі подробиці реалізації чогось в охоплюючому класу. Так, наприклад, у стандартному класі вектора ітератор виконано як вкладений клас – а сам по собі ітератор нікому не потрібен без вектора

```
namespace std
{
    template <class T>
    class vector
    {
        class iterator
        {
            ...
        };
        ...
    };
    ...
};

std::vector<int>::iterator it;    //?
```

Вкладені класи можуть активно використовувати відношення спадкування. Зокрема, вкладений клас може успадкувати (реалізувати) якийсь зовнішній базовий клас (або інтерфейс).

У деякому класі може бути побудована ціла ієрархія вкладених класів починаючи з вкладеного базового класу, наприклад




```
class Outer
{
private:
   class InnerBase
   {
   protected:
      virtual void func() = 0;
   public:
      void method() {
            cout << "Calling a method then ";
            func();
            cout << " func" << endl;
      }
   };

   class InnerDerived1 : public InnerBase
   {
   protected:
      void func() {
            cout << "InnerDerived1";
      }
   };

   class InnerDerived2 : public InnerBase
   {
   protected:
      void func() {
            cout << "InnerDerived2";
      }
   };

public:
   void getResult() {
      InnerBase* p = new InnerDerived1;
      p->method();
      delete p;
      p = new InnerDerived2;
      p->method();
      delete p;
   }
};
```

```
//--test--
int main()
{
   Outer o;
   o.getResult();
   return _getwch();
}
```

Результат:

```
Calling a method then InnerDerived1 func
Calling a method then InnerDerived2 func
```

Позначення вкладених класів на діаграмах UML

Зараз вже застаріла специфікація UML 1.4.2 визначала вкладений клас як клас, оголошений в іншому класі та належний до простору імен класу, що оголошує. Відносини між цими класами називалися «асоціація елементів, що належать простору імен».

Відповідно до UML 1.4.2 клас оголошення (вкладеність) і вкладений клас можуть відображатися з'єднаними лінією з піктограмою «якоря» на кінці, з'єднаною з класом оголошення. Піктограма прив'язки – це хрест всередині кола.

Специфікації UML 2.x, описують вкладеність класифікаторів у структурованих класах без надання явної нотації для вкладеності, проте використання нотації попередніх версій не заперечується.

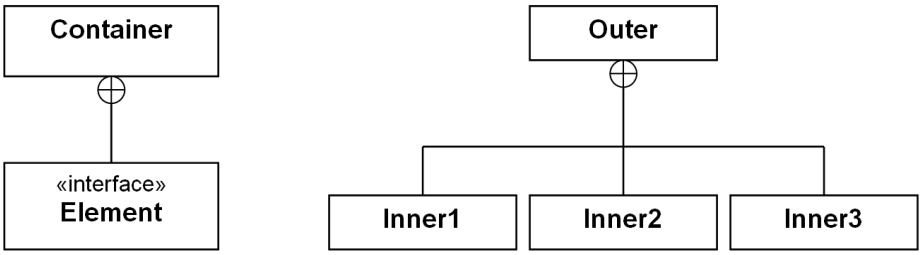


Рис. 7.3. Позначення вкладених класів на діаграмах UML

Локальні класи

Загалом усе, що оголошується **всередині функції**, є локальним. Локальними можуть бути прості перемінні стандартних типів, перемінні типів користувача, екземпляри об'єктів класів, наприклад

```
class SomeClass
{
    //... члени класу
};

void MyFunc(int arg) {

    int x, y;           //локальні прості перемінні
    float arr[100];     //локальний масив
    SomeClass obj;      //локальний об'єкт класу

    //...  використання локальних об'єктів
}
```

Зверніть увагу, що об'єкт "obj" є локальним, але його клас (SomeClass) оголошено зовні функції, тому цей клас не є локальним.

Уявимо собі ситуацію, що клас SomeClass нікому не потрібен поза межами функції MyFunc. Тоді може бути доцільним оголошення такого класу разом з його визначенням розташувати у тілі відповідної функції. Такий клас стане **локальним класом**.

Приклад локального класу всередині окремої (глобальної) функції

```
void MyFunc(int arg)
{
    class SomeClass     //локальний клас
    {
    public:
        void method(int arg) {
          std::cout << "Method of local class: arg = " << arg << "\n";
        }
    };
    SomeClass obj;      //локальний об'єкт
    obj.method(arg);
}
```

```
int main()
{
    MyFunc(13);
    return _getwch();
}
```

Тестова програма виведе на консоль рядок

```
Method of local class: arg = 13
```

А тепер можна поставити таке питання. Чи можуть бути локальні класи усередині функції-члена (метода) деякого класу? Розглянемо приклад

```
class MyClass
{
public:
    void method(int arg) {
        class Inner    //оголошення локального класу
        {
        public:
            int func(int arg) {
                std::cout << "Inner.func(" << arg << ")\n";
                return arg + 10;
            }
        };

        Inner o;       //використання локального класу
        int x = o.func(arg);
        std::cout << "Result: x = " << x << "\n";
    }
};
```

```
int main()
{
    MyClass obj;
    obj.method(3);

    return _getwch();
}
```

Тестова програма виводить у консолі два рядка

```
Inner.func(3)
Result: x = 13
```

Таким чином, локальний клас може бути оголошений як у окремій глобальній функції, так і у функції-члені (методі) іншого класу. Мають місце наступні обмеження для локальних класів:

- Область видимості локального класа обмежена тілом функції, де цей клас оголошений, тому неможливо звернутися до члена локального класу зовні цієї функції
- Функції-члени (методи) локального класу неможливо визначити поза тою функцією, де локальний клас оголошений
- Локальний клас не може мати статичних даних-членів

Може виникнути ще таке запитання: а чи можуть бути локальні функції? В мові C++ локальних функцій взагалі не може бути – ані всередині функцій-чденів класів, ані всередині окремих (глобальних) функцій. А локальні класи в C++ підтримуються – хоча клас може інкапсулювати не одну, а багато функцій-членів. Тому, якщо є потреба створювати локальні функції, то можна їх робити у вигляді функцій-членів локальних класів.

Зауваження. Деякі компілятори мови C, наприклад, gcc у якості розширення підтримують локальні функції. Крім того, локальні функції підтримуються деякими іншими мовами програмування, наприклад, C# [15].

Контрольні запитання

1. Що таке вкладений клас? Яка доцільність його використання?
2. Які права доступу у вкладених класів?
3. Як позначаються локальні класи на діаграмах класів?
4. Що таке локальний клас і які для нього обмеження?
5. Чи можуть бути локальні функції в C++?

Лекція 8

Тема 3.3. Операції з об'єктами

У цій темі увагу зосереджено на виконанні якихось дій з обєктом(ами), як єдиним цілим.

Ініціалізація одного об’єкта іншим об’єктом

Така операція являє собою створення копії вже існуючого об'єкта. Потрібно, щоб у новоствореного об’єкта В дані-члени отримали б значення від однойменних членів об’єкта А. Зокрема, якщо об'єкт містить в собі деякий масив, то усі елементи масиву об’єкта В мають отримати значення відповідних елементів масиву об’єкта А.

У якості прикладу розглянемо наступний клас.

```
class myclass
{
private:
   int xo,yo;
   int *p;
public:
   myclass(int,int);
  ~myclass();
   void show(void);
};

myclass::myclass(int x,int y)
{
cout << "Constructor\n";
xo = x;
yo = y;
p = new int[4];
for (int i=0; i<4; i++)
    p[i] = xo+yo+i;
}

myclass::~myclass()
{
cout << "Destructor\n";
if (p) delete []p;
p = 0;
}

void myclass::show(void)
{
cout << "x = " << xo << '\n';
cout << "y = " << yo << '\n';
cout << "p = " << p << '\n';
cout << "p[] = ";
for (int i=0; i<4; i++)
    cout << p[i] << " ";
cout << "\n\n";
}
```

Зробимо тестову програму

```
//--test--
int main()
{
    cout << "object A\n";
    myclass A(2, 5);
    A.show();

    cout << "object B\n";
    myclass B = A;   //створення копії - ініціалізація
    B.show();

    return _getwch();
}
```

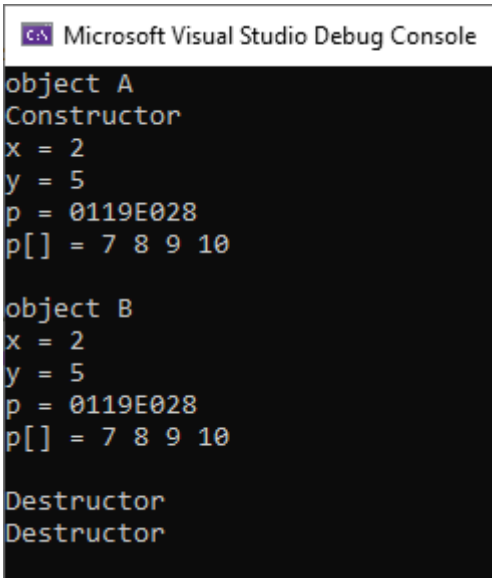


Рис. 8.1. Результат виконання тесту

При виконанні такого мають місце наступні моменти:

- чомусь не викликається конструктор об'єкту B

- після відпрацювання другого деструктора програма аварійно переривається перед самим завершенням.

Пояснення. При створенні екземпляра об'єкту B з початковою ініціалізацією іншими об'єктом (A) замість звичайного конструктора викликається так званий конструктор копії. Оскільки ми не запрограмували такий конструктор, то компілятор створює конструктор копії за замовчуванням, який автоматично копіює значення усіх членів об'єкту A в однойменні члени об’єкту B. При цьому копіюються також і значення вказівників масивів p[ ]. Вказівник p масиву об'єкту B дорівнює вказівнику масиву об'єкту A – тобто обидва вказівника вказують на той самий блок пам'яті. Це означає, що деструктори двох об’єктів намагаються знищувати той самий блок памяті двічі. А це критична помилка виконання програми.

Конструктор копії

Якщо дії конструктора копії за замовчуванням для об’єктів деяких класів є неприйнятними, то програмісту треба у цьому класі явно визначити потрібний конструктор копії. Формат оголошення конструктора копії такий:

```
class someclass
{
. . .
public:
   someclass(const someclass &ob);       //конструктор копії
   . . .
};
```

Параметром конструктора копії є посилання на інший об'єкт – його можна використати для копіювання значень відповідних членів.

З урахуванням цього вдосконалимо попередній приклад, додавши у класс конструктор копії

```
class myclass
{
private:
   int xo,yo;
   int *p;
public:
   myclass(int,int);               //звичайний конструктор
   myclass(const myclass &ob);     //конструктор копії
  ~myclass();
   void show(void);
};

//--визначення конструктора копії--
myclass::myclass(const myclass& ob)
{
    cout << "Copy constructor\n";
    xo = ob.xo;
    yo = ob.yo;
    p = new int[4];
    for (int i = 0; i < 4; i++)
        p[i] = ob.p[i];
}

...    //решта коду без змін
```

При виконанні тої самої тестової програми, текст якої наведено вище у попередньому пп. усе тепер працює як треба.

Конструктор копії забезпечує створення власного блоку памяті для масиву p[ ] створюваного об’єкту. Значення вказівника p вже не копіюється з об'єкту A у об'єкт B – копіюються тільки потрібні значення членів класу.

Деструктори коректно відпрацьовують знищення робочих масивів p[ ] для усіх об'єктів. Програма успішно завершується.

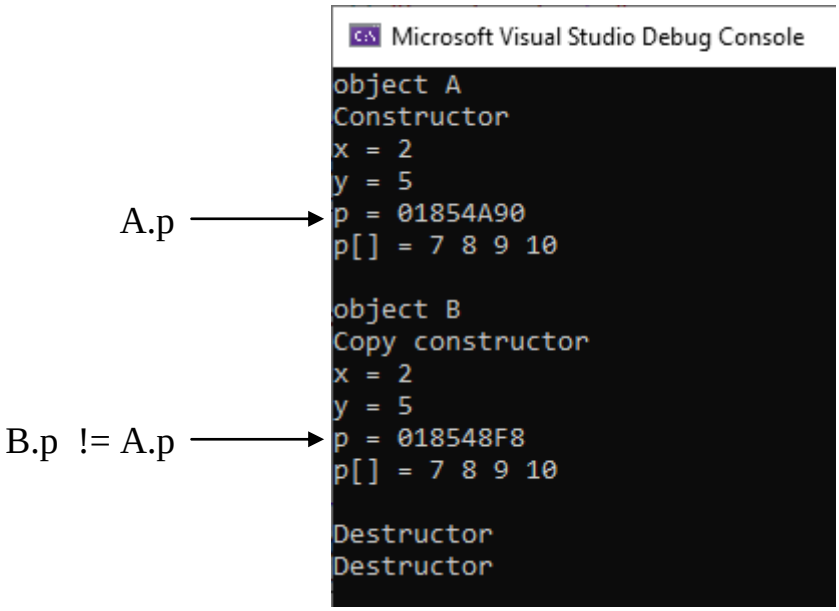


Рис. 8.2. Результати виконання тестової програми

Присвоювання об’єктів

Що означатиме присвоювання у вигляді B=A, якщо A та B є об'єктами? Таке присвоювання можливе тільки для об’єктів однойменного класу.

```
class myclass
{
. . .     //дані-члени
. . .     //функції-члени
};

myclass A,B;  //створення екземплярів об'єктів
. . .         //щось робимо з об'єктами
B = A;   //у дані-члени об’єкту B записуються значення членів A
```

За замовчуванням (якщо це не визначено якось спеціально) компілятор відповідно рядку B=A створює код, який забезпечує побітовий запис даних-членів об’єкта A у відповідні однойменні дані-члени об’єкта B. Для деяких класів така схема присвоювання є прийнятною, а для інших класів – ні.

Розглянемо приклад класу, у якому є динамічний масив.

```
class myclass
{
private:
    int xo=0, yo=0;
    int* p;             //вказівник на динамічний масив
public:
    myclass(void);
```

```
    ~myclass();
    void set(int, int);
    void show(void);
};

myclass::myclass(void)
{
    p = new int[4];    //динамічний масив для 4 елементів
}

myclass::~myclass(void)
{
    if (p) delete[]p;
    p = 0;
}

void myclass::set(int x, int y)
{
    xo = x;
    yo = y;
    for (int i = 0; i < 4; i++)
        p[i] = xo + yo + i;
}

void myclass::show(void)
{
    cout << "x = " << xo << '\n';
    cout << "y = " << yo << '\n';
    cout << "p = " << p << '\n';
    cout << "p[] = ";
    for (int i = 0; i < 4; i++)
        cout << p[i] << " ";
    cout << "\n\n";
}
```

```
//--test--
int main()
{
    myclass A, B;

    A.set(2, 5);
    B.set(0, 0);
    cout << "object A\n";
    A.show();
    cout << "object B\n";
    B.show();

    B = A;   //присвоювання об'єктів
    cout << "after B=A\n";
    cout << "object A\n";
    A.show();
    cout << "object B\n";
    B.show();

    return _getwch();
}
```

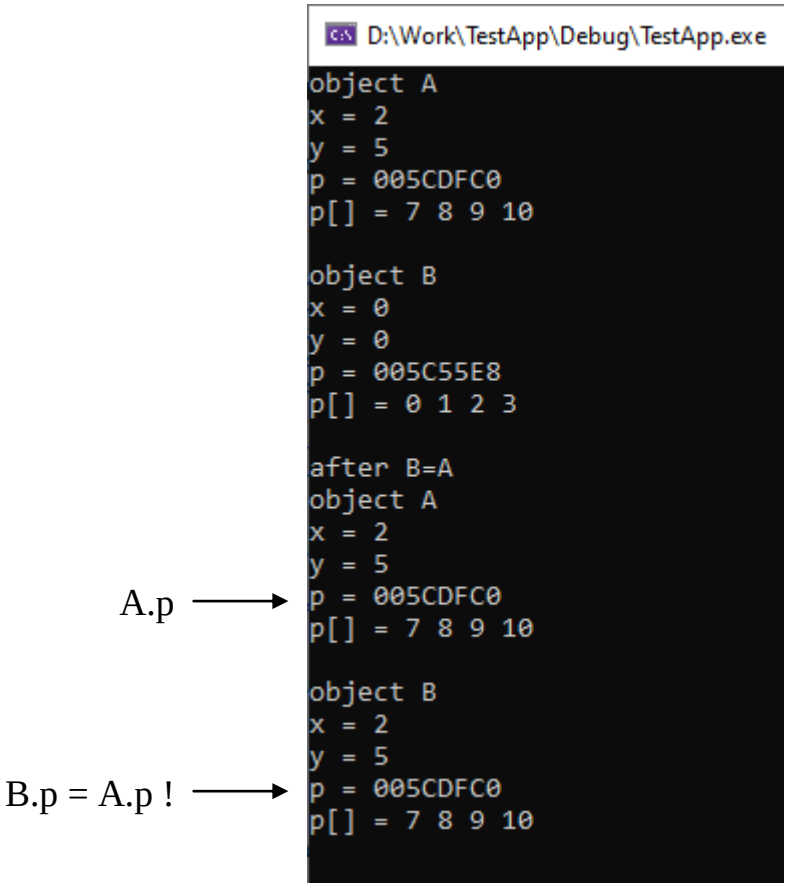


Рис. 8.3. Результати тестової програми

У результаті виконання присвоювання об'єктів (B = A) усі члени об'єкта B приймають значення однойменних членів об'єкта A, зокрема й значення вказівників p. A це призводить до того, що якщо деструктор одного об’єкта знищує блок пам'яті масиву p[ ], то деструктор іншого об'єкта намагається знищити блок пам'яті за тою самою адресою відповідно значення вказівника p. Внаслідок цього програма аварійно завершується.

Схожі проблеми ми вже констатували, коли розглядали створення об’єкта з його ініціалізацію іншим об'єктом. Тоді була виявлена потреба у конструкторі копії. У випадку присвоювання B = A для вже створених об'єктів конструктор копії ні до чого.

Як запрограмувати коректне виконання присвоєння об’єктів? Для цього необхідно перевантажити оператор присвоєння.

Перевантаження операторів в C++

Перевантаження операторів дозволяє визначити потрібні дії для виражень з операндами "класових" типів. У першу чергу це стосується операторів, які маніпулюють об’єктами класів.

У мові C++ дозволено перевантажувати наступні оператори

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

Забороняється перевантажувати оператори

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

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

Незалежно від типу не можна змінювати пріоритети операторів

Для перевантаження операторів створюється спеціальна оператор-функція (***operator function***). Така оператор-функція може бути запрограмована як:

- функція-член класу
- дружня (***friend***) функція;
- глобальний перевантажений оператор – окрема функція з простору імен класу

Оператор як функція-член класу.

Нижче надано формат запису оператора як функції-члена класу. Оголошення класу має наступний формат

```
class Ім'яКласу
{
. . .
public:
   Тип operator#(список параметрів);
   . . .
}
```

Формат визначення оператора:

```
Тип Ім'яКласу:: operator#(список параметрів)
{
. . .    //те, що виконується оператором
}
```

У програмному коді на місці символа '#' записується ідентифікатор оператора.

Перевантаження оператора "="

Нехай потрібно виконати присвоювання одного об'єкта іншому**.** Перевантаження оператора "=" у вигляді функції-члена класу визначається наступним чином

```
Ім'яКласу &Ім'яКласу::operator=(const Ім'яКласу &op2)
{
. . .    //присвоєння поточному об'єкту значень від об'єкта op2
return *this;
}
```

У функції **operator=(..)** фактично є два параметри:

- неявний параметр у вигляді вказівника **this**, який вказує на об'єкт **B**
- явний параметр **&op2**, який вказує на об'єкт **A**

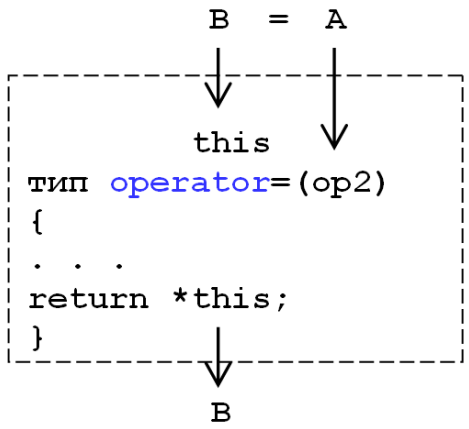


Рис. 8.4. Перевантаження оператора '='

Примітка. Якщо оператор "=" перевантажується за допомогою функції-члена operator=( ), то ця функція-член завжди повинна повертати *this.

Розглянемо приклад перевантаження оператора "=".

Оголошення класу

```
class myclass
{
private:
   int xo,yo;
   int *p;
public:
   myclass(int x,int y);
  ~myclass();
   myclass &operator=(const myclass &ob);      //перевантаження '='
   void show(void);
};
```

Далі текст визначення усіх членів класу

```
myclass::myclass(int x,int y)
{
cout << "Constructor\n";
xo = x;
yo = y;
p = new int[4];
for (int i=0; i<4; i++)
    p[i] = xo+yo+i;
}

myclass::~myclass(void)
{
cout << "Destructor\n";
if (p) delete []p;
p = 0;
}

myclass &myclass::operator=(const myclass &ob)
{
xo = ob.xo;
yo = ob.yo;
for (int i=0; i<4; i++)
    p[i] = ob.p[i];
return *this;
}

void myclass::show(void)
{
cout << "x = " << xo << '\n';
cout << "y = " << yo << '\n';
cout << "p = " << p << '\n';
cout << "p[] = ";
for (int i=0; i<4; i++)
    cout << p[i] << " ";
cout << "\n\n";
}
```

```
//--test--
int main()
{
    cout << "object A\n";
    myclass A(2, 5);
    A.show();

    cout << "object B\n";
    myclass B(0, 0);
    B.show();

    cout << "before B=A\n\n";
    B = A;

    cout << "after B=A\n";
    cout << "object A\n";
    A.show();
```

```
    cout << "object B\n";
    B.show();

    return _getwch();
}
```

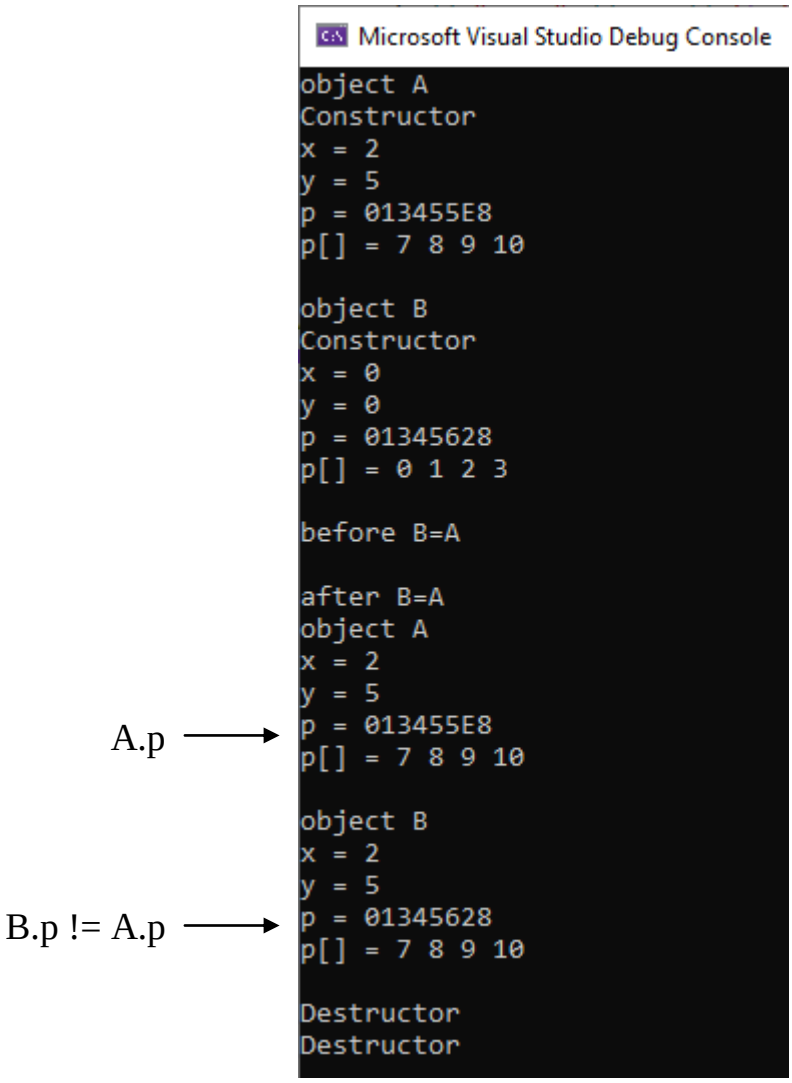


Рис. 8.5. Результати тестової програми

Тест засвідчує, що у результаті присвоювання об'єктів (B = A) усі члени об'єкта B приймають значення однойменних членів об'єкта A, за винятком значення вказівника p. Це дозволяє коректно знищувати робочі масиви об’єктів в ході роботи їхніх деструкторів.

Перевантаження оператора "+"

Як організувати додавання двох об'єктів, наприклад, у форматі

Res = A + B

де A, B та Res є об’єктами деякого класу з ім’ям **SomeClass?**

Для цього у класі можна визначити додатковий член – оператор+

```
SomeClass SomeClass::operator+(const SomeClass& op2)
{
SomeClass tmp;

tmp.x = x + op2.x;
tmp.y = y + op2.y;
tmp.z = z + op2.z;
return tmp;
}
```

Цей оператор буде перевантажувати операцію додавання двох об’єктів A + B. Явний параметр (op2) містить посилання на об’єкт (B) другого операнду додавання, а неявний параметр this буде вказувати на об'єкт (A) першого операнду. У програмному коді оператора використано тимчасовий локальний об'єкт (tmp), а для запису результату такого оператора створюється ще один об'єкт, який при створенні буде ініціалізований локальним об’єктом tmp. Це викликає конструктор копії. Об'єкт-результат оператора ‘+’ потім буде присвоєно об'єкту результату вираження. Тут спрацює оператор присвоювання.

Таким чином, окрім власне оператора ‘+’ у класі потрібно визначити ще конструктор копії та перевантажений оператор ‘=’.

```
class Vertex
{
private:
    int x, y, z;
public:
    Vertex(void);
    Vertex(int, int, int);
    Vertex(const Vertex& ob);
    ~Vertex(void);
    Vertex operator+(const Vertex& op2);
    Vertex& operator=(const Vertex& op2);
    void show(void);
};
```

```
Vertex::Vertex(void)
{
    cout << "Constructor default\n";
    x = y = z = 0;
}

Vertex::Vertex(int xs, int ys, int zs)
{
    cout << "Constructor(x,y,z)\n";
    x = xs; y = ys; z = zs;
}

Vertex::~Vertex(void)
```

```
{
    cout << "Destructor\n";
}

Vertex::Vertex(const Vertex& ob)
{
    cout << "Copy constructor\n";
    x = ob.x;
    y = ob.y;
    z = ob.z;
}

Vertex Vertex::operator+(const Vertex& op2)
{
    Vertex tmp;

    cout << "Operator+\n";
    tmp.x = x + op2.x;
    tmp.y = y + op2.y;
    tmp.z = z + op2.z;
    return tmp;
}

Vertex& Vertex::operator=(const Vertex& op2)
{
    cout << "Operator=\n";
    x = op2.x;
    y = op2.y;
    z = op2.z;
    return *this;
}

void Vertex::show(void)
{
    cout << "(" << x;
    cout << ", " << y;
    cout << ", " << z << ")\n";
}
```

```
//--test--
int main()
{
    Vertex a(3, 4, 7);
    cout << "a ";
    a.show();
    Vertex b(-1, 2, 1);
    cout << "b ";
    b.show();
    Vertex c;
    cout << "c ";
    c.show();
    Vertex d;
    cout << "d ";
    d.show();
    cout << "\n";

    cout << "c = a + b\n";
```

```
    c = a + b;
    cout << "c ";
    c.show();
    cout << "\n";

    cout << "d = a + b + c\n";
    d = a + b + c;
    cout << "d ";
    d.show();
    cout << "\n";

    return _getwch();
}
```

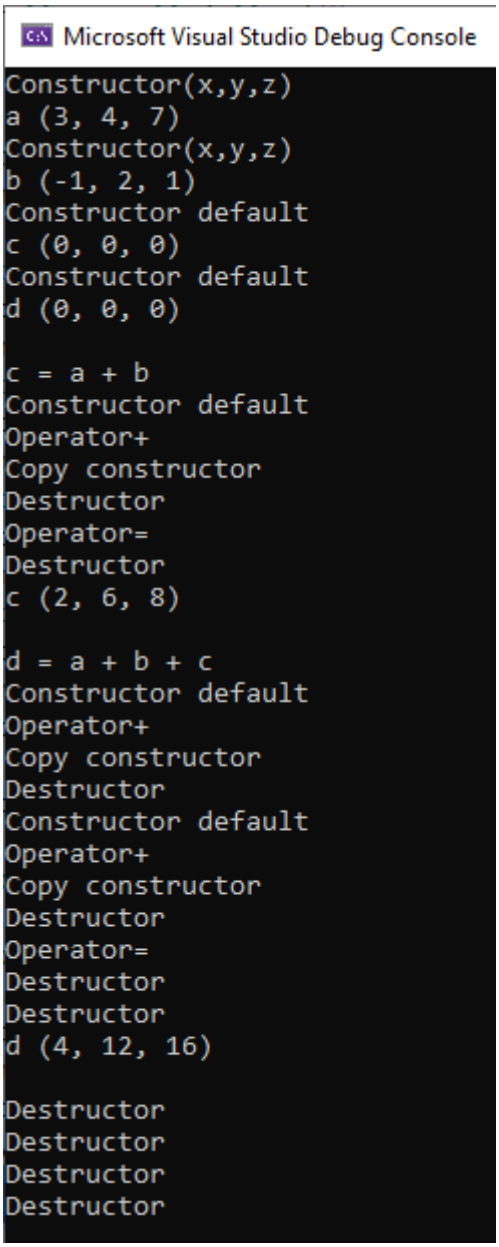


Рис. 8.6. Виконання тесту додавання об'єктів

Розглянемо, як працює перевантажений оператор "+". Функція operator+( ) фактично має два параметри, через які передаються два операнди з ланцюжка синтаксичного аналізу вираження:

- неявний перший параметр (this) – вказівник на поточний об’єкт
- явний параметр – другий операнд додавання (op2)

Те, що функція-оператор повертає об’єкт даного типу, дозволяє виконувати не тільки просту операцію (a + b), а й обчислювати складніші вирази на кшалт (a + b + c).

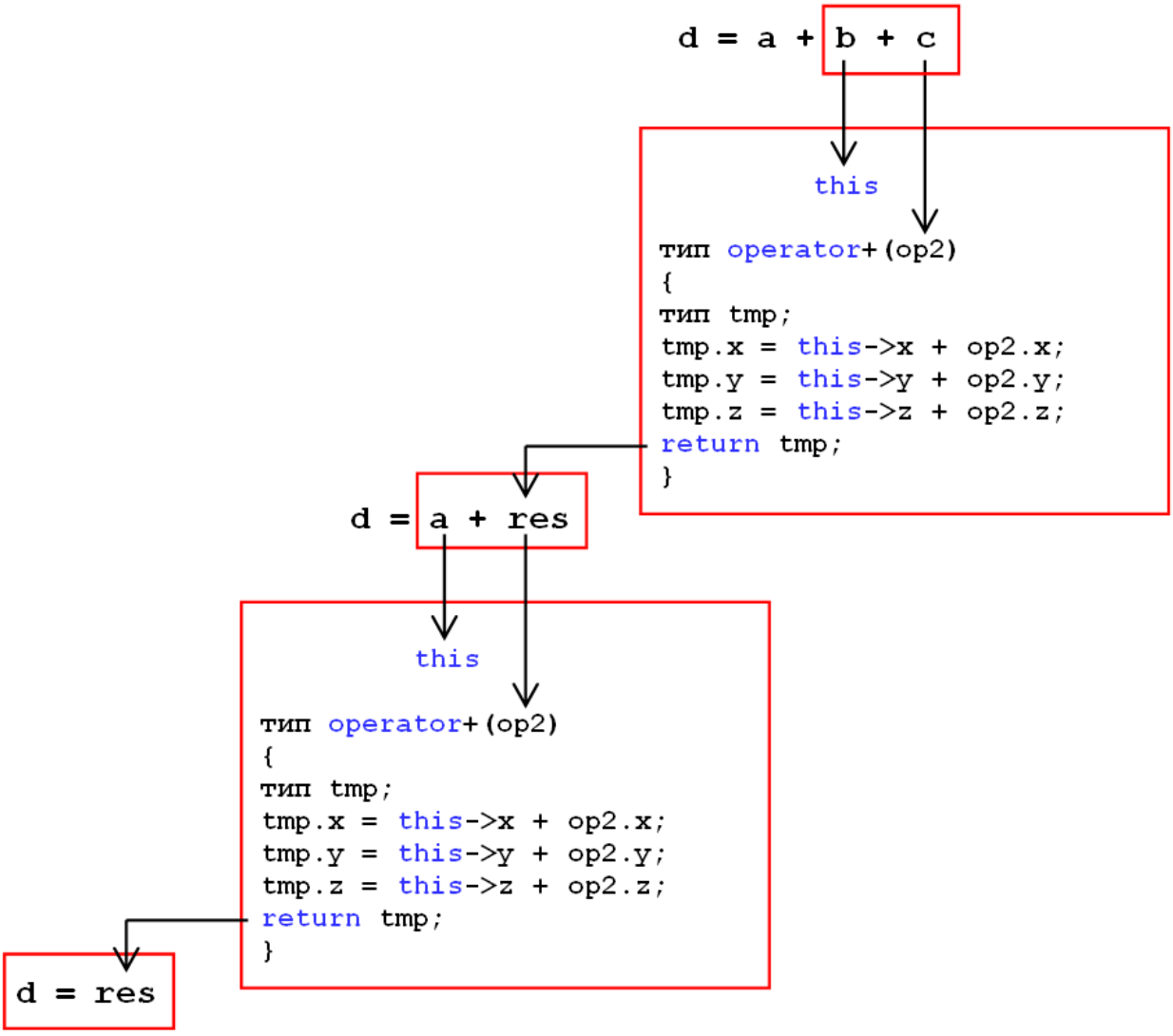


Рис. 8.7. Обчислення суми об'єктів: d = a + b + c

Перевантаження оператора "+" для випадку (об’єкт + число)

Ми вміємо виконувати операцію res = a + b. А як виконувати операцію додавання числа до об’єкту, наприклад, res = a + 10?

Рішення очевидне. Можна запрограмувати версію функції перевантаження оператора "+" з параметром відповідного числового типу, наприклад, int

```
class Vertex
{
. . .
public:
   . . .
   Vertex operator+(const Vertex &op2);  //об'єкт + об'єкт
   Vertex operator+(int op2);            //об'єкт + число
   . . .
};
```

Визначити перевантажений оператор «об'єкт + число» можна наступним чином

```
Vertex Vertex::operator+(int op2)
{
   Vertex tmp;

   cout << "Operator+(object + int)\n";
   tmp.x = x + op2;
   tmp.y = y + op2;
   tmp.z = z + op2;
   return tmp;
}
```

Перевантажені оператори "+" двох видів дозволяють виконувати:

```
c = a + b
c = a + 10
c = a + b + 10
c = a + 10 + b
```

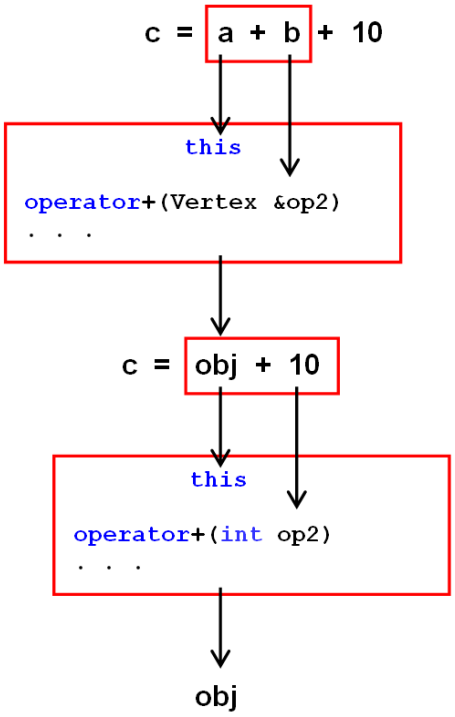


Рис. 8.8. Обчислення вираження a + b + 10

Як здається, дві версії перевантажених операторів ”+” дозволяють обчислювати багато виражень додавання, проте, вони не можуть обчислювати вираження «число + об'єкт».

Перевантаження оператора "+" для випадку (число + об’єкт)

Не можна запрограмувати у класі звичайний перевантажений оператор у вигляді функції-члена operator+() для виконання операції

```
objectC = 10 + objectA
```

оскільки операнд 10 буде вбудованим числовим типом, який не може бути переданий функції-оператору в якості першого (лівого) операнду.

Для вирішення вказаної проблеми у C++ існують два шляхи. Можливо використати

1. Глобальний перевантажений оператор у вигляді окремої (глобальної) функції з простору імен класу
2. Дружню (*friend*) функцію

Глобальний перевантажений оператор "+" має вигляд звичайної глобальної функції, хоча і має ім’я **operator+**

```
Тип operator+(Тип1 op1, Тип2 op2)
{
. . .
return . . .;
}
```

У якості параметрів явно передаються два операнда – лівий та правий. Типи **Тип**, **Тип1** та **Тип2** можуть бути будь-якими, проте існує обмеження: тип хоча б одного формального параметру (**Тип1** або **Тип2**) повинен бути "класовим". Не можна зробити, наприклад, таку оператор-функцію: **operator+(int op1, int op2) –** компілятор видасть помилку. Для класу

```
class Vertex
{
private:
   int x,y,z;
public:
   Vertex(void);
   Vertex(int,int,int);
   Vertex(const Vertex &ob);
   Vertex operator+(const Vertex &op2);
   Vertex operator+(int op2);
   Vertex operator=(const Vertex &op2);
   void show(void);
};
```

Потрібна ще така функція перевантаження оператора ‘+’

```
//---а ця функція не є членом класу Vertex---
//---ім’я класу Vertex тут повинно бути вже відоме---
Vertex operator+(int op1, const Vertex &op2)
{
   Vertex tmp(op1,op1,op1);
   return tmp + op2;
}
```

Перевантаження оператора "[ ]"

Оператор [ ] можна перевантажувати тільки як функцію-член класу. Нижче наведена основна форма визначення оператора-функції operator[](..)

```
Тип ІмяКласу::operator[](int індекс)
{
. . .

}
```

Загалом, параметр цього оператора не обов’язково повинен бути цілим, але, оскільки оператор зазвичай використовується для отримання значення по індексу масиву, то параметр зазвичай має тип **int.**

Щоб зрозуміти, як працює **operator[]**, уявимо, що об’єкт A індексується наступним чином: **res = A[9].** Такий запис компілюється у виклик функції-члена **operator[](..)**

```
res = A.operator[](9)
```

Таким чином, значення **9** передається функції **operator[](..)** у якості параметра. При цьому вказівник **this** буде посилатися на об’єкт **A** – джерело виклику.

На основі перевантаженого оператора ”[ ]” можна запрограмувати «безпечний» массив.

```
const int SIZE = 100;
//--- клас "безпечного масиву" ---
class MyArray
{
private:
    int p[SIZE];
public:
    MyArray(void);
    int& operator[](int);
};

MyArray::MyArray(void)
{
```

```
    for (int i = 0; i < SIZE; i++)
        p[i] = i;
}

int& MyArray::operator[](int index)
{
    if (index < 0 || index > SIZE - 1)
    {
        cout << "Index error\n";
        return p[0];
    }
    return p[index];
}
```

```
//--test--
int main()
{
    MyArray A;

    cout << "A[9] = " << A[9] << "\n\n";

    A[10] = A[9] + 5;
    cout << "A[10] = " << A[10] << "\n\n";

    cout << "A[-23] = " << A[-23] << "\n\n";

    int res = A[105];
    cout << "A[105] = " << res << "\n\n";

    return _getwch();
}
```

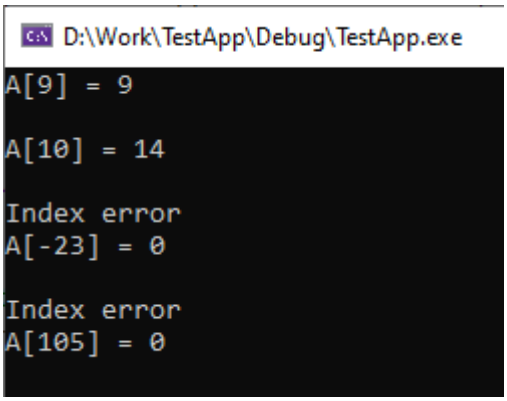


Рис. 8.9. Виконання тесту

Завдяки тому, що функція operator[] повертає посилання на елемент, що індексується, оператор індексування може розташовуватися як праворуч '=' так і у лівій частині присвоювання.

Перевантаження операторів у деяких інших мовах програмування

Мова Java. Оператори не можуть перевантажуватися у Java. Єдиний виняток – оператор “+” для об’єктів типу String (конкатенація)

```
String a = "Hello";
String b = ", world!";
String c;

c = a + b;
```

Результат **c** буде **Hello, world!**

А як, все ж таки, реалізувати додавання членів двох класів, визначених користувачем? Можна, наприклад, так:

```
class MyClass {
private:
    int x, y, z;
public:
    void add(MyClass op) {
         x += op.x;
         y += op.y;
         z += op.z;
    }
}
```

```
MyClass a = new MyClass();
MyClass b = new MyClass();
. . .
a.add(b);      //діє як  a = a + b;
```

Для додавання зроблений метод add( ).

Підсумки щодо перевантаження операторів

Перевантаження операторів дозволяє визначити операції над об’єктами класів користувача так само, як і над об’єктами стандартних типів.

Така можливість часто використовується багатьма програмістами.Мова C++ дає широкі можливості для цього.

Але, перевантаження операторів суттєво ускладнює реалізацію підтримки мови програмування. Це є головною причиною того, що автори мови Java та деяких інших мов взагалі відмовилися від перевантаження операторів [10].

На додачу, перевантаження операторів часто робить складним для розуміння вихідний текст. Особливо, якщо не зовсім зрозуміло, що виконує певний оператор для об’єктів деяких класів.

Нижче наведений перелік мов програмування з точки зору реалізації можливості перевантаження операторів за даними джерела: https://en.wikipedia.org/wiki/Operator_overloading

Таблиця 8.1

Мови програмування, у яких немає перевантаження операторів, або суттєво обмежено	Мови програмування, у яких є можливості перевантаження операторів
C, Java, JavaScript, Objective-C, Pascal, PHP	Ada, C++, C#, Object Pascal, Perl, Python, Ruby, Kotlin, Rust, Swift

Контрольні запитання

1. Як створити об'єкт, який буде копію вже існуючого об'єкта?
2. Що робить конструктор копії і як він оголошується?
3. Як записати операцію присвоювання одного об'єкта іншому?
4. Що таке перевантаження операторів в C++?
5. Як запрограмувати перевантаження оператора присвоювання?
6. Як організувати додавання об'єктів?
7. Чим особливий випадок додавання "число + об'єкт"?
8. Як перевантажити оператор "[]"?
9. Які можливості перевантаження операторів в інших мовах програмування?

Лекція 9

Тема 3.3. Операції з об'єктами (продовження)

Передача об’єкта через параметри функції

Об’єкти можна передавати функціям та методам так само, як і значення будь-якого іншого типу даних.

```
class SomeClass
{
. . .
public:
   void method(..);
};

void Func(SomeClass ob)        //ob – це копія об’єкта,
{                              //яка буде використана у функції
. . .
ob.method(..);
. . .
}

SomeClass obj;
Func(obj);      //передача об’єкта через параметр по значенню
```

Передача через параметри **по значенню** означає те, що автоматично створюється копія об’єкту, яка і відіграє потім роль значення параметра функції. Це можна вважати захистом від змін для об’єкту, який передається – але ж у тілі функції будуть виконуватися дії не над самим об’єктом, а його копією. Але для класів не усе так просто, як здається. Якщо деякий клас планується використовувати як параметр функцій (методів), то у цьому класі має бути конструктор копії. Розглянемо наступний приклад

```
class myclass
{
private:
    int xo, yo;
    int* p;
public:
    myclass(int, int);
    myclass(const myclass& ob);
    ~myclass();
    void show(void);
};

myclass::myclass(int x, int y)
{
    cout << "Constructor\n";
    xo = x;
    yo = y;
    p = new int[4];
```

```
    for (int i = 0; i < 4; i++)
        p[i] = xo + yo + i;
}

myclass::myclass(const myclass& ob)
{
    cout << "Copy constructor\n";
    xo = ob.xo;
    yo = ob.yo;
    p = new int[4];
    for (int i = 0; i < 4; i++)
        p[i] = ob.p[i];
}

myclass::~myclass()
{
    cout << "Destructor\n";
    if (p) delete[]p;
    p = 0;
}

void myclass::show(void)
{
    cout << "x = " << xo << '\n';
    cout << "y = " << yo << '\n';
    cout << "p = " << p << '\n';
    cout << "p[] = ";
    for (int i = 0; i < 4; i++)
        cout << p[i] << " ";
    cout << "\n\n";
}

void Func(myclass ob)    //"класовий" параметр
{
    ob.show();
}
```

```
//--test--
int main()
{
    myclass A(2,5);
    A.show();

    cout << "Call function\n";
    Func(A);
    cout << "Exit function\n";

    return _getwch();
}
```

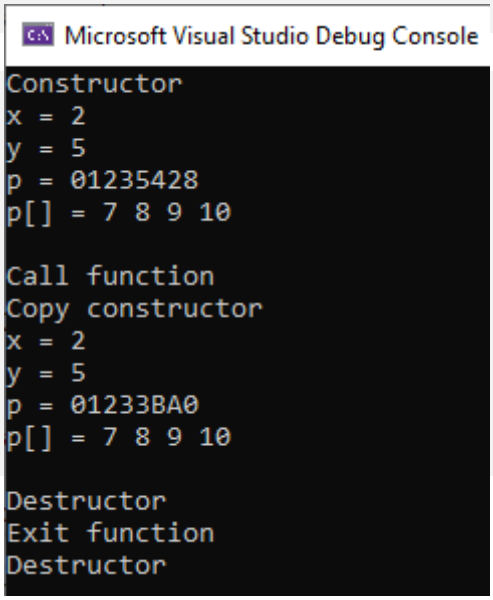


Рис. 9.1. Тестова програма працює без проблем і демонструє те, що очікувалося.

Передача функції у якості параметра вказівника на об'єкт

Як вказувалося у попередньому п., якщо функції через параметр передається екземпляр об'єкту (передавання параметра по значенню), то автоматично створюється копія об'єкту і усередині функції робота йде з цією копією.

На відміну цього можливо передавати функції у якості параметра не об'єкт, а вказівник на об'єкт, тобто адресу об'єкта

```
someclass obj;
Func(&obj);     // &obj – це адреса obj
```

У цьому випадку копія об'єкта не створюється. По вказівнику функція отримує доступ безпосередньо до вказаного екземпляра об’єкта.

Розглянемо приклад

Рис. 9.2. Два способи передачі об’єкта у якості параметру функції

У випадку отримання адреси об'єкта функція може змінювати стан цього об'єкта

Вказівник чи посилання? В мові C++ може бути такий різновид типу параметрів – посилання. Це позначвється символом ‘&’

```
Func(someclass& o)   //параметр - посилання на об'єкт класу someclass
{
   o.method();
}
```

Порівняємо це з вказівником у наступному прикладі

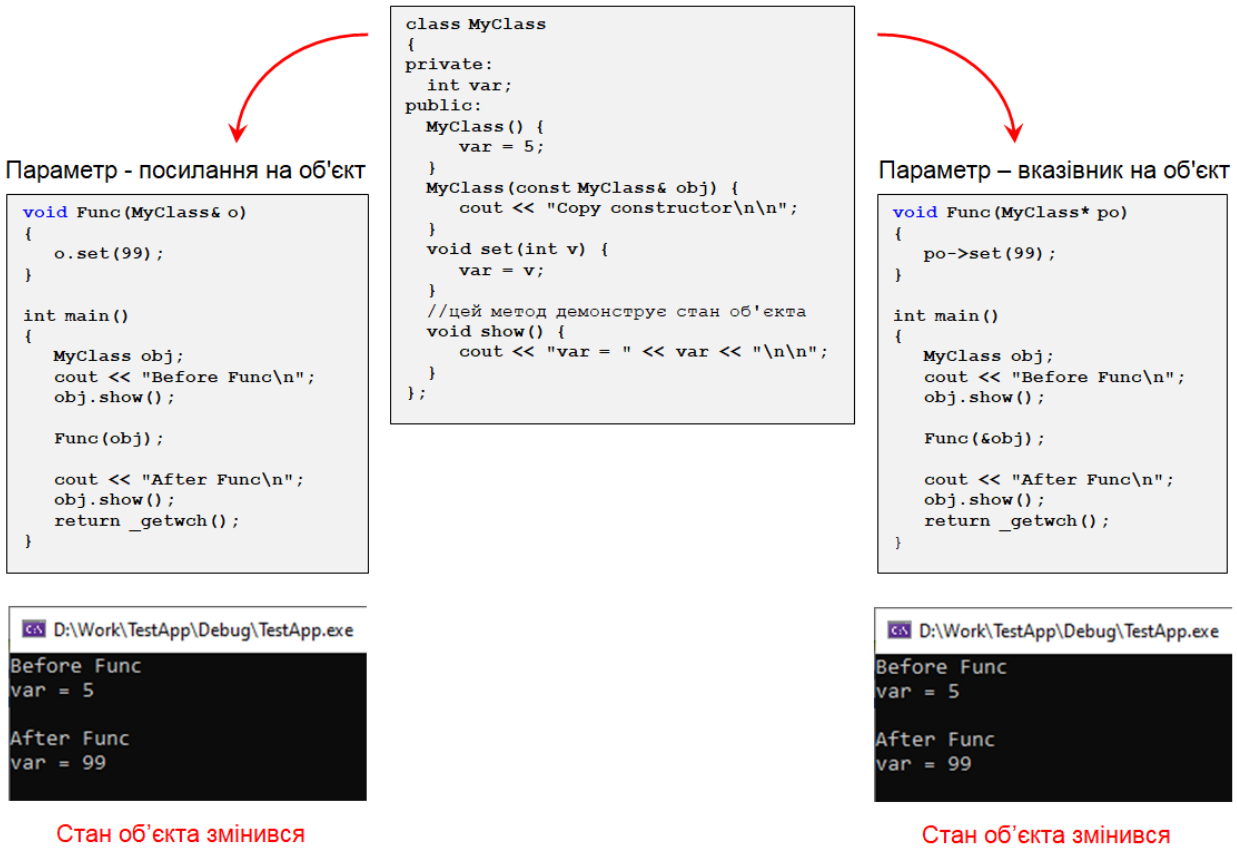


Рис. 9.3. Порівняння вказівника і посилання у якості параметру функції

Зовні не помітно якихось відмінностей у роботі досліджуваних варіантів для останніх двох тестів.

Відмінності посилань від вказівників

Посилання не є вказівником, а просто **є іншим ім'ям для об'єкта**. Головна відмінність посилання від вказівників: вказівник це число (ціле), яке може означати будь-яку адресу. З цим числом можна виконувати різноманітні операції. А для посилання доступні тільки 2 операції: копіювання і розіменування.

Принципових відмінностей дві:

1. Посилання завжди має бути ініціалізовано, а для вказівників це не обов'язково.

2. Посилання не може бути змінено після ініціалізації.

Це обумовлює плюси і мінуси використання того чи іншого:

- **посилання** краще використовувати, коли небажано або не планується зміна зв'язку посилання - об'єкт;
- **вказівник** краще використовувати, коли можливі такі моменти в перебігу життя посилання:
  - якщо потрібне посилання, яке не вказує ні на який об'єкт;
  - якщо посилання може вказувати на різні об'єкти впродовж свого часу життя.

Можна вказати ще такі моменти:

- Не можна оголосити масив посилань.
- У посилання немає адреси.
- Існує арифметика покажчиків, але немає арифметики посилань.
- Вказівник може отримати визначене недійсне значення (nulpter, null, або просто 0) з яким його можна порівняти перед використанням для запобігання помилок доступу.

Вказівники притаманні мовам C та C++. Ці мови дозволяють працювати з адресами пам'яті. Також у C/C++ можна програмувати з використанням синтаксису посилань.

Вказівники не використовуються у таких мовах програмування, як Java, C# та інших, які орієнтовані на середовище виконання у вигляді віртуальної машини. У цих мовах використовуються виключно посилання.

Об’єкт у якості результату роботи функції

Розглянемо, як працює наступний програмний код

```
SomeClass res = Func( );
```

У цьому рядку результатом роботи функції буде об'єкт класу SomeClass. Можна зробити таке припущення: це зовні схоже на ініціалізацію об’єкта **res** в процесі його створення. Це вже ми розглядали раніше і з'ясували, що у класі має бути конструктор копії.

З іншого боку, повернення результату від функції зазвичай працює як оператор присвоювання. Тоді що, треба перевантажувати operator+ ?

Для досліджень зробимо наступну тестову програму

```
class myclass
{
private:
    int xo, yo;
    int* p;
public:
    myclass(int, int);
    myclass(const myclass& ob);
    ~myclass();
    myclass& operator=(const myclass& ob);
    void show(void);
};
```

```
myclass::myclass(int x, int y)
{
    cout << "Constructor\n";
    xo = x;
    yo = y;
    p = new int[4];
    for (int i = 0; i < 4; i++)
        p[i] = xo + yo + i;
}

myclass::myclass(const myclass& ob)
{
    cout << "Copy constructor\n";
    xo = ob.xo;
    yo = ob.yo;
    p = new int[4];
    for (int i = 0; i < 4; i++)
        p[i] = ob.p[i];
}

myclass::~myclass(void)
{
    cout << "Destructor\n";
    if (p) delete[]p;
    p = 0;
}

myclass& myclass::operator=(const myclass& ob)
{
    cout << "Overloaded operator =\n";
    xo = ob.xo;
    yo = ob.yo;
    for (int i = 0; i < 4; i++)
        p[i] = ob.p[i];
    return *this;
}

void myclass::show(void)
{
    cout << "x = " << xo << '\n';
    cout << "y = " << yo << '\n';
    cout << "p = " << p << '\n';
    cout << "p[] = ";
    for (int i = 0; i < 4; i++)
        cout << p[i] << " ";
    cout << "\n\n";
}
```

```
//результат цієї функції є об’єктом класу myclass
myclass Func(void)
{
    cout << "Start function\n" << "Object A\n";
    myclass A(2, 5);
    A.show();
    cout << "End function\n";
    return A;
}

//--test--
int main()
{
    cout << "Object B\n";
    myclass B(7, 11);
    B.show();

    cout << "Call function\n";
    B = Func();
    cout << "Object B = Func()\n";
    B.show();

    return _getwch();
}
```

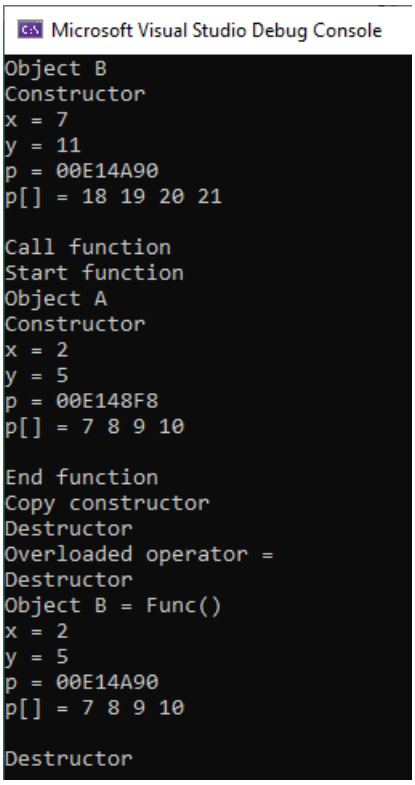


Рис. 9.4. Виконання тестової програми

З того, що тестова програма виводить на консоль, стає зрозуміло, що конструктор копії та перевантажений оператор ‘=’ дійсно потрібні і вони працюють, як і очікувалося. Об'єкт B коректно ініціалізується відповідними значеннями членів локального об'єкта A. Тест виконується без проблем.

Повернення функцією вказівника на об’єкт

Розглянемо, як працює наступний програмний код у форматі

```
SomeClass* po = Func( );
```

на прикладі класу myclass, вже опрацьованого у попередньому пп. Сам клас залишимо без змін, але змінимо тестову программу наступним чином

```
//--- тестування *po=Func() ---
myclass* Func(void)
{
    cout << "Start function\n";
    myclass A(2, 5);
    A.show();
    cout << "End function\n";
    return &A;
}

int main()
{
    cout << "Call function\n";
    myclass* po = Func();
    cout << "po = Func()\n";
    po->show();
    return _getwch();
}
```

Microsoft Visual Studio C++ створює виконуваний код, але компілятор при цьому видає таке попередження

```
warning C4172: returning address of local variable or temporary: A
```

Тестова програма аварійно переривається на вказаному нижче рядку

```
    . . .
    myclass* po = Func();
    cout << "po = Func()\n";
    po->show();     //така інструкція призводить до краху програми
```

У чому проблема? Виявляється, що Func( ) записує недійсне значення у вказівник po. Тепер стає зрозумілим попередження компілятора C++.

Функція не повинна повертати вказівник на свій локальний об’єкт. Це найгрубіша помилка програміста C та C++, свідчення його некомпетентності, оскільки треба завжди враховувати, що при виході з функції локальні об’єкти стають недійсними, а їхні адреси вже нічого не означають.

А як бути, якщо потрібна функція, яка створює об'єкт і видає вказівник на нього у якості результату? Якщо потрібен об'єкт, який має існувати після виходу з функції, то можна у тілі функції створити динамічний об'єкт, використовувати його впродовж роботи програми і не забути знищити цей об'єкт тоді, коли він вже стане не потрібен. Наступний тест

```
//--- тестування *po=Func() ---
myclass* Func(void)
{
    cout << "Start function\n";
    myclass* pc = new myclass(2, 5);
    pc->show();
    cout << "End function\n";
    return pc;
}

int main()
{
    cout << "Call function\n";
    myclass* po = Func();
    cout << "po = Func()\n";
    po->show();
    delete po;    //знищення об'єкту
    return _getwch();
}
```

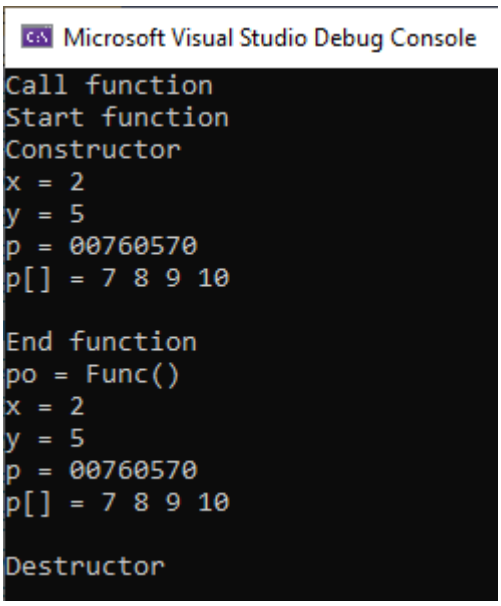


Рис. 9.5. Виконання тесту створення динамічного обєкту

Тестова програма демонструє створення уередині функції Func() динамічного об'єкту, передачу вказівника на нього, успішне використання об'єкту зовні функції та прикінцеве знищення динамічного обєкту.

Підсумки

У таблиці нижче зібрані до купи проаналізовані вище варіанти виконання операцій з об'єктами. Кожна операція зовні, як здається програмісту-користувачу певних класів, маніпулює об'єктом як єдиним цілим. Але для цього програміст-розробник цих класів повинен забезпечити у програмному коді для цих класів реалізацію потрібних операцій.

Таблиця 9.1

Дія	Запис	Примітка	Конструктор звичайний	Конструктор копії	Коментар
Присвоювання об'єктів	objB = objA;	За умовчанням -- побітове копіювання усіх даних-членів об'єктів	Працює	-	Перевантаження оператора "="
Ініціалізація об'єкта іншим об'єктом	classname objB = objA;	За умовчанням -- побітове копіювання усіх даних-членів об'єктів	-	Працює	Спеціальне визначення конструктора копії
Передавання об'єкта у якості параметра функції	Func (classname obj);	При передачі автоматично створюється копія об'єкту для використання у тілі функції	-	Працює	Спеціальне визначення конструктора копії
Передавання вказівника на об'єкт у якості параметра функції	Func (classname *po);	Копіюється тільки вказівник	Працює зовні при створенні обєкта	-	Проблем немає
Повернення об'єкта у якості результата функції	obj = Func( );	Побітове копіювання усіх даних-членів об'єктів	У тілі функції	Працює	Перевантаження оператора "=" та визначення конструктора копії
Повернення вказівника на об'єкт у якості результата функції	classname *po = Func( );	Копіюється тільки вказівник	У тілі функції	-	Функція не повинна повертати вказівник на свій локальний об'єкт

Контрольні запитання

1. Що потрібно передбачити у класі задля забезпечення можливості передавання його об'єктів у якості параметра функції?
2. Як передавати через параметр функції вказівник на об'єкт і які можливості це може дати?
3. Які відмінності посилань від вказівників на об'єкти?
4. Що потрібно передбачити, щоб функції могли повертати об'єкти у якості результату своєї роботи?
5. Чому не є коректним повернення функціями вказівників на локальні об'єкти?

Лекція 10

Тема 4.1. Шаблони C++

Поняття шаблонів

Шаблони (*templates*) у мові C++ є потужним засобом повторного використання коду. Завдяки шаблонам можна створювати узагальнені (*generic*) функції та класи, які реалізують методи та алгоритми без прив'язування до конкретних типів даних. Далі ми такі функції та класи будемо називати "шаблонними".

Оголошення шаблонного класу або шаблонної функції формально можна представити так

```
template < список параметрів > оголошення
```

Таким чином, усе починається з ключового слова **template**, далі у трикутних дужках список параметрів і далі оголошення класу або функції. Список параметрів шаблону містить один або декілька розділених через кому параметрів

```
Параметр-оголошення
class ідентифікатор [ = ім'я типу ]
typename ідентифікатор [ = ім'я типу ]
template < список параметрів > class [ідентифікатор][= ім'я]
```

Кожний параметр зі списка у трикутних дужках може бути ідентифікатором з ключовим словом class або typename, або іменем з деяким типом, наприклад

```
template < class T, int i > class MyClass...
```

У цьому рядку оголошується шаблонний клас MyClass, причому **`class T`** – це параметризований тип, **`int i`** – константний параметр, який буде репрезентувати якесь числове значення. Багатокрапкою позначено, що далі записується текст оголошення власне класу MyClass.

Шаблонні функції

Розглянемо простий приклад. Нехай потрібно мати функцію, яка обчислює максимальне значення двох параметрів типу **int**, а також функцію, яка знаходить максимальне з двох чисел типу **double**. Вочевидь, можна створити для такої функції дві її версії з різними іменами

```
int maxi(int a,int b)
{
return a > b ? a : b;
}

double maxd(double a,double b)
{
return a > b ? a : b;
}
```

А потім ці функції викликати відповідно типу значень, які потрібно обробляти

```
int ires,x,y;
double dres,val1,val2;
. . .
ires = maxi(x, y);
. . .
dres = maxd(val1, val2);
```

Зауваження. Як здається, функція **maxi** взагалі не потрібна. Можна всюди використовувати функцію **maxd** і програма буде обчислювати правильні результати

```
ires = maxd(3, 5);              //результат буде 5
. . .
dres = maxd(7.45, 4.13);        //результат буде 7.45
```

Але, оскільки double-версія такої функції працює швидше то існування скомпільованих різних версій функції обчисленя макс. значення є виправданим. А тепер перейдемо до шаблонних функцій.

Шаблонна функція оголошується і визначається наступним чином

```
template <class Ттип> типрез Ім'яФункції(список параметрів)
{
. . .     //тіло функції
}
```

Елемент **Ттип** є фіктивним іменем, яке можна використати при оголошенні та визначенні функції. Тип результату функції (**типрез**) може бути й **Ттип.** Наприклад:

```
template <class X> X maxt(X a, X b)
{
return a > b ? a : b;
}
```

Потім компілятор автоматично замінить фіктивне ім'я **X** на конретний тип. Якщо компілятору буде чітко зрозуміло, який тип повинен бути, то виклик шаблонної функції можна записати так:

```
int ires,x,y;
double dres,val1,val2;

ires = maxt(x, y);          //компілятор створить int-версію функції

dres = maxt(val1, val2);    //компілятор створить double-версію
```

Нижче наведений код з помилками:

```
int ires,x,y;
double dres,val1,val2;

ires = maxt(val1, y);    //error C2782: 'S maxt(S,S)' : template parameter 'S'
                         //is ambiguous

dres = maxt(x, val2);    //error C2782: 'S maxt(S,S)' : template parameter 'S'
                         //is ambiguous
```

При виклику функцій в один з параметрів підставлена змінна типу **int**, в інший – **double**. Виникає помилка неоднозначності – компілятор не може автоматично визначити версії функцій, які потрібно створювати та викликати.

Для вирішення цієї проблеми потрібно явно вказати, який тип компілятор повинен відпрацювати

```
int ires,x,y;
double dres,val1,val2;

ires = maxt<int>(val1, y);      //компілятор створить int-версію коду

dres = maxt<double>(x, val2);   //компілятор створить double-версію коду
```

Якщо тип вказувати явно, то можна і так:

```
dres = maxt<char>(x, val2);  //компілятор створить також і char-версію коду
```

Для шаблонів замість ключового слова **class**

```
template <class S> S maxt(S a, S b)
```

рекомендується використовувати слово **typename** (якого не було у перших версіях C++)

```
template <typename S> S maxt(S a, S b)
```

Крім того, щоб зробити текст зрозумілішим, рекомендується розділити цей рядок наступним чином

```
template <typename S>
S maxt(S a, S b)
{
return a > b ? a : b;
}
```

Якщо текст оголошення-визначення щоблонної функції міститься після її виклику, то компілятор видасть помилку про невідоме ім'я. У таких випадках треба зробити попереднє оголошення функції – так само, як і для звичайної функції

```
template <typename S> S maxt(S a, S b);   //попереднє оголошення
. . .
ires = maxt<int>(x, y);        //виклик
. . .
template <typename S> S maxt(S a, S b)    //визначення
{
return a > b ? a : b;
}
```

Спеціалізація шаблонів функцій

Незважаючи на те, що шаблонні функції автоматично типізуються (перевантажуються) компілятором, можливо для шаблонної функції явно вказати окремий тип, для якого програмний код (поведінка) буде відрізнятися від вже визначеного програмного коду для типів загалом. Це зветься **спеціалізацією** (*explicit specialization*) шаблонів.

Для спеціалізації шаблонів функцій використовується синтаксична конструкція **template<>**

```
template<> тип_функції Ім'яФункції<ТипСпец>(список параметрів функції)
```

Елемент **ТипСпец** повинен вказувати ім'я типу, для якого робиться виняток із загального визначення шаблону

Приклад. Маємо оголошення загальної поведінки функції **maxt** у вигляді шаблонної функції

```
template <typename T> T maxt(T a, T b)
{
return a > b ? a : b;
}
```

Для наступного типу – вказівника на структуру **Date**

```
struct Date
{
int year,mon,day;
};
```

визначимо спеціалізацію шаблону:

```
template<> Date* maxt<Date*>(Date* a, Date* b)
{
long da = ((a->year)*12 + a->mon)*31 + a->day;
long db = ((b->year)*12 + b->mon)*31 + b->day;
return da > db ? a : b;
}
```

Вочевидь, знаходження максимальної дати суттєво відрізняється від простого порівняння двох чисел. Крім того, функція буде повертати не значення структури, а вказівник на структуру, яка містить максимальну дату – одну зі структур, вказану у параметрах.

```
template <typename T> T maxt(T a, T b)
{
return a > b ? a : b;
}

struct Date
{
int year,mon,day;
};

template<> Date* maxt<Date *>(Date *a, Date *b)
{
long da = ((a->year)*12 + a->mon)*31 + a->day;
long db = ((b->year)*12 + b->mon)*31 + b->day;
return da > db ? a : b;
}
```

Тест:

```
Date sa,sb;
sa.year = 2015;
sa.mon = 9;
sa.day = 3;
sb.year = 2016;
sb.mon = 1;
sb.day = 2;
int ires = maxt<int>(3, 5);
Date* pres = maxt<Date *>(&sa, &sb);
```

Спеціалізацій може бути декілька. Так, наприклад, для структури типу **Vertex** доцільно вважати, що для двовимірних радіус-векторів, які

описуються структурами **Vertex,** максимальним елементом буде найдовший вектор.

Об'єднаємо оголошення-визначення шаблонної функції maxt( ) та усіх її спеціалізацій у файлі **maxt.h**

```
template <typename T> T maxt(T a, T b)
{
return a > b ? a : b;
}

struct Date
{
int year,mon,day;
};

template<> Date *maxt<Date *>(Date *a, Date *b)
{
long da = ((a->year)*12 + a->mon)*31 + a->day;
long db = ((b->year)*12 + b->mon)*31 + b->day;
return da > db ? a : b;
}

struct Vertex
{
double x,y;
};

template<> Vertex *maxt<Vertex *>(Vertex *a, Vertex *b)
{
double da = (a->x)*(a->x) + (a->y)*(a->y);
double db = (b->x)*(b->x) + (b->y)*(b->y);
return da > db ? a : b;
}
```

Далі приклади варіантів використання зробленої нами шаблонної функції maxt()

```
#include "maxt.h"

int ires,x,y;
Date sa,sb,*ps;
Vertex va,vb, *pv;

ires = maxt<int>(x, y);    //інстанціювання для int від загального шаблону

sa.year = 2015;
sa.mon = 9;
sa.day = 3;

sb.year = 2016;
sb.mon = 1;
sb.day = 2;

ps = maxt<Date*>(&sa, &sb);      //інстанціювання для спеціалізації Date*
```

```
va.x = 3.4;
va.y = 2.1;

vb.x = 10.1;
vb.y = -2.7;

pv = maxt<Vertex*>(&va, &vb);    //інстанціювання для спеціалізації
                                 // Vertex*
```

Шаблонні класи

Окрім шаблонних (узагальнених) функцій можна створити шаблонні (узагальнені) класи. При цьому створюється клас, у якому запрограмовані усі необхідні алгоритми, проте фактичний тип даних задається при створенні екземпляру об'єкта. Можна сказати, що у такий спосіб можна описати загальну поведінку об'єкта абстрагуючись, не прив'язуючись до типів якихось даних.

Просте оголошення шаблонного класу має вигляд:

```
template <typename Ттип> class Ім'яКласу
{
. . .
};
```

Примітка. Замість **typename** може записуватися слово **class**.

Фіктивне ім'я типу, яке вказується у якості **Ттип,** може використовуватися у данних-членах та функціях-членах класу при їхньому оголошенні та визначенні.

Створення екземляру об'єкта може робитися так:

```
Ім'яКласу <конкретний_тип> Ім'яОб'єкту;
```

або динамічно:

```
Ім'яКласу <конкретний_тип> *вказівник;
вказівник = new Ім'яКласу <конкретний_тип>;
```

Розглянемо приклад побудови шаблонного класу. Раніше ми вже розглядали приклад класу стека для зберігання чисел типу int

```
class IntStack
{
private:
   int p[100];
   int num;
public:
```

```
    MyStack(void);
    void Push(int);
    int Get(void);
};
```

Для узагальнення класу для типів даних, які можуть зберігатися у стеку, замість **int** позначимо тип даних фіктивним ім'ям **SType** у шаблонному класі **MyStack**

```
template <typename SType>
class MyStack
{
private:
    SType p[100];
    int num;
public:
    MyStack(void);
    void Push(SType);
    SType Get(void);
};
```

Чому наш шаблонний клас стеку має фіксовану кількість елементів робочого масиву (100)? Синтаксис шаблонів дозволяє узагальнити клас і в цьому аспекті. Оголошення шаблонного класу зробимо вже так:

```
template <int volume, typename SType>
class MyStack
{
private:
    SType p[volume];
    int num;
public:
    MyStack(void);
    void Push(SType);
    SType Get(void);
};
```

Фіктивне ім'я volume буде позначати об'єм робочого масиву стеку.

Таким чином, можна стверджувати, що ми досягли деякого узагальнення, описуючи сховище для будь-якої кількості елементів будь-якого типу. Хоча, якщо прискіпливіше оглянути шаблонний клас MyStack, то можна поставити таке запитання: а чому лічильник (num) кількості елементів та та обсяг (volume) ми зробили int. Невже і тут треба узагальнювати – замість типу int для num та volume також записувати фіктивний параметризований тип? Авжеж, починаючи з певної кількості елементів, вона (кількість) вже не може бути описана значенням типу int. Залишимо це на розсуд програміста.

Далі визначення функцій-членів шаблонного класу **MyStack**:

```
template <int volume, typename SType>
MyStack<volume, SType>::MyStack(void)
{
num = 0;
}

template <int volume, typename SType>
SType MyStack<volume, SType>::Get(void)
{
if (num < 1) return 0;
num--;
return p[num];
}

template <int volume, typename SType>
void MyStack<volume, SType>::Push(SType elem)
{
if (num < volume)
   {
   p[num] = elem;
   num++;
   }
}
```

Створення екземплярів двох об’єктів-стеків і робота з ними

```
int ires;
double dres;

MyStack <25, int> SI;
MyStack <30, double> *psd;

SI.Push(99);
SI.Push(13);
ires = SI.Get();

psd = new MyStack <30, double>;

psd->Push(8.73);
psd->Push(1.52);
dres = psd->Get();

delete psd;
```

Шаблони і узагальнене програмування

Шаблони C++ використовуються для підтримки *узагальненого програмування.*

Примітка. Не плутати шаблони C++ (*templates*) з патернами проєктування (*design patterns*), які часто також називають шаблонами.

Під узагальненим програмуванням можна розуміти опис потрібних алгоритмів у вигляді такого вихідного тексту (для програмного коду), який не привязується до конкретних типів даних, і може бути застосований без змін для будь-яких типів даних.

Звичайно, конкретні деталі поведінки алгоритму можуть бути визначені конкретними типами даних, для яких створюються екземпляри об’єктів.

Шаблони дозволяють створювати єдиний код для алгоритмів, поведінка яких повинна бути узагальнена на різні типи даних. Імена узагальнених типів даних діють як деякі параметри для узагальнених функцій і класів.

Можна сказати, що шаблони C++ – це *поліморфізм параметризованих типів*.

Багато мов програмування надають загальні можливості програмування, подібні до шаблонів C++, наприклад, Java, C# та деякі інші. Доволі часто для позначки узагальнення використовується англійське слово “*generic*”.

Контрольні запитання

1. Яким ключовим словом позначаються шаблонні функції та шаблонні класи?
2. Яким ключовим словом позначаються параметризовані типи в шаблонах?
3. Навіщо потрібна спеціалізація шаблонів?
4. Що таке поліморфізм параметризованих типів?
5. Як шаблони втілюють поліморфізм?
6. Що таке узагальнене програмування?

Лекція 11

Тема 4.2. Стандартна бібліотека C++. Контейнери та алгоритми

Огляд стандартної бібліотеки C++

Стандартна бібліотека C/C++ містить множину класів та функцій, які програмісти можуть вільно використовувати у своїх проєктах C++

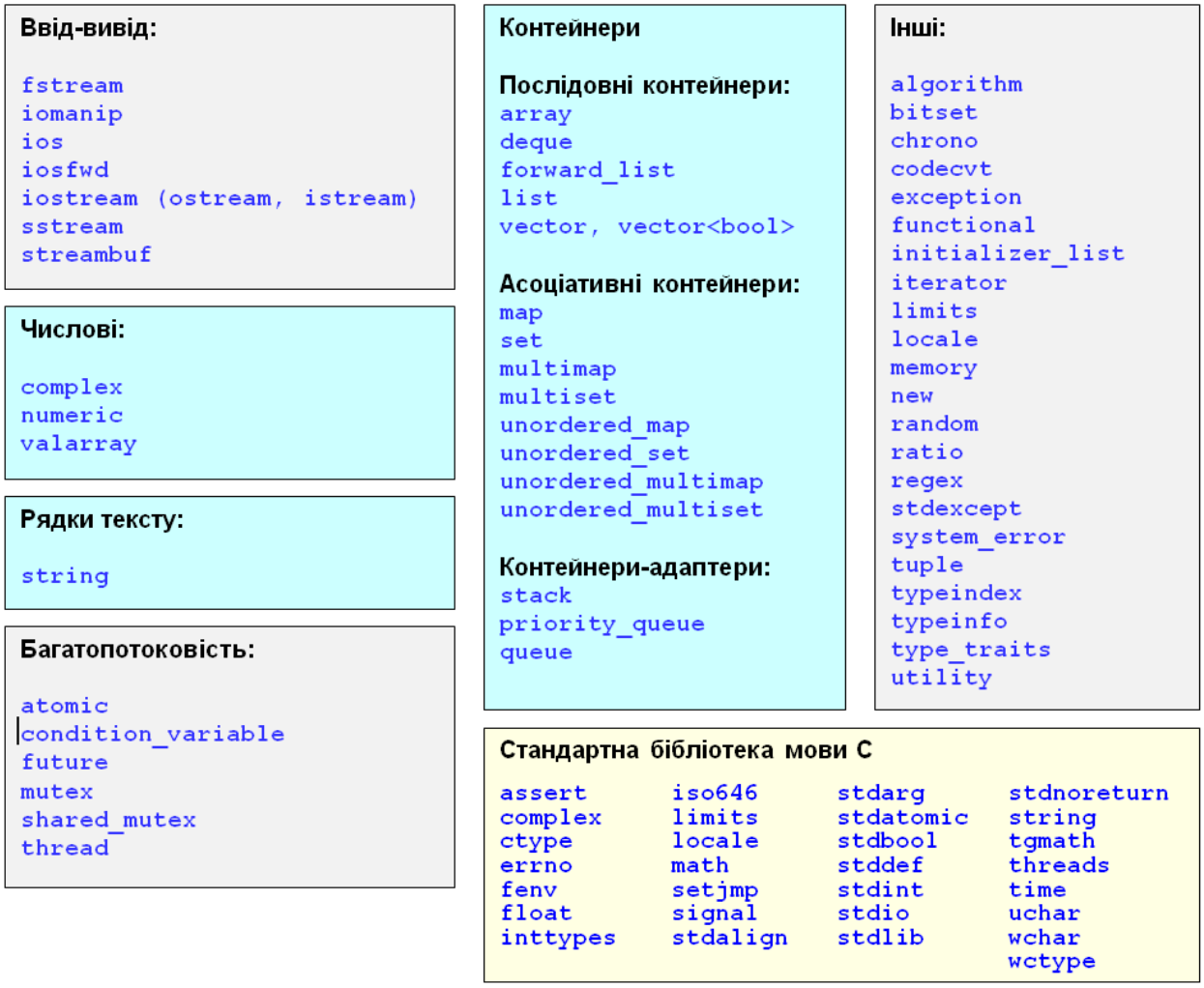


Рис. 11.1. Склад стандартної бібліотеки C/C++

У стандартній бібліотеці C++ міститься множина класів, які зроблені як шаблонні класи. Тому цю частину бібіотеки часто звуть стандартною бібліотекою шаблонів (STL – *Standard Templates Library*). У цій бібліотеці можна виділити такі поняття:

1. Контейнер (*container*) – для зберігання множини об'єктів у памяті
2. Ітератор (*iterator*) – для забезпечення доступу до елементів контейнера
3. Розподільник (*allocator*) – для виділення пам’яті для елементів

4. Алгоритм (*algorithm*) – визначення процедури обробки

5. Адаптер (*adaptor*) – адаптація різних інтерфейсів для узгодження забезпечення доступу, наприклад, у випадку контейнерів-адаптерів.

6. Функціональний об’єкт (*functor*) – реалізація у об’єкті функції, яка визначає поведінку деяких інших компонентів

Побудова бібліотеки у вигляді шаблонних класів та функцій дозволяє зменшити кількість компонентів. Наприклад, замість написання усіх версій функції пошуку для кожного типу контейнера, забезпечується тільки одна версія, яка здатна працювати з кожним типом, який задовольняє основним вимогам.

Для використання класів, наприклад, класу complex зі стандартної бібліотеки шаблонів C++, треба записати

```
#include <complex>
using namespace std;
```

Необхідно зазначити, що у теперішній час, зокрема, у версії стандарту ISO C редакції 2022р., підмножина STL вже не виділяється і ніяк не позначається як якась окрема частина стандартної бібліотеки C/C++.

В стандартну бібліотеку C/C++ також включена стандартна бібліотека мови C, яка містить набір функцій, оголошених у відповідних заголовочних файлах. Наведений вище перелік компонентів стандартної бібліотеки C відповідає стандарту ISO C.

Для того, щоб використовувати у якомусь модулі C/C++, наприклад, математичні функції стандартної бібліотеки C math, у вихідному тексті цього модуля проєкту Visual Studio C++ можна записати

```
#include <math.h>
```

Для заголовків вже можуть не використовуються розширення .h і вони позначаються з додаванням ‘c’ попереду – наприклад, замість <math.h> може бути <cmath>.

Існують декілька реалізацій та розширень стандартної біліотеки C, наприклад, реалізація для Unix-подібних ОС, розширення Microsoft C run-time library тощо.

Повне викладення усіх складових стандартної бібліотеки C/C++ виходить за межі цього курсу, але щоб отримати певне уявлення про неї, далі розглянемо деякі окремі компоненти цієї бібліотеки.

Клас complex

Специфікація шаблонного класу **complex** описується наступним чином

```
template <class T> class complex
```

Клас complex містить декілька конструкторів:

```
complex(const T &real = T(), const T &img = T());
complex(class complex &ob);
template <class T1> complex(const complex<T1> &ob);
```

Перший конструктор створює об'єкт класу **complex** із дійсною частиною **real** та мнимою частиною **img**. Якщо не вказано інше, то за замовчуванням ці значення дорівнюють 0. Другий конструктор створює копію об'єкта ob, третій – об'єкт класу **complex** на основі об'єкта ob.

Також у класі complex визначено перевантаження операторів '+', '-', '*', '/', які забезпечують виконання відповідних операцій згідно математиці комплексних чисел. Крім того визначено декілька тригонометричних та інших функцій як методт цього класу.

Клас T визначає тип, який використовується для зберігання елементів комплексних чисел. Існують три спеціалізації класу **complex**

```
class complex<float>
class complex<double>
class complex<long double>
```

Так, зокрема, визначення спеціалізації для типу double описується у вихідному тексті шаблонного класу **complex** наступним чином

```
template<> class complex<double>
{
public:
   complex
   (
      double _RealVal = 0,
      double _ImagVal = 0
   );
   complex
   (
      const complex<double>& _ComplexNum
   );
   explicit complex
   (
      const complex<long double>& _ComplexNum
   );
   // rest same as template class complex
};
```

Приклад створення об’єктів класу **complex** у тестовій програмі

```
#include <iostream>
#include <complex>
using namespace std;

//--test--
int main()
{
    complex<double> cm1(5, 3);
    complex<double> cm2;          //конструктор за замовчуванням
    complex<double> cm3 = cm1;    //конструктор копії
    complex<double> cm4(cm1);

    cout << "cm1 = " << cm1 << endl;
    cout << "cm2 = " << cm2 << endl;
    cout << "cm3 = " << cm3 << endl;
    cout << "cm4 = " << cm4 << endl;

    return _getwch();
}
```

```
D:\Work\TestApp\Debug\TestApp.exe
cm1 = (5,3)
cm2 = (0,0)
cm3 = (5,3)
cm4 = (5,3)
```

Нижче ілюстрація виконання операцій з комплексними числами

```
#include <iostream>
#include <complex>
using namespace std;

//--test--
int main()
{
    complex<double> cm1(5, 3);
    complex<double> cm2(1, -1);
    complex<double> cm3;

    cm3 = cm1 + cm2;

    cout << "cm1 = " << cm1 << endl;
    cout << "cm2 = " << cm2 << endl;
    cout << "cm3 = " << cm3 << endl;

    cm3 *= 10;
    cout << "cm3 = " << cm3 << endl;

    cout << "abs(cm3) = " << abs(cm3) << endl;
    cout << "imag(cm3) = " << imag(cm3) << endl;

    return _getwch();
}
```

```
D:\Work\TestApp\Debug\TestApp.exe
cm1 = (5,3)
cm2 = (1,-1)
cm3 = (6,2)
cm3 = (60,20)
abs(cm3) = 63.2456
imag(cm3) = 20
```

Клас pair

Класс pair можна вважати ще одним з варіантів опису парних значень. На відміну від класу complex, клас pair описує пари значень будь-яких типів. Оголошення шаблонного класу pair нижче

```
template<typename Type1, typename Type2>
struct pair {
   typedef Type1 first_type;
   typedef Type2 second_type;
   Type1 first;
   Type2 second;
   pair(  );
   pair(const Type1& v1, const T2& v2);
   template< typename Other1, typename Other2>
      pair(const pair<Other1, Other2>& _Right);
      void swap(pair<Type1, Type2>& _Right);
};
```

Цей клас має конструктори копіювання та переміщення. Також передбачено функцію **swap**. Для ініціалізації значень елементів пар окрім конструктора можна також скористатися шаблонною функцією **make_pair**, наприклад

```
pair <int, double> p1(10, 1.1e-2);

pair <int, double> p2;
p2 = make_pair(35, 2.22e+10);   //інший спосіб ініціалізації

p2.swap(p1);   //обмін значень елементів пар p1 та p2
```

Доступ до елементів пари можна виконувати прямо і безпосередньо, так і опосередковано через функцію **get** з індексом 0 або 1, наприклад:

```
int a = p1.first;
double b = p1.second;

int x = get<0>(p1);    //альтернатвний спосіб доступу
double y = get<1>(p1);
```

Об'єкти класу **pair** використовуються як елементи деяких контейнерних шаблонних класів стандартної бібліотеки C++, зокрема, асоціативних контейнерів, таких як **map**, **multimap**, **unordered_map**, **unordered_multimap**.

Клас string

Навіщо було створювати клас **string**?

Прийняте у мові С представлення рядків тексту як масив символів, що завершується нулем, породжує незручності, зокрема, такі:

1. Не можна для операцій над рядками використовувати такі оператори, як ‘=’, ‘+’, ‘==’, ‘>’, ‘<’
2. Виконання деяких дій над масивами небезпечно

Фактично, клас **string** є спеціалізацією базового шаблонного класу **basic_string,** який оголошено наступним чином

```
template <
    class CharType,
    class Traits=char_traits<CharType>,
    class Allocator=allocator<CharType>
>
class basic_string
```

CharType – описує символьний тип

Traits – клас властивостей, який визначає операції над символьним типом

Allocator – тип виділення пам’яті

Загалом, є дві спеціалізації класу **basic_string**

- клас **string,** який обробляє рядки однобайтових (char) символів;
- клас **wstring** для обробки рядків розширених (wchar_t) символів

Приклад створення та ініціалізації екземплярів об’єктів-рядків

```
#include <iostream>
#include <string>
using namespace std;

//--test--
int main()
{
    string s1("First string");
    string s2 = "Another string";
    string s3 = s2;

    cout << "s1 = " << s1 << endl;
    cout << "s2 = " << s2 << endl;
    cout << "s3 = " << s3 << endl;

    return _getwch();
}
```

```
D:\Work\TestApp\Debug\TestApp.exe
s1 = First string
s2 = Another string
s3 = Another string
```

Клас **string** містить багато перевантажених операторів

Оператор	Що означає
=	Присвоювання
+	Конкатенація
+=	Присвоювання та конкатенація
==	Рівність
!=	Нерівність
<	Менше
<=	Менше або дорівнює
>	Більше
>=	Більше або дорівнює
[]	Індексація
<<	Вивід
>>	Увід

Достатньо популярною є конкатенація (додавання) рядків. Можна порівняти програмний код з оператором '+' для класу **string** з програмним кодом у випадку представлення рядків масивами і при виконання тієї ж конкатенації рядків функціями **strcat_s** стандартної бібліотеки С.

Рядки – об'єкти класу **string**

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
string s1, s2, s3;

s1 = "First string";
s2 = "Second string";

s3 = s1 + " + " + s2;

cout << "s1 = " << s1 << endl;
cout << "s2 = " << s2 << endl;
cout << "s3 = " << s3 << endl;
return _getwch();
}
```

Рядки представлені масивами символів

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

int main()
{
char s1[64], s2[64], s3[64];

strcpy_s(s1, sizeof(s1), "First string");
strcpy_s(s2, sizeof(s2), "Second string");
strcpy_s(s3, sizeof(s3), s1);
strcat_s(s3, sizeof(s3), " + ");
strcat_s(s3, sizeof(s3), s2);

cout << "s1 = " << s1 << endl;
cout << "s2 = " << s2 << endl;
cout << "s3 = " << s3 << endl;
return _getwch();
}
```

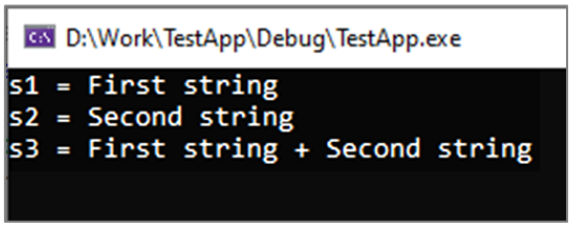


Рис. 11.2. Виконання тесту конкатенації рядків

Функції-члени класу string

Незважаючи на те, що майже усі основні дії над рядками можна виконати з використанням тільки операторів, у класі string є багато функцій-членів.

Функція **assign**( ) присвоює значення одного рядка іншому. Має дві форми з різними наборами параметрів

```
string &assign(const string &strob, size_type start, size_type num);
string &assign(const string *str , size_type num);
```

Безумовно, набагато простіше присвоювати рядок за допомогою оператора "=", але основне призначення метода assign( ) – присвоювання частини від іншого рядка, а також присвоєння рідку-об'єкту рядка від масиву символів.

Функція insert( ) вставляє у рядок інший рядок, або його частину.

```
string &insert (size_type start, const string &strob);
```

Функції знаходження довжини рядка

length( ), size( ) – довжина поточного рядка у байтах

capacity( ) – поточний розмір сховища об'єкта для зберігання рядка

max_size( ) – максимально можливий розмір сховища для рядка для поточної реалізації бібліотеки для системного середовища виконання.

Приклад використання вказаних функцій

```
//--test--
int main()
{
    string s1("My flowers in window! Quick fox eat it");
    string s2;

    cout << "s1 = " << s1 << endl;

    cout << endl << "s2 = part of s1" << endl;
    s2.assign(s1, 22, 100);
    cout << "s2 = " << s2 << endl;

    cout << endl << "s2 = insert \"strong\"" << endl;
    s2.insert(6, "strong ");
    cout << "s2 = " << s2 << endl;

    cout << "s2.length = " << s2.length() << endl;
    cout << "s2.size = " << s2.size() << endl;
    cout << "s2.capacity = " << s2.capacity() << endl;
    cout << "s2.max_size = " << s2.max_size() << endl;
    return _getwch();
}
```

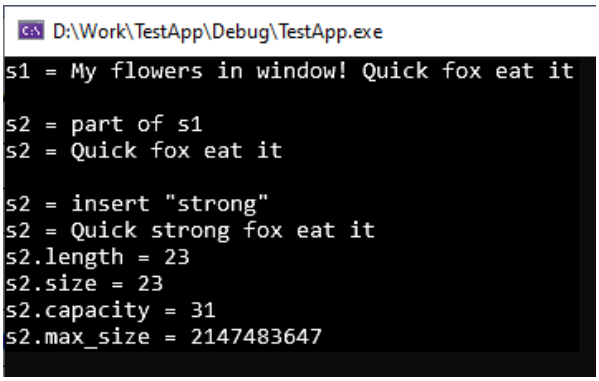


Рис.11.3. Результат тесту функцій assign, insert, length, size, capasity

Клас vector

Клас **vecto**r є шаблонним контейнерним класом, який може зберігати об’єкти будь-якого типу, впорядкувавши їх у вигляді елементів лінійного одновимірного масиву (вектора). Забезпечується можливість довільного доступу для будь-якого елементу вектора, у тому числі можна використовувати звичайний спосіб індексування оператором [ ].

Клас **vector** є динамічним масивом, розмір якого може змінюватися по мірі необхідності при додаванні або вилученні елементів.

Специфікація шаблонного класу **vector**

```
template <
    class Type,
    class Allocator = allocator <Type>
>
class vector
```

Параметри шаблону:

Type – тип елементів, які зберігаються у векторі

Allocator – об’єкт, який визначає виділення та звільнення пам’яті для вектора. Цей параметр є опціональним і значенням за замовчуванням є allocator <Type>

Клас **vector** має декілька різновидів конструкторів, які передбачають різні способи початкової ініціалізації вмісту створюваного об'єкту-вектора.

```
vector <int> v0;        //створюється порожній вектор v0
vector <int> v1(3);     //створюється вектор v1 з 3 нульовими елементами
vector <int> v2(5, 2); //створюється вектор v2 з 5 елементами = 2
vector <int> v3(v2);    //створюється вектор v3, який є копією вектора v2
vector <int> v4(10);

//--створюється вектор v5 з копіюванням частини вектора v4 --
vector <int> v5(v4.begin() + 1, v4.begin() + 7);
```

Функції-члени класу vector наведені у таблиці

Назва функції	Коментар
assign	Очищує вектор і копіює вказані елементи в порожній вектор.
at	Повертає посилання на елемент у вказаному місці у векторі.
back	Повертає посилання на останній елемент вектора.
begin	Повертає ітератор довільного доступу до першого елемента в контейнері.
capacity	Повертає кількість елементів, які може містити вектор без виділення додаткового обсягу пам’яті.
clear	Очищує (обнуляє) елементи вектора.
empty	Перевіряє, чи порожній контейнер.
end	Повертає ітератор довільного доступу, який вказує відразу за кінець вектора.
erase	Видаляє елемент або діапазон елементів у векторі з указаних позицій.
front	Повертає посилання на перший елемент у векторі.
get_allocator	Повертає об’єкт до класу розподілювача, який використовується вектором.
insert	Вставляє елемент або кілька елементів у вектор у вказаній позиції.
max_size	Повертає максимальну довжину вектора.
pop_back	Вилучає елемент у кінці вектора.
push_back	Дописує елемент у кінець вектора.
rbegin	Повертає ітератор до першого елемента в оберненому векторі.
rend	Повертає ітератор до кінця оберненого вектора.
reserve	Резервує мінімальну тривалість зберігання для векторного об’єкта.
resize	Визначає новий розмір для вектора.
size	Повертає кількість елементів у векторі.
swap	Взаємний обмін двох векторів.своїми елементами

Нижче приклад використання функцій push_back, empty, size, capacity

```
#include <iostream>
#include <vector>
using namespace std;

void show_vector_values(const char* name, vector<double>& v)
{
cout << "vector " << name << ".size = " << v.size() << endl;
cout << "vector " << name << ".capacity = " << v.capacity() << endl;
cout << "vector " << name;
if (v.empty())
    cout << " is empty";
else
   {
   cout << " elements = ";
```

```
   for (const auto& elem : v) cout << elem << ' ';
   }
   cout << "\n\n";
}

int main()
{
    vector <double> A;            //створюємо порожній вектор A
    show_vector_values("A", A);

    for (int i = 0; i <= 10; i++)
    {
        A.push_back(i);          //додаємо елемент у кінець вектора
        show_vector_values("A", A);
    }
    return 0;
}
```

Рис.11.4 Результат тесту функцій push_back, empty, size, capacity

Примітка. Для показу вмісту поточного вектора записано цикл показу усіх елементів цього вектора наступним чином

```
for (const auto& elem : v) cout << elem << ' ';
```

Це так званий цикл “for each”, який можна розглядати як можливий альтернативний варіант замість традиційного циклу індексів на кшталт

```
for (int i=0; i<(int)v.size(); i++) cout << v[i] << ' ';
```

Двовимірний масив на основі вектора

Оскільки у вектора у якості елементів можуть бути елементи деякого типу, зокрема й класового, то чому б не зберігати у векторі інші вектори. Двовимірний масив-матрицю можна реалізувати на основі класу вектора як вектор векторів.

```
vector<vector<Type>>
```

Розглянемо наступний приклад

```
void show_matrix(vector<vector<int>>& v)
{
    for (int y=0; y<(int)v.size(); y++)
    {
        cout << "Matrix[" << y << "][]" << " size = " << v[y].size();
        cout << " elements = ";
        for (int x=0; x<(int)v[y].size(); x++) cout << v[y][x] << ' ';
        cout << endl;
    }
    cout << endl;
}

//--test--
int main()
{
    vector<vector<int>> Matrix(4, vector<int>(5));

    show_matrix(Matrix);

    Matrix[1][0] = 7;
    Matrix[2][4] = 3;

    cout << "Matrix[1][0] = 7\n";
    cout << "Matrix[2][4] = 3\n";
    show_matrix(Matrix);


    return 0;
}
```

```
Microsoft Visual Studio Debug Console
Matrix[0][] size = 5 elements = 0 0 0 0 0
Matrix[1][] size = 5 elements = 0 0 0 0 0
Matrix[2][] size = 5 elements = 0 0 0 0 0
Matrix[3][] size = 5 elements = 0 0 0 0 0

Matrix[1][0] = 7
Matrix[2][4] = 3
Matrix[0][] size = 5 elements = 0 0 0 0 0
Matrix[1][] size = 5 elements = 7 0 0 0 0
Matrix[2][] size = 5 elements = 0 0 0 0 3
Matrix[3][] size = 5 elements = 0 0 0 0 0
```

Об’єкти власного класу в якості елементів вектора

Як правило, елементи, які вставляються у контейнер STL, можуть бути об’єктами практично будь-якого типу, якщо їх можна копіювати. Але, деякі додаткові обмеження, все ж таки, мають місце.

Вимоги для елементів контейнерів бібліотеки STL C++:

- Для впорядкованих асоціативних контейнерів необхідно визначити відкритий оператор порівняння. За замовчуванням це оператор operator<, проте зазвичай підтримуються навіть типи, які і не працюють з operator<.
- Для деяких операцій в контейнерах може також знадобитися відкритий конструктор за замовчуванням та відкритий оператор рівності. Наприклад, невпорядкованим асоціативним контейнерам потрібна підтримка порівняння на рівність та хешування.
- Деструктор не може викликати виключення.

Нижче наведено приклад запису у вектор об'єктів власного класу

```
class myclass       //оголошення класу користувача
{
private:
    int x, y;
public:
    myclass(void) { x = y = 0; }
    myclass(int vx, int vy)
    {
        x = vx;
        y = vy;
    }
    void method(void)
    {
        cout << "x = " << x << ", y = " << y << endl;
    }
};
```

```
//--test--
int main()
{
    myclass m1;
    myclass m2(10, 3);
    vector <myclass> A;

    A.push_back(m1);  //запис об'єктів у вектор
    A.push_back(m2);

    A[0].method();
    A[1].method();

    return 0;
}
```

Результат виконання тесту

Позитивні та негативні риси векторів

Як і всі реалізації динамічного масиву, вектор не використовує додаткових структур даних, дані розміщені в пам'яті поруч, за рахунок чого вони добре кешуються.

Вектор може швидко виділяти пам'ять, необхідну для зберігання конкретних даних. Це особливо корисно для зберігання даних у списках, довжина яких може бути невідома до створення списку, а видалення (за винятком, можливо, в кінці) необхідно рідко.

Як й інші контейнери STL, може містити примітивні типи даних, а також складні або ж визначені користувачем.

Вектор дозволяє швидкий довільний доступ до елементів; тобто на елемент вектора можна посилатись так само, як на елемент масиву (по індексу). Зв'язані списки, навпаки, не підтримують довільний доступ і арифметичні операції над вказівниками.

Видалення елемента з вектора або навіть очистка вектора абсолютно не обов'язково звільнить пам'ять, зв'язану з цим елементом. Це тому, що максимальний розмір вектора з моменту його створення є гарною оцінкою розміру для нового вектора.

Вектори є неефективними для вставки елементів в будь-які місця, крім кінця. Така операція має складність O(n) в порівнянні з O(1) для Зв'язаних списків. Це компенсується швидкістю доступу і швидкістю видалення. Доступ до довільного елемента вектора має складність O(1) в порівнянні з O(n) для Зв'язаного списку і O(log n) для дерева. Видалення має складність O(2) (перестановка і видалення).

Клас list

Клас **list** є шаблонним контейнерним класом для зберігання елементів, які вишукуються у лінійній послідовності і забезпечують швидкі операції вставки та вилучення елементів. Множина елементів зберігається як двоспрямований зв’язний список, який зберігає елементи однакового типу.

Специфікація шаблону для класу **list**

```
template <
   class Type,
   class Allocator=allocator<Type>
>
class list
```

Параметри шаблону:

Type – тип елементів, які зберігаються у списку

Allocator – об’єкт, який визначає виділення та звільнення пам’яті для списку. Цей параметр є опціональним і значенням за замовчуванням є **allocator <Type>**

Функції-члени класу list наведені у таблиці

Назва функції	Коментар
assign	Копіює набір елементів до цільового списку.
back	Повертає посилання на останній елемент списку.
begin	Повертає ітератор, що адресує перший елемент у списку.
clear	Стирає всі елементи списку.
empty	Перевіряє, чи порожній список.
end	Повертає ітератор, який звертається до місця, наступного за останнім елементом у списку.
erase	Видаляє елемент або діапазон елементів у списку з указаних позицій.
front	Повертає посилання на перший елемент у списку.
get_allocator	Повертає копію об’єкта розподілювача, який використовується для створення списку.
insert	Вставляє один або декілька елементів у список.
max_size	Повертає максимальну довжину списку.
merge	Вилучає елементи зі списку аргументів, вставляє їх у цільовий список і впорядковує новий об’єднаний набір елементів у порядку зростання або в іншому визначеному порядку.
pop_back	Видаляє елемент у кінці списку.
pop_front	Видаляє елемент на початку списку.
push_back	Додає елемент у кінець списку.
push_front	Додає елемент на початок списку.
rbegin	Повертає ітератор, що звертається до першого елемента в зворотному списку.
remove	Вилучає елементи зі списку, які відповідають вказаному значенню.
remove_if	Вилучає елементи, для яких задовольняється вказаний предикат.
rend	Повертає ітератор, який звертається до місця, наступного за останнім елементом у зворотному списку.
resize	Визначає новий розмір для списку.
reverse	Змінює порядок, у якому елементи зустрічаються в списку.
size	Повертає кількість елементів у списку.
sort	Сортує у порядку зростання або згідно іншого порядку.
splice	Вилучає вказані елементи і вставляє їх у цільовий список.
swap	Обмінює елементи двох списків.
unique	Видаляє зі списку суміжні повторювані елементи або суміжні елементи, які задовольняють інший двійковий предикат.

У наступному прикладі злиття та сортування списків

```
#include <iostream>
#include <list>
#include <string>
using namespace std;
```

```
//--показ вмісту списку--
void show_list_values(const char* name, list<string>& lst)
{
    list<string>::iterator p;
    cout << "list " << name;
    if (lst.empty()) cout << " is empty";
    else
    {
        cout << " size = " << lst.size() << '\n';
        cout << "elements: ";
        p = lst.begin();
        while (p != lst.end())
        {
            cout << *p << ' ';
            p++;
        }
    }
    cout << "\n\n";
}
```

```
int main()
{
    list<string> EUROPE, ASIA;      //створюємо два списка

    EUROPE.push_back("Latvia");     //заповнюємо список
    EUROPE.push_back("Ukraine");
    EUROPE.push_back("France");
    EUROPE.push_back("Italy");
    EUROPE.push_back("Liechtenstein");
    show_list_values("EUROPE", EUROPE);   //відображаємо список

    ASIA.push_back("Mongolia");     //заповнюємо інший список
    ASIA.push_back("China");
    ASIA.push_back("Laos");
    show_list_values("ASIA", ASIA);

    cout << "sort EUROPE list\n";
    EUROPE.sort();                  //сортування списку EUROPE
    show_list_values("EUROPE", EUROPE);

    cout << "sort ASIA list\n";
    ASIA.sort();                    //сортування списку ASIA
    show_list_values("ASIA", ASIA);

    cout << "merge EUROPE <- ASIA\n";
    EUROPE.merge(ASIA);             //злиття списків
    show_list_values("EUROPE", EUROPE);
    show_list_values("ASIA", ASIA);
    return 0;
}
```

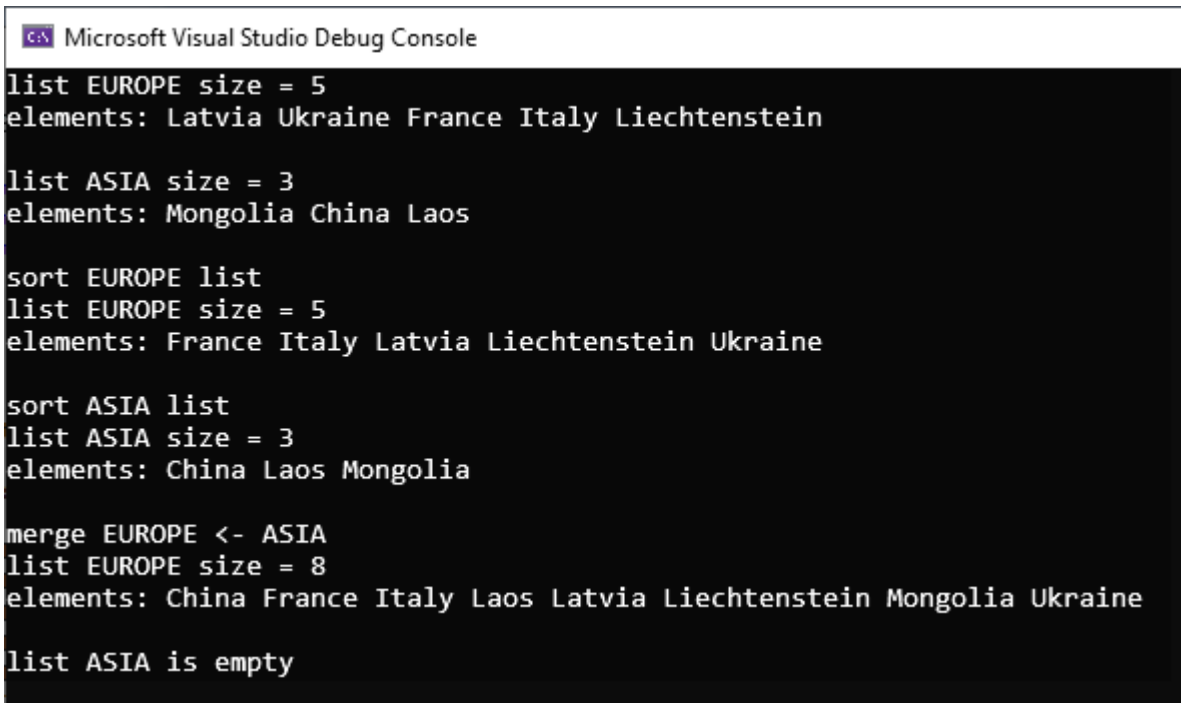


Рис. 11.5. Приклад виконання сортування та злиття списків

Примітка. Метод **list::merge** дає помилку, якщо об’єднувати невідсортовані списки.

Клас map

Цей контейнерний клас призначений для зберігання та читання пар значень. Кожна пара містить значення даних та ключа впорядкування. Значення ключа унікально й використовується для автоматичного сортування даних.

Значення даних можна змінити. Значення ключа є константою і його змінити не можна. Замість цього значення ключів, пов’язані зі старими елементами, необхідно знищити і вставити нові значення ключів для нових елементів.

Специфікація шаблонного класу **map**

```
template <
   class Key,
   class Type,
   class Traits = less<Key>,
   class Allocator=allocator<pair <const Key, Type> >
>
class map
```

Параметри шаблону наступні:

Key – тип даних ключа

Type – тип даних, які записуються у map

Traits – тип, який є об'єктом, який забезпечує порівняння двох елементів у якості ключів сортування, щоб визначити їхній порядок. Цей параметр є необов'язковим і за замовчуванням використовується бінарний предикат **less<``Key``>.**

Allocator – тип, який є об'єктом розподільника, він зберігається та інкапсулює відомості про спосіб виділення та звільненні пам'яті. Цей параметр є необов'язковим і за замовчуванням використовується **allocator<pair <const Key, Type > >.**

Основні характеристики класу map

1. **Шаблонний клас**, типи елементів та ключі, функція порівняння та алокатор визначаються як параметри шаблону

2. **Асоціативний контейнер перемінного розміру**, який реалізує ефективне знаходження потрібних значень, яке базується на співставленні зі значенням ключа. Кожний елемент має унікальне неповторне значення ключа

3. **Сортований контейнер пар**, елементи впорядковуються відповідно значенням ключів у контейнері у відповідності визначеної функції порівняння

4. **Реверсивний**, тому що реалізує двонаправлені ітератори для доступу до елементів контейнера

Розглянемо приклад використання класу map. Уявимо собі телефонну книгу з переліком абонентів у вигляді пар (номер телефону, ім'я абонента). Номер телефону – неповторне значення, **його можна використати** як **ключ, зокрема,** для пошуку імен абонентів. Для пошуку скористаємося функцією find класу map. Ця функція повертає ітератор, який вказує на елемент із шуканим значенням ключа.

```
#include <iostream>
#include <map>
#include <string>
using namespace std;

void show_map(map <string, string> &m)    //показ усіх елементів map
{
map <string, string>::iterator p;
cout  << "Mapped elements:\n keys\t\tvalues\n";
for (p = m.begin(); p != m.end(); p++ )
    cout << p->first << "\t" << p->second << endl;
}
```

```
//--тест--
int main(void)
{
map<string, string> directory;
directory.insert(pair<string, string>("555-4533", "Tom"));
directory.insert(pair<string, string>("555-9678", "Victor"));
directory.insert(pair<string, string>("555-0823", "Pedro"));
directory.insert(pair<string, string>("555-8195", "John")) ;
directory.insert(pair<string, string>("777-0849", "Pedro"));
directory.insert(pair<string, string>("555-0809", "Elvis"));
show_map(directory);

cout << "Enter telephone: ";
string s;
cin >> s;
map<string, string>::iterator p;
p = directory.find(s);
if(p != directory.end())
     cout << "Abonent name: " << p->second << '\n';
else cout << "This telephone is not present.\n";

return 0;
}
```

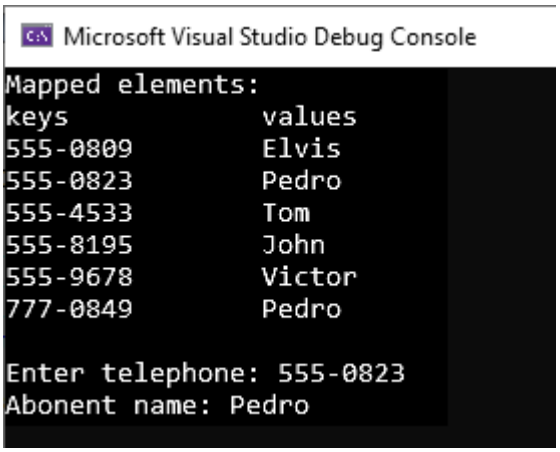


Рис. 11.6. Тест пошуку по ключу – номеру телефону

Необхідно відзначити, що такий варіант організації списку абонентів є лише простою ілюстрацією об'єкта классу map як сховища пар з можливістю пошуку по ключовому значення. Вочевидь, для реалізації сховища з розвинутими пожливостями пошуку потрібно дещо більше.

Шаблони алгоритмів як елементи стандартної бібліотеки C++

Алгоритми, які належать до стандартної бібліотеки шаблонів C++, містяться у заголовку **<algorithm>**. Такі алгоритми виконують операції над вмістом контейнерів. Існують алгоритми для ініціалізації, сортування, пошуку і зміни вмісту контейнерів.

Алгоритми оформлені у вигляді шаблонних функцій бібліотеки STL. Такі алгоритми універсальні, тому що можуть працювати із різноманітними структурами даних. Такими структурами даних можуть бути не тільки класи контейнерів STL, такі як vector і list, але також і визначені програмою структури даних і масиви елементів, які задовольняють вимогам відповідного алгоритму.

Алгоритми досягають такого рівня універсальності завдяки доступу до елементів контейнера опосередковано за допомогою ітераторів. Вони відіграють роль вказівників на елементи контейнерів.

Числові алгоритми STL, які надаються для числової обробки, містяться у окремому заголовку **<numeric>.**

Функціональні об’єкти записані у заголовку **<functional>**. Функціональні об’єкти, які повертають логічні значення, звуться предикатами.

Алгоритми STL розширюють дії, які підтримуються операціями і функціями-членами кожного контейнера STL. В позначеннях назв алгоритмів можуть використовуються суфікси, які вказують деякі різновиди алгоритмів.

Суфікс **_if** вказує, що алгоритм використовує деякий функціональний об’єкт для визначення потрібної умови відбору елементів контейнерів. Так, наприклад, алгоритм **find_if** виконує пошук елементів, значення яких задовільняють критерію, визначеному функціональним об’єктом.

Суфікс **_copy** означає, що цей алгоритм не тільки маніпулює значеннями елементів, но також копіює змінені значення у діапазон призначення. Алгоритм reverse змінює порядок елементів у діапазоні на зворотний, а алгоритм **reverse_copy** також копіює результат у діапазон призначення.

Повний перелік алгоритмів стандартної бібліотеки шаблонів наведено у документації, зокрема, для відповідних реалізацій C++, наприклад, Microdoft Visual C++

Розглянемо деякі з бібліотечних алгоритмів.

Алгоритм count_if

Обчислює кількість елементів, які задовільняють умові. Цей алгоритм визначено у вигляді шаблонної функції з трьома параметрами:

```
template<class InputIterator, class Predicate>
   typename iterator_traits<InputIterator>::difference_type count_if(
      InputIterator _First,
      InputIterator _Last,
      Predicate _Pred
   );
```

_First – вхідний ітератор, який вказує позицію першого елементу у діапазоні пошуку

_Last – вхідний ітератор, який вказує наступну позицію за останнім елементом у діапазоні пошуку

_Pred – функціональний об'єкт предикат, який визначає умову, якій повинен задовільняти елемент, щоб бути врахованим. Функцію-предикат визначає користувач, у неї повинен бути **один аргумент** і повертається значення типу **bool** – **true** або **false**

Примітка. Шаблонна функція **count_if** є узагальненням алгоритму **count** шляхом заміни предикату "дорівнює" на будь-який предикат користувача.

У якості прикладу використання **count_if** розглянемо підрахунок кількості певних елементів вектора.

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;

string sval;

void show_vector_values(const char *name,
                        vector<string> &v)
{
cout << "vector " << name << endl;
cout << v.size() << " elements: ";
for (int i=0; i<(int)v.size(); i++)
    cout << v[i] << ' ';
cout << "\n\n";
}

//функція, яка перевірятиме умову для count_if
bool equalname(string value)
{
return (value == sval);
}
```

```
//--тест--
int main()
{
vector <string> S;
S.push_back("Paul");
S.push_back("Victor");
S.push_back("Jerry");
S.push_back("Pedro");
S.push_back("John");
S.push_back("Pedro");
S.push_back("Jerry");
S.push_back("Jerry");
show_vector_values("S", S);

cout << "Enter name: ";
cin >> sval;

vector<string>::iterator::difference_type result;
result = count_if(S.begin(), S.end(), equalname);

cout << "The number of elements \"" << sval
     << "\" is: " << result << ".\n";

return 0;
}
```

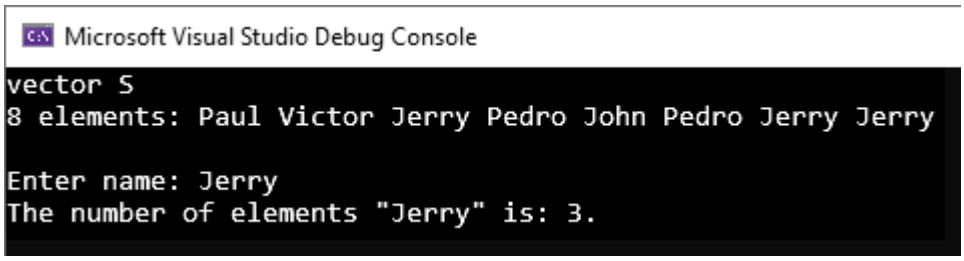


Рис. 11.7. Приклад роботи тесту для **count_if**

Алгоритм find_if

Знаходить позицію першого елементу, який задовільняє деякій умові. Алгоритм find_if є шаблонною функцією, яка оголошена подібно до розглянутого вище алгоритму count_if

```
template<class InputIterator, class Predicate>
   InputIterator find_if(
      InputIterator _First,
      InputIterator _Last,
      Predicate _Pred
   );
```

Примітка. Ця шаблонна функція є узагальненням алгоритму **find.**

Розглянемо приклад використання алгоритму **find_if** для пошуку у контейнері **map** не по ключу, а по значенню.

```
#include <iostream>
#include <string>
#include <map>
#include <algorithm>
using namespace std;

void show_map(map <string, string>& m)   //показ усіх елементів map
{
    map <string, string>::iterator p;
    cout << "Mapped elements:\n keys\t\tvalues\n";
    for (p = m.begin(); p != m.end(); p++)
        cout << p->first << "\t" << p->second << endl;
}

pair<string, string> tmp;
```

```
//функціональний об'єкт – предикат умови пошуку
bool equalname(pair<string, string> value)
{
    if (tmp.second == value.second) return true;
    return false;
}
```

```
int main(void)
{
    map<string, string> Tel;
    Tel["555-4533"] = "Tom";
    Tel["555-9678"] = "Victor";
    Tel["555-8195"] = "John";
    Tel["555-0809"] = "Elvis";
    Tel["555-0823"] = "Pedro";
    Tel["777-0825"] = "Jerry";
    Tel["777-0826"] = "Pedro";
    Tel["444-0726"] = "Pedro";
    Tel["555-1342"] = "Sam";
    show_map(Tel);

    cout << "Enter abonent name: ";
    cin >> tmp.second;

    map<string, string>::iterator result;
    result = find_if(Tel.begin(), Tel.end(), equalname);

    cout << "Found: ";
    if (result == Tel.end())
        cout << "not found\n";
    else cout << result->first << " " << result->second << endl;
    return 0;
}
```

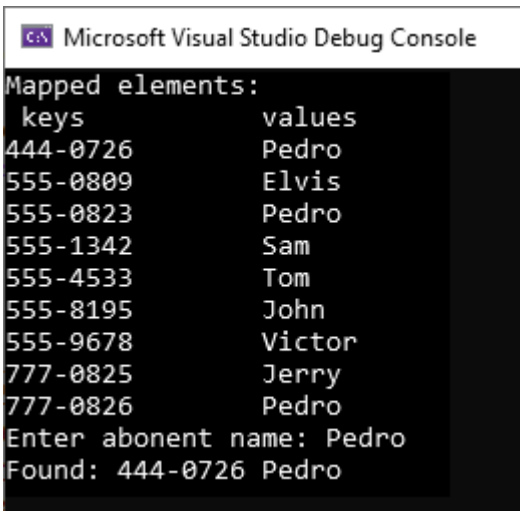


Рис. 11.8. Приклад роботи тесту пошуку **find_if**

Як бачимо, можливо організувати пошук елементів map не по ключу, а по значенню, але наведений приклад не є досконалим – виводить лише одну першу пару з декількох можливих.

Алгоритм for_each

Застосовує функцію користувача до усіх елементів контейнера у визначеному робочому діапазоні позицій і повертає значення функції

```
template<class InputIterator, class Function>
   Function for_each(
      InputIterator _First,
      InputIterator _Last,
      Function _Func
      );
```

Параметри шаблону алгоритму:

_First – вхідний ітератор, який вказує позицію першого елемента у робочому діапазоні

_Last – вхідний ітератор, який вказує наступну позицію за останнім елементом у робочому діапазоні

_Func – визначений користувачем функціональний об’єкт, який застосовується до кожного елементу у робочому діапазоні

У якості прикладу розглянемо використання алгоритму **for_each** для подальшого вдосконалення пошуку елементів сховища **map** з попереднього прикладу. Нашою метою є забезпечення показу списку елементів, які задовільняють умові пошуку.

```
#include <iostream>
#include <string>
#include <map>
#include <vector>
#include <algorithm>
using namespace std;

. . . //show_map(...) - ця функція така сама, як і у попередньому прикладі
pair<string, string> tmp;
vector<string> Result;
```

```
//функціональний об'єкт для for_each пошуку
void CmpAndStore(pair<string, string> value)
{
    if (tmp.second == value.second)
        Result.push_back(value.first);
}
```

```
//функціональний об'єкт для for_each показу списку елементів, що знайдено
void Show(string value)
{
    cout << value << "\n";
}
```

```
int main(void)
{
    map<string, string> Tel;
    Tel["555-4533"] = "Tom";
    Tel["555-9678"] = "Victor";
    Tel["555-8195"] = "John";
    Tel["555-0809"] = "Elvis";
    Tel["555-0823"] = "Pedro";
    Tel["777-0825"] = "Jerry";
    Tel["777-0826"] = "Pedro";
    Tel["444-0726"] = "Pedro";
    Tel["555-1342"] = "Sam";
    show_map(Tel);
    cout << "Enter abonent name: ";
    cin >> tmp.second;
    for_each(Tel.begin(), Tel.end(), CmpAndStore);        //пошук
    for_each(Result.begin(), Result.end(), Show);         //показ
    return 0;
}
```

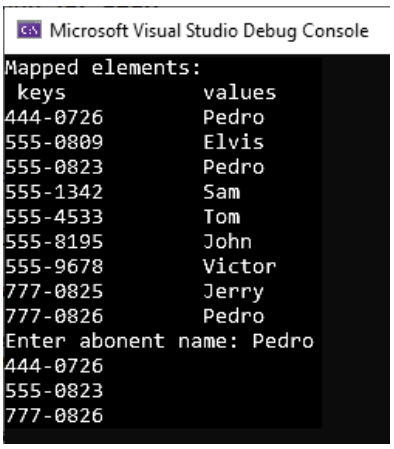


Рис. 11.9. Приклад роботи тесту алгоритму **for_each**

Контрольні запитання

1. Які складові стандартної бібліотеки C++?
2. Чи можливо в програмі на C++ використати бібліотеку C?
3. Чим відрізняється клас complex від pair?
4. Які типи символів підтримує клас string?
5. Що таке конкатенація?
6. Які відмінності size від capacity в класі vector?
7. Як запрограмувати двовимірний масив на основі класу vector?
8. Які недоліки у контейнерів типу vector?
9. Чим контейнер list відрізняється від vector?
10. Що таке асоціативний контейнер?
11. Навіщо у контейнерах map зберігаються пари?
12. Що означає суфікс _if?
13. Що таке ітератор? Наведіть приклади
14. Що таке функціональний об'єкт?
15. Чим відрізняється алгоритм count_if від find_if?
16. Що робить алгоритм for_each?

Лекція 12

Тема 5.1. Callback-функції

Формулювання проблеми

Нехай **модуль 1** залежить від **модуля 2**. А модуль 2 не залежіть від модуля 1. Іншими словами, має місце така *#include*-ієрархія:

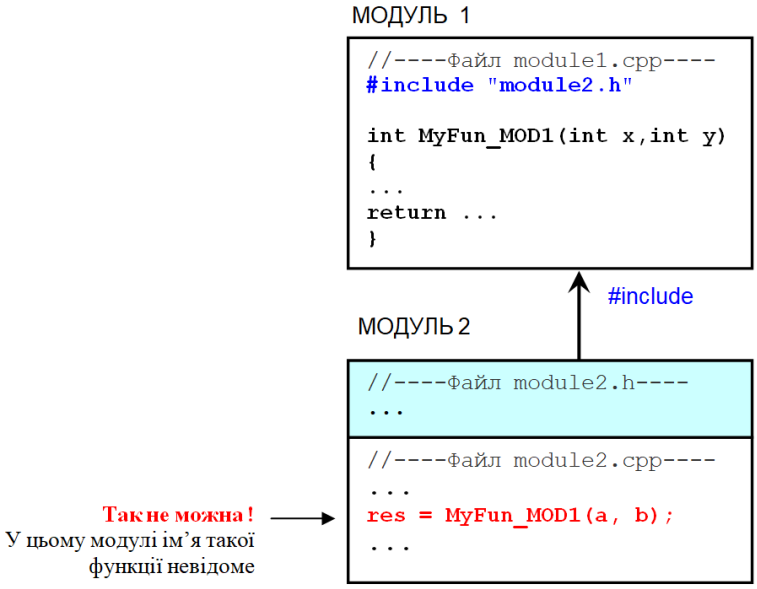


Рис. 12.1. Ілюстрація проблеми

Не можна у модулі 2 викликати функції модуля 1, оскільки про ці функції модуль 2 нічого не знає. Але, якщо, все ж таки, дуже потрібно використати у модулі 2 якусь функцію з модуля 1, то як це зробити?

Тривіальне рішення. Можна записати оголошення потрібної функцію у файл “module1.h” і у module2.cpp записати #include “module1.h”

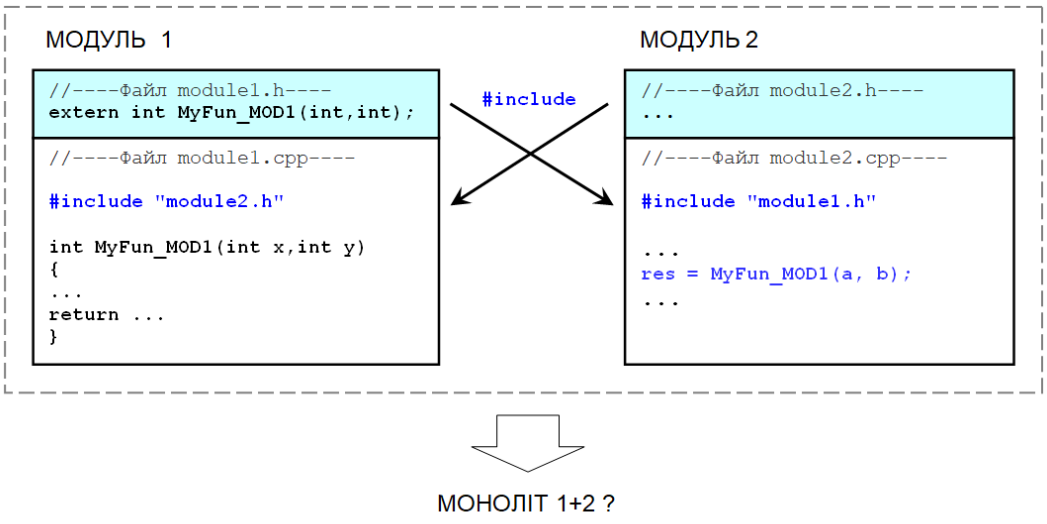


Рис. 12.2. Тривіальне рішення

Недолік такого рішення – перехресні *#include*-зв'язки модулів. Перехресні зв’язки модулів зазвичай дуже шкідливі і є свідченням похибок проектування архітектури. Фактично, замість модульності вже вимальовується моноліт з коду двух модулів – залежності модулів стають занадто сильними. У модулі 2 явно записується ім'я функції з модуля 1.

А якщо взагалі неможливо вписати в модуль 2 явний виклик (по імені) якоїсь функції з модуля 1? Уявіть, що модуль 2 є частиною бібліотеки із закритим кодом, яку написав хтось інший. Тоді так просто проблема не вирішується.

Суть технології Callback

Callback-функція – це функція, посилання (вказівник) на яку передається комусь іншому, щоб потім він її викликав. Наприклад, уявимо собі, що потрібно організувати виклик callback-функцій у тілі деякої функції Fun_MOD2. Для передачі цій функції вказівника на callback-функцію можна зробити параметр функціонального типу, наприклад

```
void Fun_MOD2(.., long fun(int, int), ...);
```

У даному прикладі функціональний параметр (fun) оголошено як будь-яка функція, яка має два параметри типу int і повертає значення типу long.

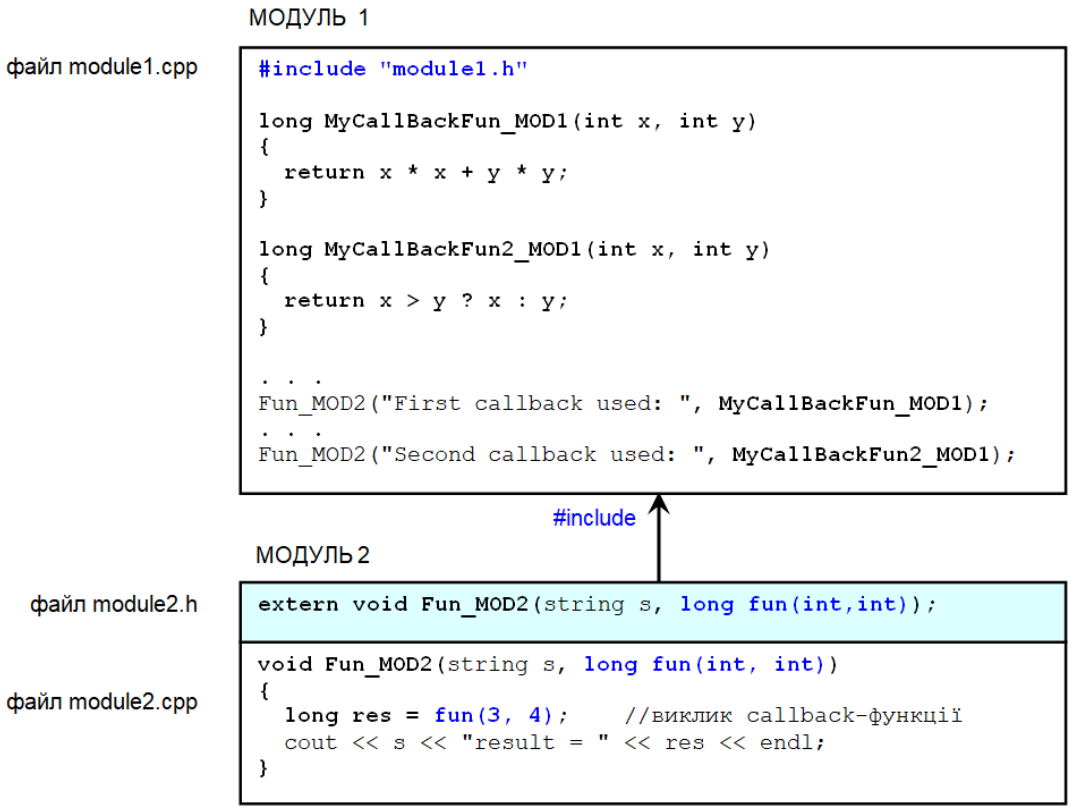


Рис. 12.3. Організація виклику у модулі 2 функцій модуля 1

В ході виконання даного прикладу на консолі буде виведено

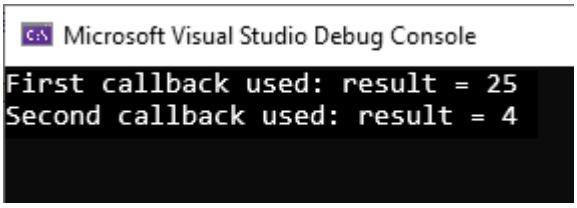


Рис. 12.4. Результати роботи Fun_MOD2 для двох варіантів callback-функції

Якщо при виклику певної функції X() ми передаємо їй у якості параметру деяку функцію (fun), то можна стверджувати, що у такий спосіб можна впливати на поведінку функції X() і отримувати різні результати.

У якості прикладу можна навести функцію **qsort** стандартної бібліотеки **stdlib** мови C. Ця функція містить декілька параметрів, один з яких – вказівник на callback-функцію порівняння двох значень

```
void qsort(,,, int fun_comp(const void*, const void*));
```

У наступному прикладі визначено callback-функцію порівняння для сортування масивів елементів типу long за зростанням значень

```
int FunCompare(const void* arg1, const void* arg2)
{
  long* p1 = (long *)arg1;
  long* p2 = (long *)arg2;
  if (*p1 == *p2) return 0;
  if (*p1 > *p2) return 1;
  return -1;
}

long MyArray[1000000];
qsort(MyArray, 1000000, sizeof(long), FunCompare);
```

А якщо змінити callback-функцію порівняння, то можна виконати сортування, наприклад масивів елементів типу int за зменшенням значень

```
int FunCompare2(const void* arg1, const void* arg2)
{
  int* p1 = (int *)arg1;
  int* p2 = (int *)arg2;
  if (*p1 == *p2) return 0;
  if (*p1 > *p2) return -1;
  return 1;
}

int MyArray[1000000];
qsort(MyArray, 1000000, sizeof(int), FunCompare2);
```

Приклади використання callback-функцій в Windows API

Техніка callback-функцій широко використовується при організації функціонування програмних додатків для Windows. Найбільш помітним є callback для реалізації графічного інтерфейсу користувача та взаємодією ОС із програмним додатком. Для цього використовується система повідомлень – *Windows messages*. Що таке повідомлення? Це виклик з боку ОС Windows callback-функції вікна програмного додатку. А для цього ОС має знати адресу функції вікна.

Організація callback для головного вікна програми. Адресу callback-функції власного головного вікна програмний додаток записує у структуру даних для реєстрації класу цього вікна, яка потрібна для його створення.

```
//---головна функція – точка входу в програму---
int APIENTRY wWinMain(...)
{
   . . .
   WNDCLASSEXW wcex;     //структура опису класу вікна
   wcex.lpfnWndProc = WndProc;    //адреса функції вікна
   . . .                          //інші поля структури
   RegisterClassExW(&wcex);       //реєстрація класу вікна
   . . .   //створення вікна та підтримка прийому повідомлень
}
```

```
//---callback-функція головного вікна програми---
LRESULT CALLBACK WndProc(...)
{
   switch (message)
   {
      . . .   //обробка повідомлень
   }
   return 0;
}
```

Після реєстрації і створення вікна Windows починає надсилати програмному додатку повідомлення – викликає функцію WndProc головного вікна у відповідь на дії користувача та інші події.

Організація callback для вікон діалогу. Для створення модального вікна діалогу *(modal dialog)* призначена функція **DialogBox**, одним з аргументів якої є адреса callBack-функції цього вікна

```
DialogBox(hInst, MAKEINTRESOURCE(...), hWndParent, DlgProc);
```

Немодальні вікна діалогу *(modeless dialog)* створюються викликом функції **CreateDialog,** одним з аргументів якої є адреса callBack-функції цього вікна

```
CreateDialog(hInst, MAKEINTRESOURCE(...), hWndParent, DlgProc);
```

Програмний код callback-функцій для наведених вище різновидів вікон виконує обробку повідомлень майже однаково, хоча і з невеликими відмінностями для модальних та немодальних вікон діалогу та головного вікна.

Інверсія керування та інверсія залежностей

Якщо модуль 1 використовує щось з модуля 2, то значить, модуль 1 залежіть від модуля 2. На діаграмах UML така залежність показується стрілкою, спрямованою від модуля 1 до модуля 2 (навпаки стрілкам #include-ієрархії). Модуль 1 є клієнтом модуля 2. Якщо у модулі 1 робиться виклик функції модуля 2, то лінію залежності можна позначити «call»

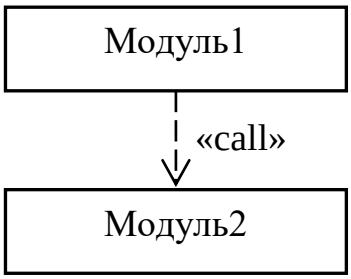


Рис. 12.5. Показ залежності модулів в нотації UML

А тепер, якщо в модулі 1 визначена callback-функція, адреса якої передається в модуль 2 для виклику у модулі 2, то це можна відобразити наступним чином

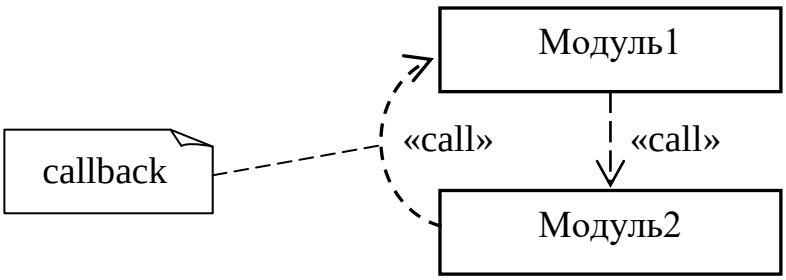


Рис. 12.6. Ілюстрація явищ інверсії керування, інверсії залежностей

У callback-техніці втілюється так званий *голівудський принцип:* «не дзвоніть нам – ми подзвонимо вам».

Реалізація callBack-техніки віртуальними функціями класів

Оголосимо базовий абстрактний клас із потрібними інтерфейсними методами, які планується використати у якості callback-функцій

```
//---файл BaseBlass.h---
class BaseClass
{
public:
   virtual void Method1(void) = 0;
   virtual void Method2(void) = 0;
};
```

Конкретні callback-функції будуть реалізовані у класі-спадкоємці в модулі верхнього рівня, наприклад, у головному файлі. Тут також міститься виклик якоїсь функції з модуля, якій у якості параметра передаємо адресу об'єкта. У якості такої функції може бути як окрема глобальна функція (SomeFunc), так і функція-член деякого іншого класу

```
//---файл Main.cpp---
#include "BaseClass.h"
#include "Module.h"

class DerivedClass : public BaseClass
{
public:
   void Method1(void)            //перша CallBack-функція
   {
   . . .
   }

   void Method2(void)            //друга CallBack-функція
   {
   . . .
   }
};

SomeFunc(new DerivedClass);      //передаємо адресу об'єкта
```

У модулі нижчого рівня виклик callback-функцій модуля вищого рівня запрограмуємо як поліморфний виклик методів похідного класу

```
//---файл Module.cpp---
#include "BaseClass.h"

void SomeFunc(BaseClass *pbc)     //це може бути окрема функція,
                                  //або функція-член якогось класу
{
. . .
pbc->Method1();
```

```
. . .

pbc->Method2();
. . .
}
```

Оскільки клас може інкапсулювати декілька функцій-членів, то передаючі вказівник на об'єкт цього класу, фактично ми у такий спосіб організовуємо декілька callback-функцій.

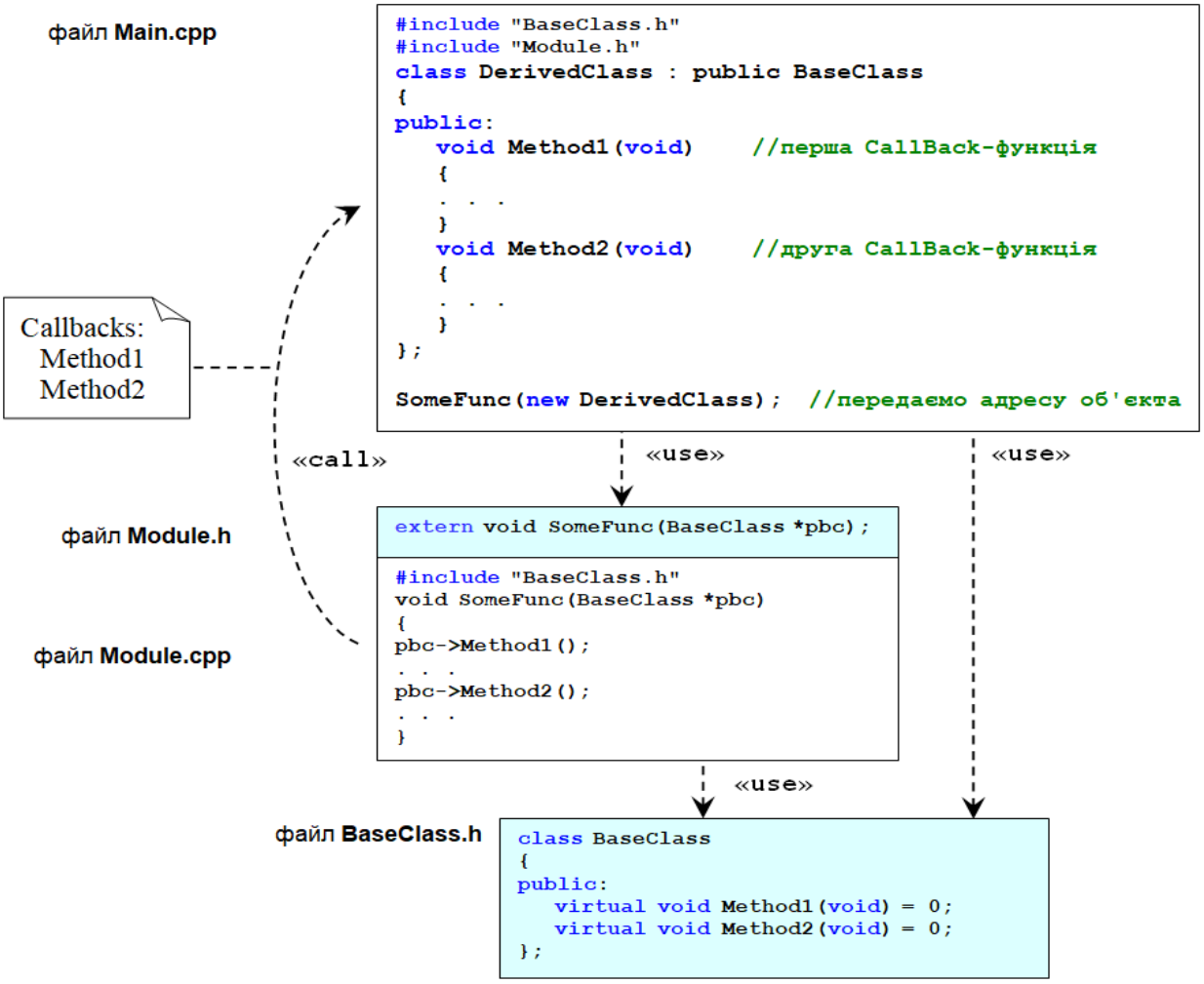


Рис. 12.7. Реалізація callback-функцій віртуальними функціями класу

Коментуючи наведений вище приклад, варто зазначити, що у модулі Module.cpp нічого не відомо про похідний клас DerivedClass, але завдяки поліморфізму у тілі функції SomeFunc по вказівнику на BaseClass коректно викликаються методи Method1 і Method2 класу DerivedClass модуля вищого рівня. І ця техніка спрацьовує як при передаванні вказівників на клас через параметр окремих глобальних функцій (які дозволені у C++) так і при передаванні таких вказівників функціям-членам інших класів.

Також можна помітити, що реалізація техніки callback через вказівник на клас дещо відрізняється від callback-функцій у мові С, адреси яких можуть безпосередньо передаватися у потрібне місце, якщо усі типи є стандартними (вбудованими). Оскільки клас є типом користувача, то його потрібно оголосити і для використання цього типу необхідnj знати його оголошення. Для цього використовується #include файлу BaseClass.h, у якому міститься оголошення базового класу. Зручно робити такий базовий клас абстрактним з усіма чисто віртуальними функціями – тоді не треба жодного визначення цих функції у базовому класі і відкласти це визначення на похідні класи. Ця ідея знайшла своє втілення у класах-інтерфейсах, які ми розглянемо нижче.

Підсумки

Основна сутність техніки callback – це організація передачі деякій функції (методу) або модулю посилання на програмнй код callback-функції, причому цей код може бути замінений на іншій. Така ідея щодо callback отримала подальший розвиток у таких поняттях, як *інверсія керування* (Inversion of Control), *інверсія залежностей* (Dependency Inversion) та *впровадження залежностей* (Dependency Injection).

Виклик callback-функцій по значенню вказівника (в С та С++) широко використовується для побудови бібліотек загального використання, зокрема, це використано у Windows API. В тих мовах програмування, де немає вказівників (але все одно є поліморфізм), ідея callback також використовується, проте реалізовується трохи по-іншому.

Техніка callback використовується для покращення структурованості програм. Це один з методів побудови множини слабо зв’язаних модулів, що дозволяє перемагати зростаючу складність програм. Безпосереднім втіленням ідеї callback можуть вважатися такі патерни об'єктно-орієнтованого дизайну, як Observer, Listener, Strategy та інші.

Контрольні запитання

1. Що таке і навіщо потрібен callback?
2. Що треба зробити, щоб організувати callback?
3. Як використовується callback в функції сортування qsort стандартної бібліотеки С?
4. Як використовується callback в графічному інтерфейсі користувача?
5. Як впливає callback на залежності модулів?
6. Що може означати правило «не дзвоніть нам – ми подзвонимо вам» при застосуванні його у програмуванні?

Тема 5.2. Класи-інтерфейси

Поняття класів-інтерфейсів

Класи-інтерфейси є відомим та популярним засобом проектування гнучких архітектурних рішень програмних засобів шляхом роз'єднання інтерфейсів та іхньої реалізації. Можна відзначити цінність підходу використання класів-інтерфейсів при командній розробці складних проектів з множини слабо зв'язаних класів та модулів.

Оголошення класу-інтерфейсу. Клас-інтерфейс оголошується у вигляді базового абстрактного класу – він містить тільки чисто віртуальні функції, без даних-членів (перемінних) і без конструкторів та деструктора:

```
class Іназва
{
public:
   virtual тип1 Func1(параметри) = 0;
   virtual тип2 Func2(параметри) = 0;
   ...
   virtual типN FuncN(параметри) = 0;
};
```

Примітка. Далі ми трохи вдосконалимо формат коректного оголошення класу-інтерфейсу.

Існує традиція назви класів-інтерфейсів розпочинати з букви I.

Приклад класу-інтерфейсу:

```
class IOpenable
{
public:
   virtual int Open(void) = 0;
   virtual void Close(void) = 0;
};
```

Таким чином, клас-інтерфейс записується тільки у вигляді оголошення.

Не може бути створений об'єкт-екземпляр даного класу.

Не можуть бути визначені методи у цьому класі, оскільки заборонено визначати будь-яку чисто віртуальну функцію даного класу. Визначення функцій інтерфейсу покладається на похідні класи. Прийнято казати, що похідний клас *реалізує* інтерфейс.

Згідно канонів ООП класи-інтерфейси мають такі загальні риси:

- Клас-інтерфейс, або просто інтерфейс – це абстракція, яка описує деякий набір функцій
- Може мати тільки public функції-члени (методи). Для жодної з функцій класу-інтерфейсу не може бути у цьому класі визначена реалізація за замовчуванням. Не може містити дані-члени (поля), конструктор та деструктор
- Інтерфейс може розширювати інший інтерфейс
- Клас який успадковує (реалізує) клас-інтерфейс для створення об'єктів, повинен реалізовувати усі функції інтерфейсу
- Класи можуть успадковувати (реалізовувати) декілька інтерфейсів

Необхідно зазначити, що даний перелік основних властивостей класів-інтерфейсів є ідеалізованим. Далі розглянемо специфіку програмування таких класів мовою C++.

Приклади на C++ з використанням класів-інтерфейсів

Розглянемо приклад оголошення класу-інтерфейсу, який містить три методи

```
class IExecutable
{
public:
    virtual int start() = 0;
    virtual void run() = 0;
    virtual void end() = 0;
};
```

Для класу-інтерфейсу **IExecutable** зробимо дві реалізації – класи Work та Game

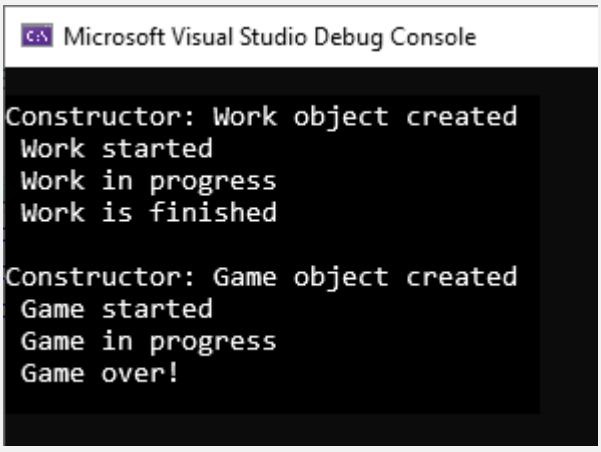
```
class Work : public IExecutable
{
public:
    Work() {
        std::cout << "\nConstructor: Work object created\n";
    }
    int start() {
        std::cout << " Work started\n";
        return 1;
    }
    void run() {
        std::cout << " Work in progress\n";
    }
    void end() {
        std::cout << " Work is finished\n";
    }
```

```
    ~Work() {
        std::cout << "Destructor: Work object removed\n";
    }
};
```

```
class Game : public IExecutable
{
public:
    Game() {
        std::cout << "\nConstructor: Game object created\n";
    }
    int start() {
        std::cout << " Game started\n";
        return 1;
    }
    void run() {
        std::cout << " Game in progress\n";
    }
    void end() {
        std::cout << " Game over!\n";
    }
    ~Game() {
        std::cout << "Destructor: Game object removed\n";
    }
};
```

В тестовій програмі в функції MyProcess на рівні абстракції описується послідовність поліморфних викликів методів інтерфейсу IExecutable. А у функції main створюються екземпляри динамічних об’єктів двох класів.

```
//--test--
void MyProcess(IExecutable* p)
{
    if (p->start())
    {
        p->run();
        p->end();
    }
    delete p;
}

int main(void)
{
    MyProcess(new Work);
    MyProcess(new Game);
    return 0;
}
```



Як видно на консолі, виконання даної тестової програми відбувається некоректно – не спрацьовують деструктори класів Work та Game. Чому це?

Тепер варто згадати положення щодо реалізації поліморфізму, викладені раніше у Лекції 3, зокрема, щодо деструкторів.

Для коректної реалізації поліморфізму необхідно визначити у базовому класі віртуальний деструктор. Але ж за канонічним визначенням інтерфейс не повинен його мати? Тим не менше, давайте спробуємо його ввести у класс-інтерфейс IExecutable.

```
class IExecutable
{
public:
    virtual ~IExecutable() { };
    virtual int start() = 0;
    virtual void run() = 0;
    virtual void end() = 0;
};
```

Решту коду змінювати не потрібно.

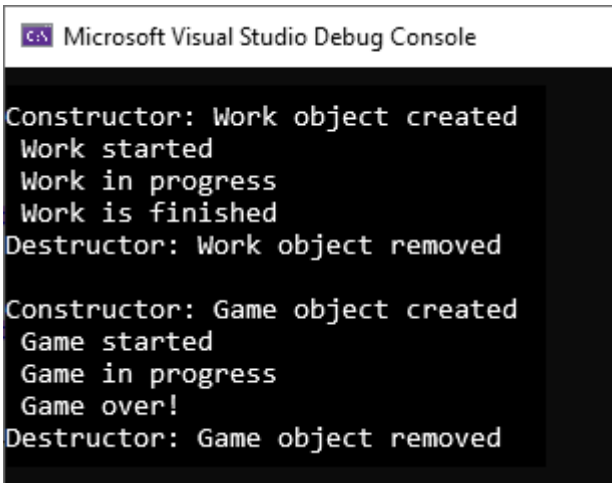


Рис. 12.8. Тепер усе працює нормально

Можна стверджувати, що в мові C++ класи-інтерфейси (або просто інтерфейси) в «чистому» канонічному вигляді реалізувати не можна. Впродовж численних ітерацій розширення синтаксису мови C++ у багатьох стандартах так і не була втілена «чиста» конструкція інтерфейсу. Вважається, що поняття абстрактного класу разом із можливостями множинного успадкування в C++ повністю заперечує необхідність існування інтерфейсу як окремого синтаксичного елементу мови. Проте у деяких реалізаціях мови C++ можна зустріти спроби ввести слово «інтерфейс». Так, зокрема, у Microsoft Visual C++ є навіть два ключових слова: **interface** та **__interface**. Перше з них призначене для .Net Framework, а друге – для native C++.

Спробуємо розібратися з цим докладніше шляхом створення відповідної тестової програми у середовищі Microsoft Visual Studio C++. Взявши за основу вихідний текст попереднього прикладу, використаємо для оголошення классу інтерфейсу ключове слово **__interface**.

```
__interface IExecutable
{
    int start();
    void run();
    void end();
};
```

Завдяки ключовому слову **__interface** оголошення інтерфейсу стає «канонічним» – не треба тепер записувати слова **public**, **virtual** – усі функції оголошеного інтерфейсу автоматично є чисто віртуальними публічними.

Після таких змін в оголошенні **IExecutable** тестова програма успішно компілюється і викликається на виконання. Але, така тестова програма не викликає деструктори класів-реалізаторів (похідних класів) Work та Game.

Спробуємо вирішити цю проблему введенням віртуального деструктора в оголошення класу-інтерфейсу, як ми робили до того

```
__interface IExecutable
{
    virtual ~IExecutable() { };
    int start();
    void run();
    void end();
};
```

Але Microsoft Visual Studio C++ видає помилку компіляції:

```
Build started...
1>------ Build started: Project: TestApp, Configuration: Debug Win32 ------
1>TestApp.cpp
1>D:\Work\TestApp\TestApp.cpp(8,30): error C2849: 'IExecutable': an interface cannot have a destructor
1>Done building project "TestApp.vcxproj" -- FAILED.
========== Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped ==========
```

Компілятор нагадує нам, що інтерфейс не може мати деструктор. Студентам пропонується самостійно докладніше розібратися з можливостями використання ключового слова **interface** у різних реалізаціях C++.

Таким чином можна констатувати, що класи-інтерфейси в C++ потрібно моделювати абстрактним класом специфічного вигляду. Тут варто сказати, що поняття «interface» означає дещо вищій рівень абстракції, ніж просто абстрактний класс. Так, зокрема, у класі-інтерфейсі усі функції мають бути чисто віртуальними та public.

На відміну цього у деяких інших мовах програмування, наприклад, Java є окреме ключове слово **interface**, яке дозволяє оголошувати справжні «канонічні» інтерфейси. В Java на відміну від успадкування класів (**extends**), успадкування (реалізація) інтерфейсів записується словом **implements**. Крім того, в мові Java множинне успадкування дозволено лише для інтерфейсів [10]. Докладніше про використання інтерфейсів в ООП у наступних лекціях.

Контрольні запитання

1. Що містить клас-інтерфейс?
2. Чого не повинно бути у класі-інтерфейсі згідно з канонами ООП?
3. Як класи-інтерфейси можна реалізувати на C++?
4. Чи може бути в класі-інтерфейсі віртуальний деструктор?
5. Які ключові слова можна використати в реалізації Visual C++ для оголошення класів-інтерфейсів?
6. Чи можна створити об'єкт типу класу-інтерфейсу?
7. У чому відмінність класів-інтерфейсів від абстрактних класів?
8. Чи можна назвати класс-інтерфейс найвищим рівнем абстракції в ООП?

Лекція 13

Тема 5.3. Статичні члени

Статичні перемінні-члени класів

Для звичайних перемінних-членів при створенні кожного об’єкта створюється їхня копія, і доступ до кожної копії можливий тільки через цей об’єкт. Іншими словами, для звичайних перемінних кожний об’єкт має власні копії перемінних-членів.

Перемінні-члени классу можна оголосити як статичні – ключовим словом **static**. Якщо деяка перемінна-член класу оголошується статичною, то може існувати тільки один екземпляр цієї перемінної – незалежно від того, скільки об’єктів даного класу буде створено. Усі об’єкти сумісно використовують цю єдину перемінну. Окрім цього, ця єдина статична перемінна буде використовуватися усіма класами, похідними від класу, в якому ця статична перемінна міститься.

Статична перемінна–член класу створюється ще до того, як буде створений об’єкт цього класу. По суті, такий статичний член класу – це просто глобальна перемінна, область бачення якої обмежена класом, у якому вона оголошена.

У результаті, доступ до статичної перемінної-члену класу можливий без жодного зв’язку із будь-яким об’єктом. Коли у класі оголошуються статичні перемінні-члени, вони не можуть бути визначеними всередини цього класу. Визначити їх треба десь зовні класу, при цьому за допомогою операції ‘::’ треба вказати, у якому класі ця перемінна була оголошена.

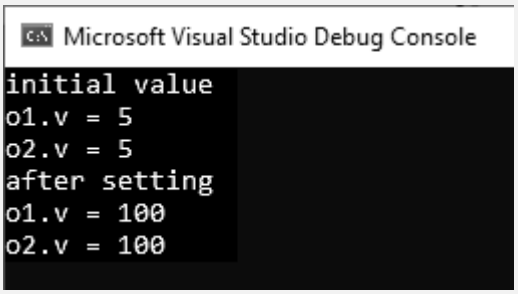
Усі статичні перемінні-члени за замовчуванням ініціалізуються нулем. Проте, якщо необхідно, статичній перемінній класу можна надати будь-яке початкове значення.

Розглянемо приклад класу зі статичною перемінною-членом

```
class MyClass
{
private:
    static int v;
public:
    void setv(int vv) {
        v = vv;
    }
    int getv(void) {
        return v;
    }
};
```

```
//зовнішнє визначення-ініціалізація статичної перемінної v
//ця перемінна була і надалі буде закритим членом класу
int MyClass::v = 5;
```

```
//--test--
int main(void)
{
    MyClass o1, o2;
    cout << "initial value\n";
    cout << "o1.v = " << o1.getv() << "\n";
    cout << "o2.v = " << o2.getv() << "\n";

    o1.setv(100);
    cout << "after setting\n";
    cout << "o1.v = " << o1.getv() << "\n";
    cout << "o2.v = " << o2.getv() << "\n";

    return 0;
}
```



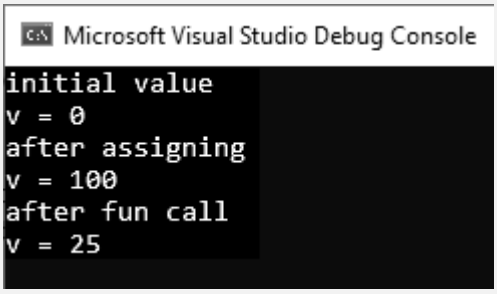
Контрольне запитання: чому при виконанні цього тесту виводяться на консолі однакові значення для o1.v та o2.v?

Оскільки статична перемінна–член класу існує ще до створення об’єкта цього класу, доступ до неї у програмі може бути реалізований без будь-якого об’єкта – якщо така перемінна у класі оголошена як public. У наступному прикладі зробимо статичну перемінну public-членом класу (тоді робити для неї сетер та гетер вже немає жодного сенсу).

```
class MyClass
{
public:
   static int v;
   void fun(int x, int y) {
       v = x * x + y * y;
   }
};
```

```
int MyClass::v;    //без цього лінкер видасть помилку
```

```
//--test--
int main(void)
{
   cout << "initial value\n";
   cout << "v = " << MyClass::v << "\n";
   MyClass::v = 100;   //об'єкти не вказуються
   cout << "after assigning\n";
   cout << "v = " << MyClass::v << "\n";

   MyClass o; //тільки заради виклику fun
   o.fun(3, 4);
   cout << "after fun call\n";
   cout << "v = " << MyClass::v << "\n";
   return 0;
}
```



Статичні функції-члени класів

У класах можна оголошувати статичними й функції-члени. Статичні public функції-члени можуть викликатися зовні без створення об'єкту цього класу як окремі глобальні функції. Вкажемо наступні особливості статичних функцій-членів:

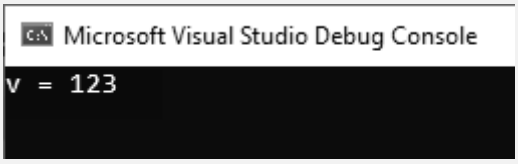
- у статичної функції-члена немає вказівника this
- статичні функции-члени не можуть бути віртуальними.

Розглянемо приклад класу, у якого є private статична перемінна-член та дві public функції: одна звичайна, а друга – статична. Статична функція-член setv буде використана для запису потрібного значення у private перемінну v.

```
class MyClass
{
private:
    static int v;
public:
    static void setv(int x) { v = x; }
    void show(void) { cout << "v = " << v; }
};
```

```
int MyClass::v;      //визначення перемінної v
```

```
//--test--
int main(void)
{
    MyClass::setv(123);   //для цієї функції об’єкт не потрібен

    MyClass Q;
    Q.show();

    return 0;
}
```



Навіщо потрібні статичні функції-члени класів? Як здається, в програмах на C++ вони не дуже широко застосовуються, оскільки в C++ поряд з існуванням класів у коді дозволено використання «некласових» глобальних функцій. Можна вказати лише, що статичні функції-члени має сенс робити, наприклад, для ініціалізації зовні private перемінних-членів класів.

На відміну від C++, у програмах на інших об'єктно-орієнтованих мовах програмування статичні функції-члени досить широко застосовуються. Так, зокрема у мові Java, де усі перемінні та функції мають бути членами якихось класів, статичні перемінні-члени та статичні функції-члени можуть бути застосовані подібно до глобальних перемінних та глобальних функцій [10].

Контрольні запитання

1. Що може означати ключове слово static у різних контекстах программного коду C++?
2. Як довго зберігає значення локальна статична перемінна?
3. Навіщо потрібні статичні перемінні - члени класів?
4. Як ініціалізувати значення статичної перемінної - члену класу?
5. Чи можуть використовуватися статичні перемінні - члени класів без створення об'єктів?
6. Які особливості статичних функцій-членів класів?
7. Як можна використати статичну функцію-член класу?
8. Наскільки широко можуть використовуватися статичні члени класи в об'єктно-орієнтованому програмуванні?
9. Наскільки може відповідати використання статичних членів класів канонам ООП?

Тема 6.1. Поняття патерну проєктування. Патерн Singleton

Поняття патерну проєктування та програмування

Патерни проектування (від англ. *design patterns*) – це напрацьовані ефективні підходи, техніки та правила вирішення задач при створенні програмного забезпечення. Вони не прив’язуються до певної мови програмування і можуть бути застосованими в основному незалежно від конкретної мови. Тим не менш, цілком доцільно назвати їх також паттернами програмування, особливо у випадку орієнтації на якусь мову або технологію програмування.

З англійської слово *“pattern”* можна перекласти як шаблон, модель, схема, зразок і т.п. Доволі часто у літературі використовується термін «шаблон проєктування» (див. наприклад, статтю «Шаблони проєктування програмного забезпечення» у Вікіпедії https://uk.wikipedia.org/wiki/шаблони_проєктування_програмного_забезпечення).

У цьому курсі було прийняте рішення використовувати не «шаблон», а слово «патерн» у контексті *design patterns*. Чому? А тому, що в україномовному описі використання мови C++ шаблонами позначаються *templates* – шаблонні функції та шаблонні класи.

Фундаментальною працею у галузі патернів об'єктно-орієнтованого проєктування та програмування стала надрукована у 1994 році книга *Design Patterns: Elements of Reusable Object-Oriented Software* чотирьох авторів – Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. У цій книзі були зібрані та класифіковані відомі на той час зразки рішень, прийомів об'єктно-орієнтованого проектування. Вони оформлені у вигляді шаблонів (патернів) проектування і представлені діаграмами UML. Відповідно змісту книги Design Patterns маємо наступний перелік патернів

Породжувальні (твірні) патерни:

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

Структурні патерни:

- Adapter
- Bridge
- Composite
- Decorator

- Facade
- Flyweight
- Proxy

Патерни поведінки:

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

Наведений ще наприкінці минулого тисячоріччя перелік патернів ще й дотепер не втрачає актуальності. З часом поступово множина патернів розширюється, для деяких патернів стають уживаними нові назви. Можна навести такі приклади патернів, як

- Dependency Injection
- Multiton
- Simple Factory

Також треба згадати патерн Model-View-Controller, відомий ще по Smaltalk-79.

Головною метою, яку проголошували автори книги *Design Patterns* є сформулювати елементи методології «правильного» об'єктно-орієнтованого дизайну, який дозволяє досягти гнучкості, елегантності та ефективності проекту і зробити такий дизайн здатним для повторного використання.

Навіщо потрібно програмісту знати патерни проектування? Це потрібно для того, щоб швидше узнати та опанувати вже відомі методи, прийоми, техніки об'єктно-орієнтованого проектування та програмування, які вже були раніше напрацьовані і вже була доведена їхня доцільність у багатьох вдалих проектах. Далі розглянемо деякі зразки патернів.

Патерн Singleton

Породжувальний патерн Singleton описує можливість побудови такого класу, для якого може бути створено лише один екземпляр об'єкту. Іншими

словами, клас сам має контролювати створення об’єктів цього класу і забороняти можливість існування декількох екземплярів об’єктів.

Де можна застосувати такий клас? Можно уявити собі програму, яка має єдиний об'єкт-інтерфейс користувача, або єдиний об'єкт для зв’язку з базою даних, або єдиний менеджер роботи з файлами, мережею тощо.

Singleton відноситься до *породжувальних* патернів. Необхідно зазначити, що реалізація такого патерну може бути значною мірою залежною від особливостей конкретної мови програмування. Загалом, перелік патернів ООП, зокрема у книзі *Design Patterns,* був описаний з орієнтацією переважно на мову C++. Це стосується також відомих прикладів реалізації патернів. Воно і не дивно – на момент написання книги мова програмуваня C++ домінувала у промисловій розробці програмного забезпечення.

Класична реалізація Singleton

Класична реалізація на C++ патерну Singleton грунтується на використанні у класі статичного вказівника – члена цього класу.

```
class Singleton
{
private:
   static Singleton * p_instance;
   Singleton() {}
   Singleton( const Singleton& );
   Singleton& operator=( Singleton& );
public:
   static Singleton * getInstance()
   {
      if(!p_instance)
          p_instance = new Singleton();
      return p_instance;
   }
};

Singleton* Singleton::p_instance = 0;
```

Конструктори та оператор присвоювання недоступні клієнтам – вони оголошені як private. Клиенти створюють єдиний та неповторний об’єкт класу через статичну функцію-член **getInstance()**, яка при першому виклику динамично виділяє пам’ять для цього объекту і потім повертає вказівник на цей блок пам’яті. Наприкінці роботи клієнти повинні самі потурбуватися про звільнення пам’яті за допомогою оператора delete. Останній момент є серйозним недоліком класичної реализації патерна Singleton. Якщо клас сам контролює створення єдиного об’єкту, природніше було б покласти на нього і знищення об’єкта.

Розглянемо приклад класу на основі синглтона, у якому також міститься деяка змістовна функція-член. Цей класс буде у окремому модулі

```
//--Файл Singleton.h--
class Singleton
{
private:
   static Singleton * p_instance;
   Singleton() {}
   Singleton( const Singleton& );
   Singleton& operator=( Singleton& );
public:
   static Singleton * getInstance();
   void Func();   //змістовна функція для клієнта
};
```

```
//--Файл Singleton.cpp--
#include "singleton.h"

Singleton* Singleton::p_instance = 0;

Singleton* Singleton::getInstance()
{
   if(!p_instance)
       p_instance = new Singleton();
   return p_instance;
}

void Singleton::Func()
{
   cout << "Calling Singleton Func() member\n";
   cout << "Pointer this = " << this << "\n";
}
```

```
//--test--
#include "singleton.h"

int main(void)
{
   Singleton* s = Singleton::getInstance();

//--спробуємо створити ще один екземпляр—
   Singleton* r = Singleton::getInstance();

//--дослідимо адреси об'єктів--
   cout << "Instance s = " << s << "\n";
   cout << "Instance r = " << r << "\n";

//--викликаємо змістовну функцію-член--
   s->Func();
   r->Func();
   return 0;
}
```

Незважаючи на те, що у даному тесті робиться спроба створити два об’єкта класу-синглтона, тест працює нормально. Здавалось би, створюються два динамічних екземпляри об’єктів – але фактично є лише один об'єкт,

оскільки адреси у вказівниках r і s однакові. Звісно ж, ніщо не заважає зберігати однакову адресу у двох різних перемінних-вказівниках.

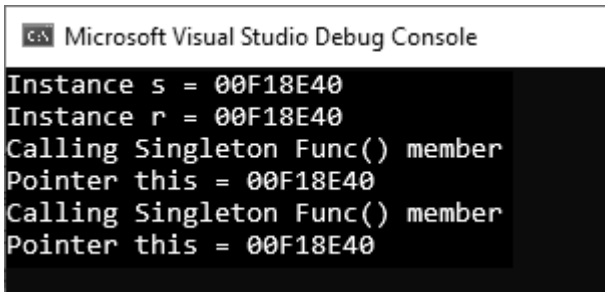


Рис. 13.1. Виконання тесту класу-синглтона

Singleton Меєрса

Вказаний вище недолік класичної реалізації синглтона щодо проблем з його знищенням відсутній в реалізації Singleton, вперше запропонованої Скотом Мейерсом.

```
class Singleton
{
private:
    Singleton() {}
    Singleton( const Singleton& );
    Singleton& operator=( Singleton& );
public:
    static Singleton& getInstance()
    {
        static Singleton  instance;
        return instance;
    }
};
```

Всередині **getInstance()** використовується статичний екземпляр цього класу. Стандарт C++ гарантує автоматичне знищення статичних об'єктів по завершенню програми. Дострокове знищення і не потрібно, оскільки об'єкт Singleton зазвичай є довгожителем у програмі. Статична функція-член getInstance() повертає не вказівник, а посилання на цей объект, цим ускладнюючи можливість помилкового звільнення пам'яті клієнтами.

Наведена реалізация патерну Singleton використовує так звану **відкладену ініціалізацію** (*lazy initialization*) об'єкта, коли об'єкт класу ініціалізується не при старті програми, а при першому виклику getInstance(). Це забезпечується тим, що статична перемінна instance оголошена всередині функції-члена getInstance(), а не як статичний член даних цього класу. Відкладену ініціалізацію, у першу чергу, має сенс використовувати в тих

випадках, коли ініціалізація об’єкта являє собою важку операцію и не завжди використовується.

Недоліки реалізації Мейерса: складність створення об’єктів похідних класів та проблеми безопасного доступу декількох клієнтів до єдиного об’єкта при багатопотоковому виконанні.

```
class Singleton
{
private:
   Singleton() {}
   Singleton( const Singleton&);
   Singleton& operator=( Singleton& );
public:
   static Singleton& getInstance()
   {
   static Singleton  instance;
   cout << "Singleton created\n";
   return instance;
   }
   void Func(void);     //змістовна функція для клієнта
};

void Singleton::Func(void)
{
cout << "Singleton.this = " << this << "\n";  //показ адреси об'єкта
}

int main(void)
{
//--створюємо перший екземпляр об'єкта--
   Singleton& a = a.getInstance();

//--спробуємо створити ще один екземпляр--
   Singleton& b = b.getInstance();

   a.Func();
   b.Func();
   return 0;
}
```

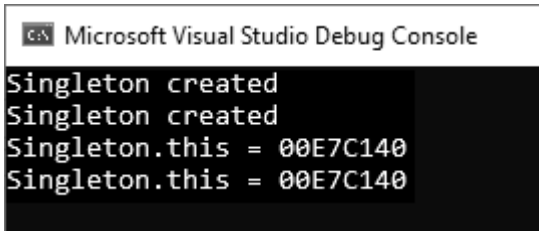


Рис. 13.2. Виконання тесту для синглтону Меєрса

Виконання тесту засвідчує, що замість створення іншого об'єкта функція **`getInstance`** повертає посилання на той самий об'єкт, який був до того створено.

Контрольні запитання

1. Що означає поняття патерну об'єктно-орієнтованого проєктування?
2. Хто вважається фундатором у галузі патернів об'єктно-орієнтованого проєктування?
3. Яке літературне джерело стало класичним у галузі патернів ООП?
4. Які відомі категорії патернів ООП?
5. Що може означати породжувальний паттерн?
6. Яке призначення патерну Singleton?
7. На чому грунтується класична реалізація Singleton?
8. У чому відмінність Singleton Меєрса від класичної реалізації?
9. Чи можливо створювати об'єкти класу Singleton декілька разів?
10. Як знищувати об'єкт класу Singleton?
11. Що доцільно реалізувати у програмі у вигляді об'єкту класу Singleton?

Лекція 14

Тема 6.2. Різновиди патернів Factory

Патерни Factory є *породжуючими* патернами.

Патерн Factory Method

Уявимо собі, що функція клієнта func (глобальна або функція-член класу), створює об'єкт деякого класу продуктів та викликає методи цього об’єкту. Нехай класи продуктів є похідними від деякого базового класу (або інтерфейсу) Product.

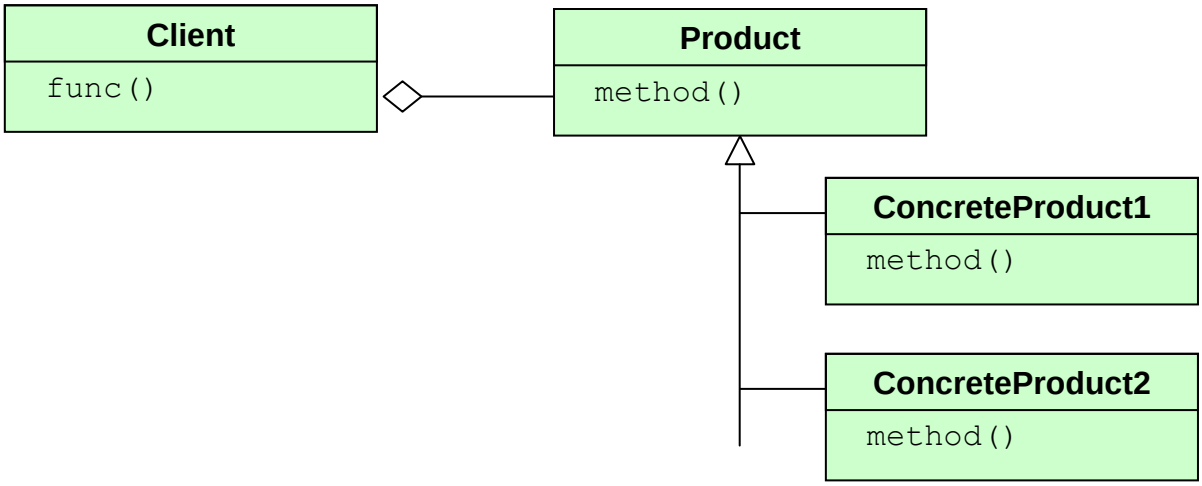


Рис. 14.1. Функція клієнта використовує об'єкти класів продуктів

Якщо функція func описує якісь дії, які однаково можна виконувати з різними конкретними продуктами, то можна зробити у цієї фукції параметр типу вказівник на абстрактний базовий клас (або інтерфейс) продуктів

```
//--визначення функції клієнта--
void Client::func(Product* p)
{
   p->method();
   delete p;
}

//--використання клієнтом конкретних продуктів--
Client A;
A.func(new ConcreteProduct1);
...
A.func(new ConcreteProduct2);
```

У цьому коді закладено те, що коли викликається функція func,то для неї завжди створюється параметр – об'єкт конкретного продукту.

А якщо потрібно, щоб функція func сама створювала об'єкт конкретного продукту, причому не завжди, а при виконанні деяких умов. Може не треба при кожному виклику функції func завжди створювати об’єкти продуктів ще й тому, що якщо такі об’єкти витрачатимуть багато ресурсів, то не створювати їх зайвий раз – це означатиме не витрачати зайвого?

Розглянемо наступний код

```
void Client::func()
{
   If (someCondition())
      {
      Product* p = new ConcreteProduct1;
      p->method();
      delete p;
      }
}
```

Тут передбачено створення лише одного конкретного продукту. А як написати код для, універсальний для усіх типів продуктів? Чи можна передавати якось тип об'єкта, який потрібно створювати у тілі цієї функції?

Для рішення цієї проблеми був запропонований патерн Фабричний Метод (Factory Method). Його суть полягає в тому, щоб створити допоміжні класи фабрики. У класі фабрик є метод, який створює потрібні об'єкти продуктів. Уявимо ієрархію класів фабрик відповідно класам продуктів

```
class Creator       //--базовий клас фабрики продуктів--
{
public:
   virtual Product* factoryMethod() = 0;
};

class ConcreteCreator1 : public Creator
{
   Product* factoryMethod() {
      return new ConcreteProduct1;
   }
};

class ConcreteCreator2 : public Creator
{
   Product* factoryMethod() {
      return new ConcreteProduct2;
   }
};
```

А тепер можна запрограмувати функцію func наступним чином

```
void Client::func(Creator* pc)
{
   If (someCondition())
      {
      Product* p = pc->factoryMethod();
      p->method();
      delete p;       //продукт більше не потрібен
      }
    delete pc;    //фабрика вже не потрібна
}
```

Програмний код цієї функції не залежіть від конкретних продуктів і конкретних фабрик – є залежність лише від абстракцій (базових класів). Тут заздалегідь не відомо, об’єкти яких класів треба створювати. Для того, щоб вказати, з яким конкретним продуктом працювати клієнту, функції клієнта через параметр будемо передавати об'єкт-фабрику конретного продукта

```
//--використання клієнтом конкретних продуктів--
Client A;
A.func(new ConcreteCreator1);
...
A.func(new ConcreteCreator2);
```

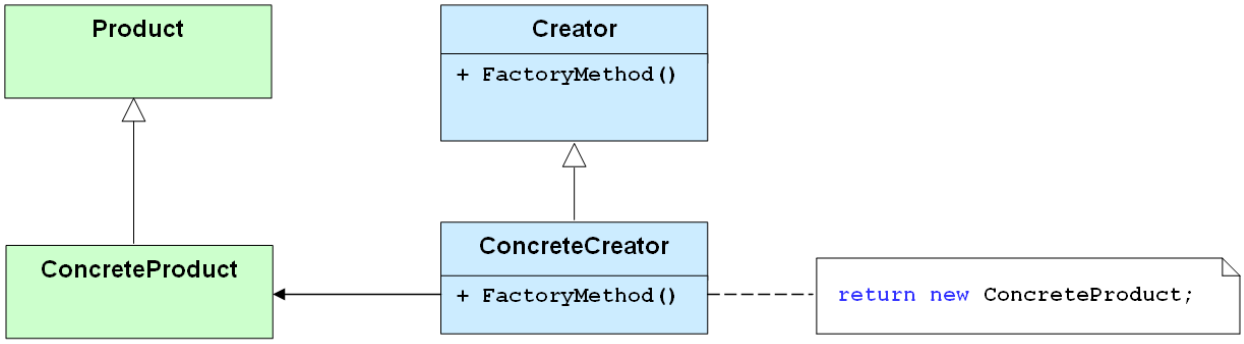


Рис. 14.2. Патерн Фабричний Метод (Factory Method)

Таке відокремлення клієнта від створення об'єктів конкретних продуктів має сенс коли:

- класу не відомо заздалегідь, об'єкти яких саме класів йому потрібно створювати;
- клас спроектовано так, щоб об'єкти, котрі він створює, специфікувалися підкласами;

Патерн проектування Фабричний Метод також інколи зветься як Віртуальний конструктор

Патерн Abstract Factory

Цей патерн також породжує об'єкти. Згідно GoF, патерн Abstract Factory (Абстрактна Фабрика) "надає інтерфейс для створення сімейств взаємоповязаних або взаємозалежних обєктів, не вказучи специфікації їхніх конретних класів" [8].

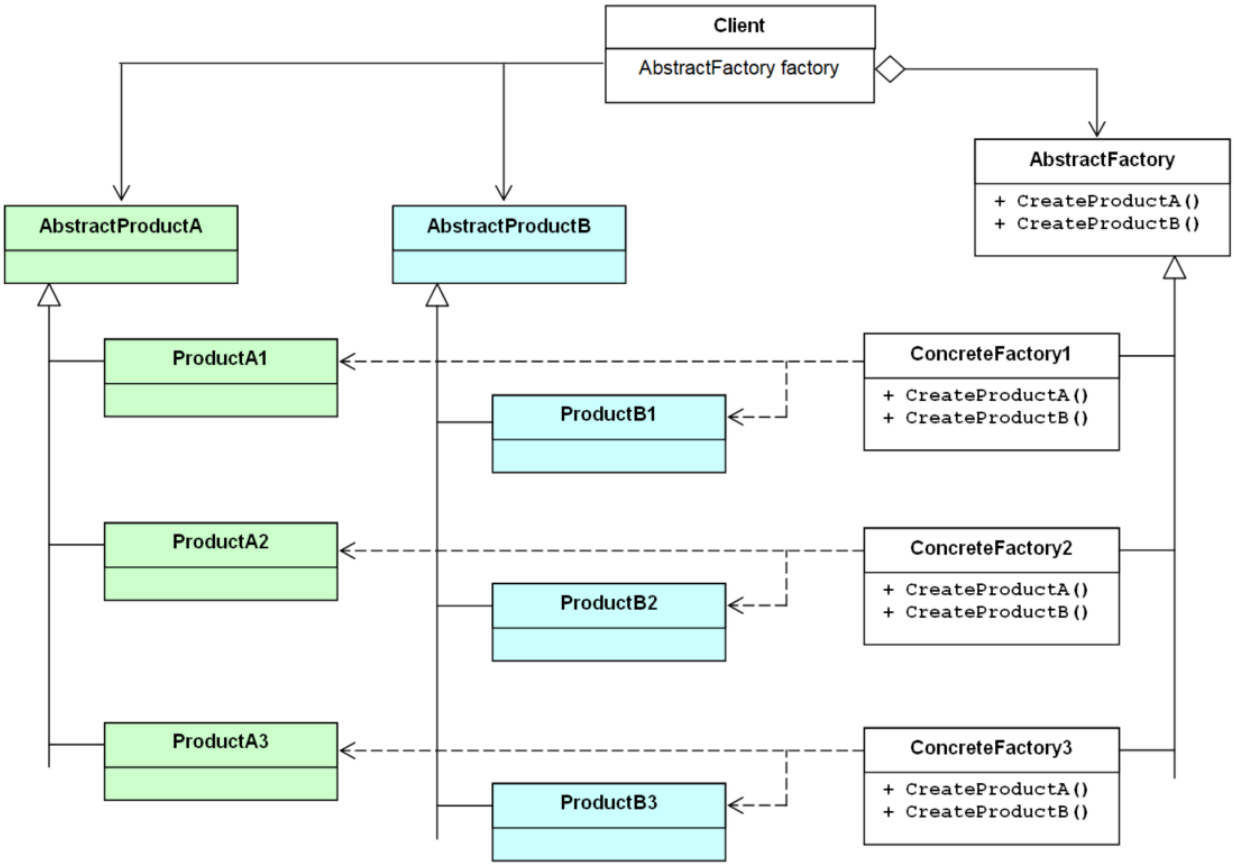


Рис. 14.3. Патерн Abstract Factory

AbstractFactory – оголошує інтерфейс для операцій створення об'єктів

ConcreteFactoryX – реалізують операції створення множини об'єктів конкретних продуктів

AbstractProduct – оголошує інтерфейс для типів об'єктів продуктів

ProductX – визначає об'єкт, якій створюється для певного типу продукту

Client – його код залежіть тільки від оголошень інтерфейсів **AbstractFactory** та **AbstractProduct**

Розглянемо приклад множини фабрик транспортних засобів відповідно патерну Abstract Factory

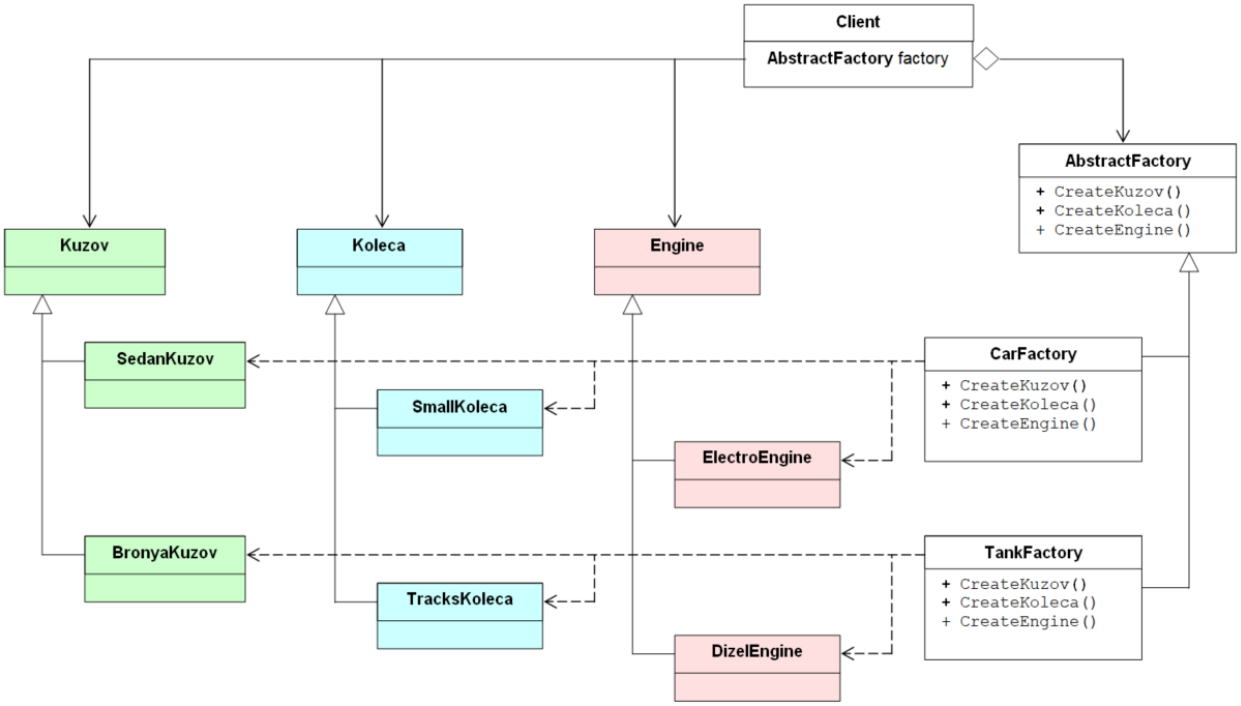


Рис. 14.4. Фабрики транспортних засобів

Кожна фабрика виготовляє множину конкретних продуктів, які відповідають певному типу транспортного засобу. Клієнту немає клопоту вникати, які конкретні типи продуктів потрібно створювати. Це бере на себе конкретна фабрика – забезпечити, щоб для відповідного виробу в комплекті були відповідні складові (об’єкти конкретних продуктів).

Далі розглянемо приклад програмного коду. Класи для продуктів певної категорії, починаючи з абстрактного базового класу:

```
class Kuzov
{
public:
   virtual ~Kuzov() {}
   virtual void Info() = 0;
};

class SedanKuzov : public Kuzov
{
public:
   void Info() { cout << "Sedan Kuzov\n"; }
};

class BronyaKuzov : public Kuzov
{
public:
   void Info() { cout << "Bronya Kuzov\n"; }
};
```

```
class Koleca
{
public:
   virtual ~Koleca() {}
   virtual void Info() = 0;
};

class SmallKoleca : public Koleca
{
public:
   void Info() { cout << "Small Koleca\n"; }
};

class TracksKoleca : public Koleca
{
public:
   void Info() { cout << "Tracks Koleca\n"; }
};
```

```
class Engine
{
public:
   virtual ~Engine() {}
   virtual void Info() = 0;
};

class ElectroEngine : public Engine
{
public:
   void Info() { cout << "Electro Engine\n"; }
};

class DizelEngine : public Engine
{
public:
   void Info() { cout << "Dizel Engine\n"; }
};
```

Класи фабрик

```
class AbstractFactory
{
public:
   virtual ~AbstractFactory() {}
   virtual Kuzov* CreateKuzov() = 0;
   virtual Koleca* CreateKoleca() = 0;
   virtual Engine* CreateEngine() = 0;
};

class CarFactory : public AbstractFactory
{
public:
   virtual Kuzov* CreateKuzov() {
      return new SedanKuzov;
   }
   virtual Koleca* CreateKoleca() {
      return new SmallKoleca;
   }
   virtual Engine* CreateEngine() {
      return new ElectroEngine;
   }
};
```

```
class TankFactory : public AbstractFactory
{
public:
   virtual Kuzov* CreateKuzov() {
      return new BronyaKuzov;
   }
   virtual Koleca* CreateKoleca() {
      return new TracksKoleca;
   }
   virtual Engine* CreateEngine() {
      return new DizelEngine;
   }
};
```

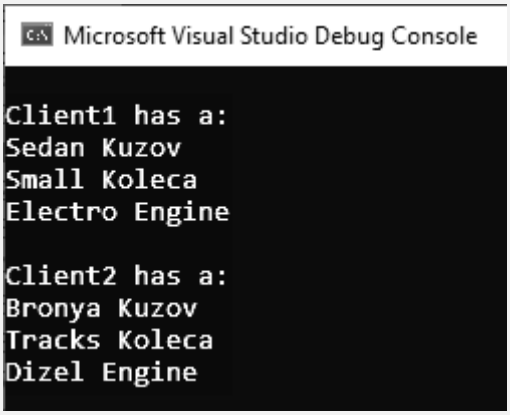
Клас для клієнта фабрик – клас який викликає фабричні методи

```
class Client
{
private:
   Kuzov* pKuzov;
   Koleca* pKoleca;
   Engine* pEngine;
public:
   //--Конструктор викликає потрібну фабрику--
   Client(AbstractFactory* factory)
   {
      pKuzov  = factory->CreateKuzov();
      pKoleca = factory->CreateKoleca();
      pEngine = factory->CreateEngine();
   }

   //--Деструктор знищує усі продукти--
   ~Client()
   {
      if (pKuzov)  delete pKuzov;
      if (pKoleca) delete pKoleca;
      if (pEngine) delete pEngine;
   }

   //--Змістовний метод: клієнт якось використовує продукти фабрики--
   void Run()
   {
      if (pKuzov)  pKuzov->Info();
      if (pKoleca) pKoleca->Info();
      if (pEngine) pEngine->Info();
   }
};
```

Зверніть увагу, у цьому класі використовуються тільки базові класи продуктів та фабрик. Клас Client не залежіть від класів конкретних продуктів.

```
int main(void)
{
AbstractFactory* factory;

//--перший клієнт замовляє у CarFactory--
factory = new CarFactory;
Client* client1 = new Client(factory);
delete factory;
```

```
//--а другий клієнт замовляє у TankFactory--
factory = new TankFactory;
Client* client2 = new Client(factory);
delete factory;

cout << "\nClient1 has a:\n";
client1->Run();

cout << "\nClient2 has a:\n";
client2->Run();

delete client1;
delete client2;
return 0;
}
```



Підсумки

Одною з цілей патернів Фабричний Метод та Абстрактна Фабрика є відокремлення клієнта, звільнення його від необхідності самому створювати екземпляри об’єктів конкретних продуктів. Абстрактна фабрика також зменшує залежності у системі – клієнт вже не залежіть від класів конкретних продуктів і одна фабрика взаємодіє з декількома класами продуктів.

Класи конкретних продуктів є похідними від базового класу (інтерфейсу) продуктів, але кожний продукт може мати певні особливості щодо створення. Так, наприклад, при створенні деяких типів продуктів потрібно формувати для них унікальний серійний номер, а для деяких – ні. Для деяких типів об'єктів може бути потрібна якась спеціальна ініціалізація одразу після створення. – це може взяти на себе фабричний метод відповідно типу конкретного продукту. Таке різноманіття може бути втілено викликом конструкторів з різними списками аргументів а також викликом відповідних функцій членів класів конкретних продуктів. Фабричні методи конкретних класів фабрик можуть брати на себе реалізацію різноманіття особливостей створення об’єктів продуктів, приховуючи від клієнта зайві подробиці. А патерн Абстрактна Фабрика ще й бере на себе зобов’язання створівати сімейства узгоджених екземплярів об’єктів з потрібними властивостями, наприклад, формування персонажів та інших об’єктів з певними властивостями саме для даного конкретного рівня гри.

Можна сказати, що патерни Фабричний Метод та Абстрактна Фабрика означають, зазвичай, існування двох паралельних ієрархій успадкування – класів продуктів та класів фабрик. Зазвичай класи продуктів є окремими класами від класів фабрик, проте можлива реалізація класів продуктів вкладеними класами у класах фабрик.

Контрольні запитання

1. Що виконує фабричний метод у класі?
2. Навіщо може бути потрібен фабричний метод у деякому класі? Які проблеми можна вирішити за його допомогою?
3. Яка структура патерну Фабричний метод?
4. У чому зручність для клієнта замість безпосереднього створення потрібних об’єктів створювати для цього фабрику?
5. Що може приховувати фабрика класів від клієнта?
6. Чому патерн Фабричний метод інколи звуть віртуальним конструктором?
7. Які відмінності патерну Абстрактна фабрика від патерну Фабричний метод?

Лекція 15

Тема 6.3. Патерни Facade, Adapter, Dependency Injection, Bridge

Патерн Facade

Патерн Facade відноситься до категорії *структурних патернів*. Патерн проголошує створення інтерфейсу для системи таким чином, щоб елементи зовні системи могли би звертатися до неї лише через деякий інтерфейс і не мали б доступу безпосередньо до внутрішніх елементів системи.

Оскільки все є системою, а будь-яка система, у свою чергу, складається з інших систем (для неї це підсистеми) то таке налагодження взаємодії між підсистемами може здійснюватися на багатьох рівнях, ієрархічно.

Уявимо, що деяка система охоплює класи клієнта та елементи, які мають відпрацьовувати службові дії, а також відповідати на якісь запити клієнтів. Якщо виділити множину службових елементів системи у підсистему і відокремити їх від клієнтів фасадом з відповідним інтерфейсом, то це може призвести до спрощення зв’язків у системі і чіткіше окреслити повноваження кожного елемента.

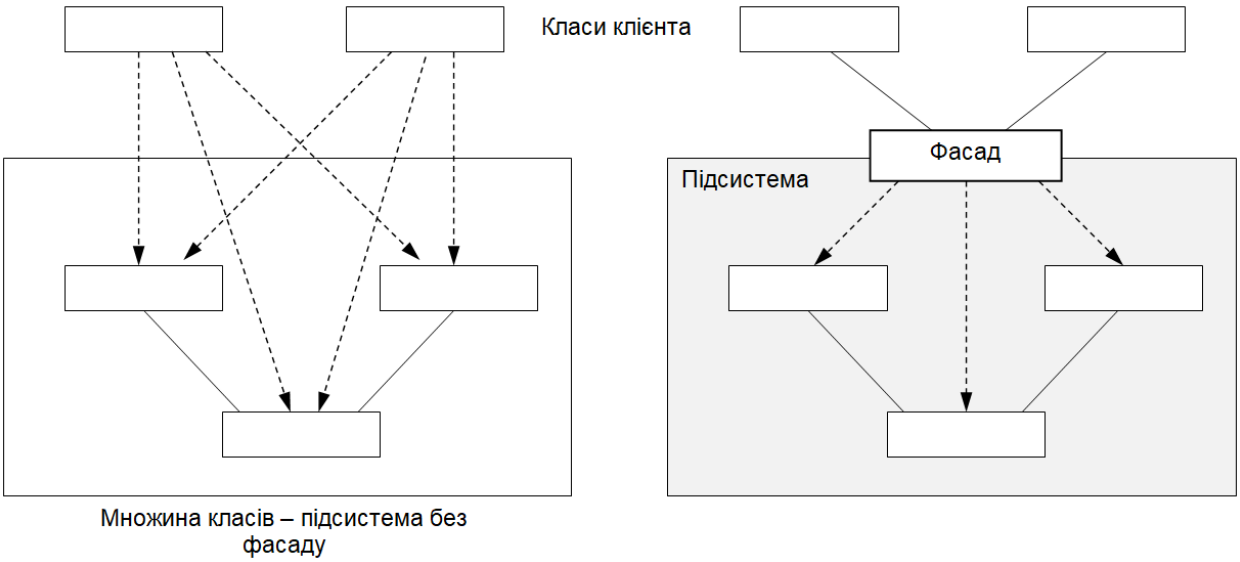


Рис. 15.1. Два варіанти – з фасадом і без нього (джерело: [8])

Як відзначали автори книги “Design patterns” щодо патерну Facade, мотивацією є структурування на підсистеми з виділенням для кожної з них відповідного спрощеного інтерфейсу, що у результаті призведе до зменшення та спрощення зв’язків у системі [8].

Одною з можливих реалізацій фасаду є клас Facade, який має публічні методи – інтерфейс підсистеми для використання класами клієнтів.

Звісно, клас Facade буде мати також деякі закриті поля та методи, які будуть означати якісь внутрішні подробиці реалізації фасаду. Можна сказати, що такий клас буде деякою мірою інкапсулювати підсистему.

Окрім іншого, реалізація патерну Façade дозволяє досягти повного контролю за доступом та використанням підсистеми з боку її клієнтів. Крім того, якщо знадобиться виправляти якісь помилки, вдосконалювати, або й навіть суттєво змінювати внутрішню побудову підсистеми, то можна це робити не чіпаючи класи клієнтів – усі зміни зосереджуються у коді (вихідних текстах) елементів підсистеми і, можливо, лише у закритих членах класу Facade.

Патерн Adapter

Призначення патерну Adapter: "Перетворення стандартного інтерфейсу класа в інтерфейс, більш придатний для потреб клієнта. Застосування шаблона Adapter дозволяє організувати сумісну роботу класів, які мають несумісні інтерфейси" ([DP] перекладено з англ.).

Як здається, найпростіший шлях зрозуміти призначення цього патерна – розглянути на прикладі.

Нехай є множина типів геометричних форм, які описуються ієрархією наступних класів

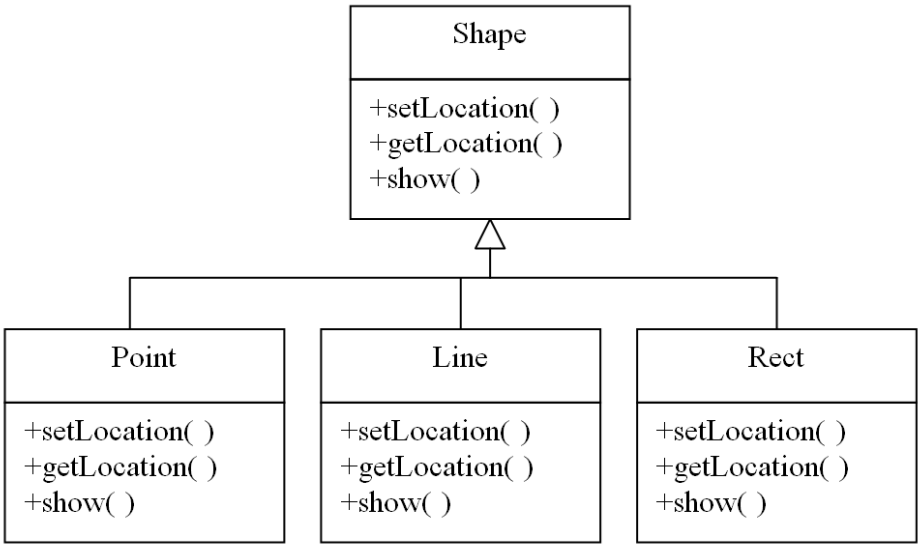


Рис. 15.2. Ієрархія типів форм

Кожний з цих типів описуються вже існуючим на даний момент програмним кодом. Уявимо, що виникла потреба додати у цю множину клас Circle.

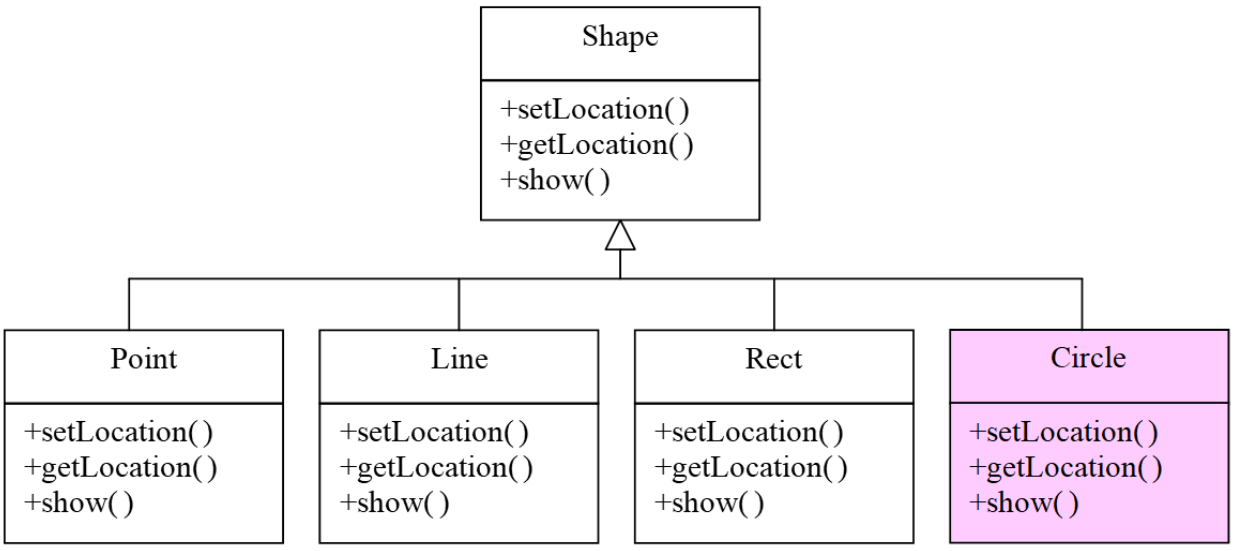


Рис. 15.3. Потрібно додати клас Circle

Звичайно, можна спробувати і запрограмувати клас Circle власноруч, але якось стало відомо, що вже існує (можливо, створений у іншому проекті) клас EllipseFigure, який вирішує подібні завдання, але має інший інтерфейс. Розробник цього класу не хоче змінювати цей клас – тоді будуть порушені вже зроблені проєкти.

Розглянемо можливий варіант рішення, який назвемо «композиційним»

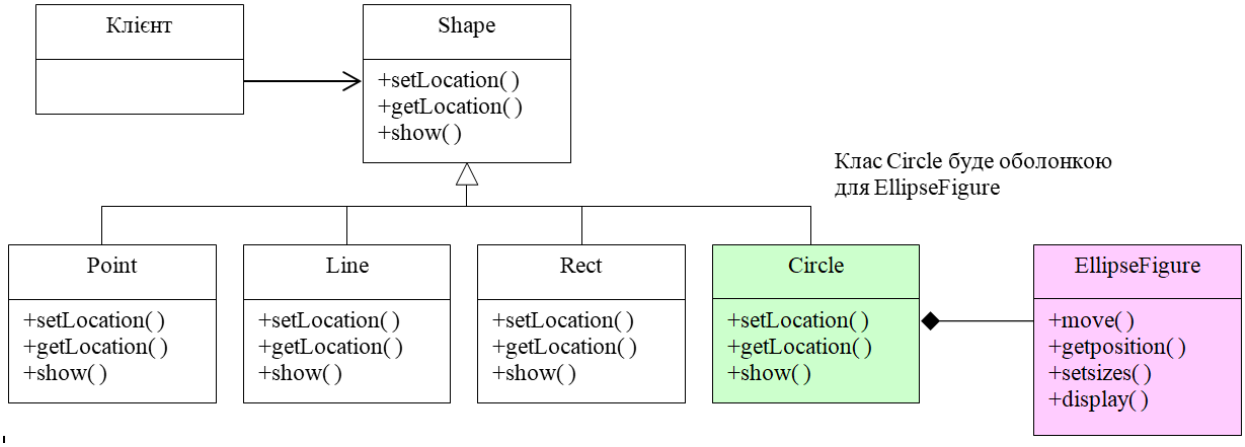


Рис. 15.4. Композиційне рішення

Нижче приклад програмного коду на C++

```
class Circle : public Shape
{
private:
   EllipseFigure *pe;
public:
   Circle() {pe = new EllipseFigure;}
  ~Circle() {delete pe;}
   virtual void setLocation(..);
   virtual void getLocation(..);
   virtual void show(..);
};
```

```
void Circle::setLocation(..)
{
   pe->setsizes(..);
   pe->move(..);
}

void Circle::show(..)
{
   pe->display(..);
}
```

Таким чином, клас Circle буде обгорткою для EllipseFigure. Таке рішення дозволяє робити поліморфізм для функцій класу Circle, так само, як і для інших класів від Shape

```
Shape* ps;
ps = new Circle;
ps->show(..);          //поліморфний вмклик
```

Завдання адаптування вирішено.

Але можливий дещо інший варіант рішення – на основі множинного успадкування класів.

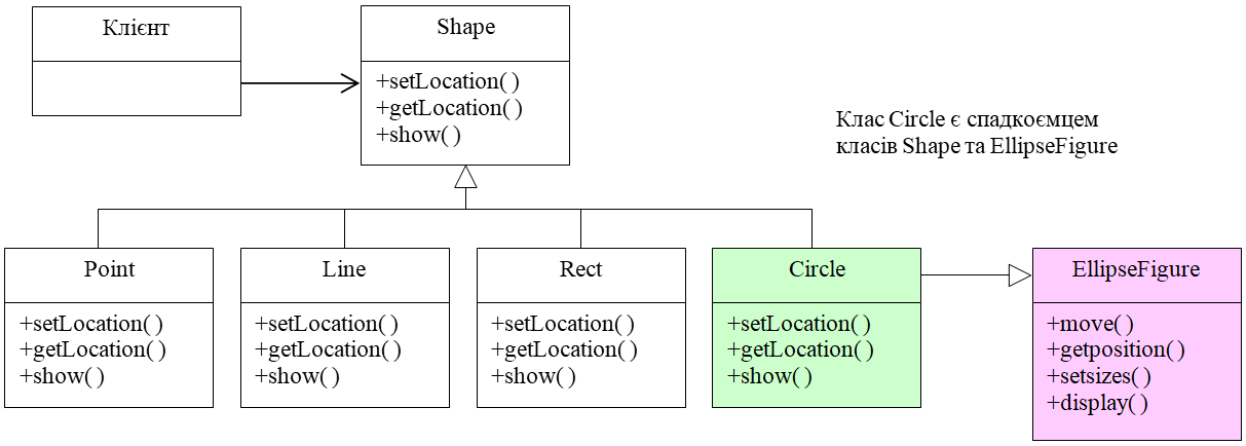


Рис. 15.5. Адаптування на основі множинного успадкування

```
class Circle : public Shape, public EllipseFigure
{
public:
   virtual void setLocation(..);
   virtual void getLocation(..);
   virtual void show(..);
};
void Circle::setLocation(..)
{
   EllipseFigure::setsizes(..);
   EllipseFigure::move(..);
}
void Circle::show(..)
{
   EllipseFigure::display(..);
}
```

Таким чином, можливі, як мінімум два варіанти рішення завдання адаптування. У книзі “Design Patterns” наведені відповідно два варіанти UML діаграм для патерну Adapter. Нижче відображені ці діаграми (з несуттєвими змінами по відношенню до джерела)

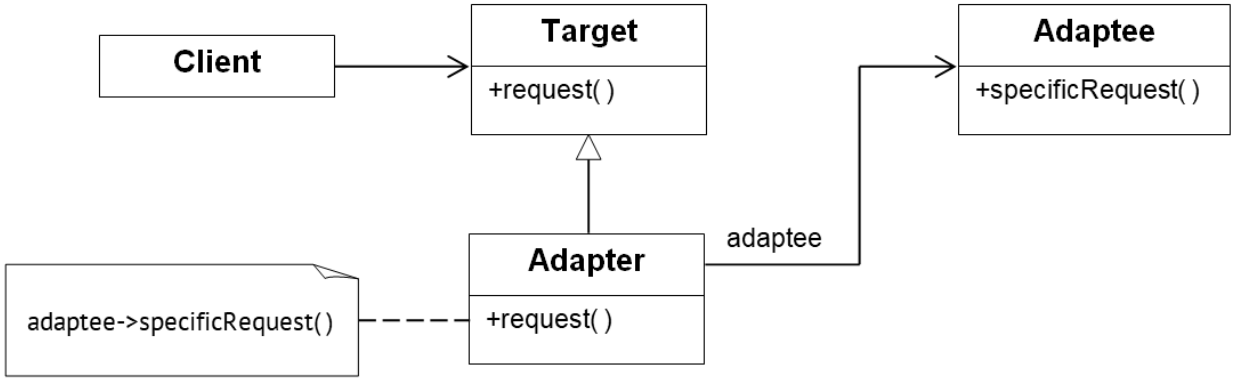


Рис. 15.6. Композиційний варіант патерну Adapter
(джерело: [8])

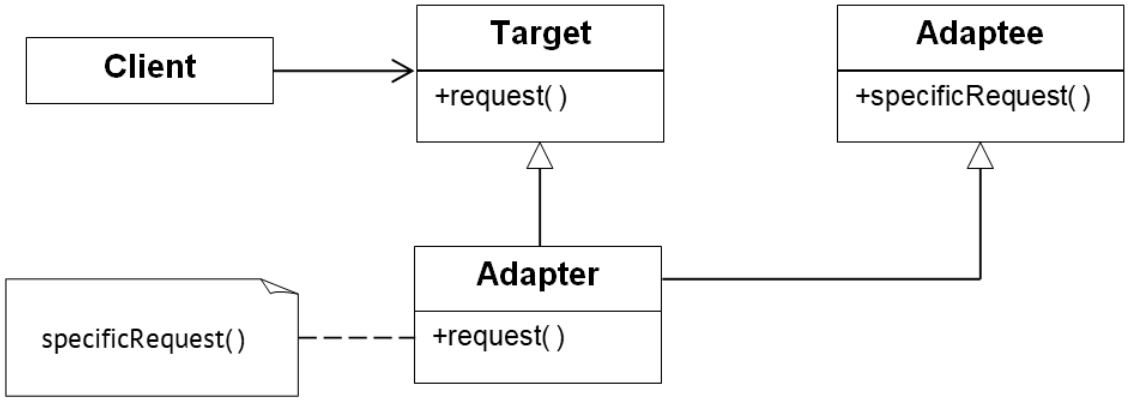


Рис. 15.7. Варіант патерну Adapter на основі множинного успадкування
(джерело: [8])

Патерн Adapter інколи також ще називають як Wrapper (*Обгортка*), хоча слово “Wrapper” застосовується синонимічно і для інших патернів (у першу чергу, для патерну Decorator).

Патерн Dependency Injection

Назву поняття *Dependency Injection* (DI) можна перекласти як *Впровадження Залежностей*. В книзі ”Design Patterns” такого патерну немає. Але, оскільки техніка DI дуже часто використовується в ООП, то у багатьох виданнях це стали назвали патерном DI, починаючі, вірогідно зі статті ”Inversion of Control Containers and the Dependency Injection pattern”, написаної у 2004 році відомим фахівцем ООП Мартіном Фаулером [12].

У чому полягає суть патерну DI? Уявимо, що деякий клас (або модуль) створює і використовує об’єкти інших класів. І якщо у програмному коді цього класу (або модуля) записані імена цих інших класів – значить він залежіть від них, він є їхнім клієнтом.

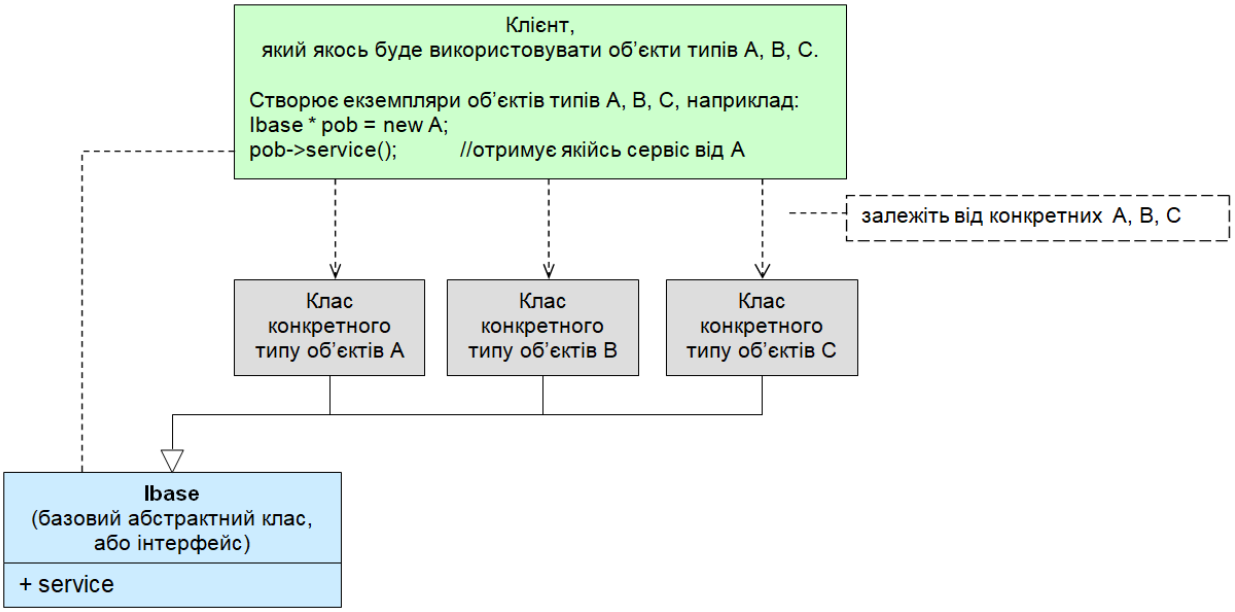


Рис. 15.8. Типова схема залежностей

Техніка Dependency Injection дозволяє прибрати з клієнтського коду створення об’єктів інших класів, зберігши можливості використання таких об’єктів вже опосередковано – при умові що їхні класи є похідними від деякого базового.

Уявимо, що об'єкти типів A, B, C тепер буде створювати не модуль (клас) Клієнт, а інший модуль – назвемо його Інжектором. Цей Інжектор буде передавати Клієнту адресу відповідного об’єкту, сервіс якого має використати Клієнт (рис. 15.9).

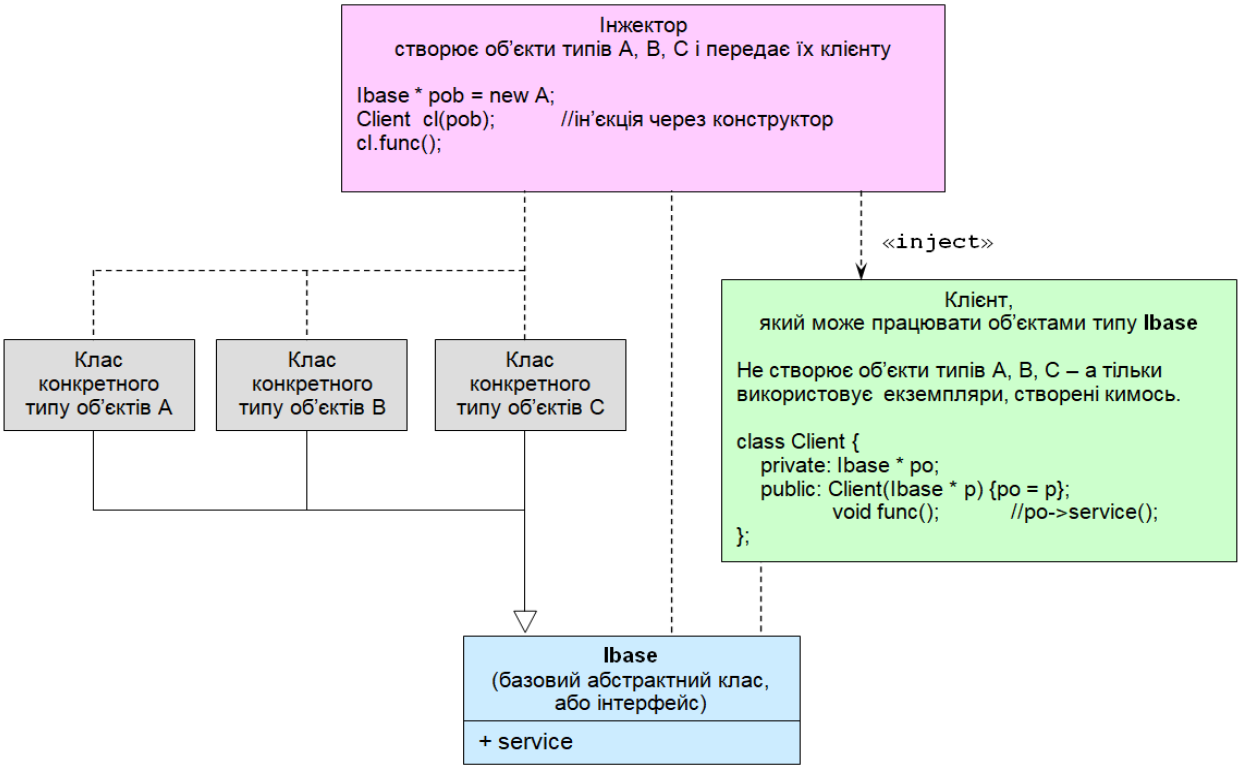


Рис. 15.9. Змінена ієрархія залежностей

Тепер Клієнт вже не залежіть від класів конкретних типів об'єктів A,B,C, хоча Клієнт їх продовжує використовувати. Клієнт тепер залежіть тільки від абстракції – базового класу, або інтерфейсу Ibase. Ось така відбулася інверсія керування (Inversion of Control).

Ін'єкція може виконуватися трьома способами: через конструктор класу клієнта, через спеціальний метод (сетер) або через поле-член.

Через конструктор з параметром

```
class Client
{
private:
   Ibase * po;
public:
   Client(Ibase * p) {
      po = p;
   }

   void func() {
      po->service()
   }
};
```

```
void Injector()
{
Ibase * pob = new A;
Client obj(pob);    //injection
obj.func();
}
```

Або через спеціальний метод

```
class Client
{
private:
   Ibase * po;
public:
   void set(Ibase * p) {
      po = p;
   }

   void func() {
      po->service()
   }
};
```

```
void Injector()
{
Ibase * pob = new A;
Client obj;
obj.set(pob);      //injection
obj.func();
}
```

Або через властивість (поле-член)

```
class Client
{
public:
   Ibase * po;

   void func() {
      po->service()
   }
};
```

```
void Injector()
{
Ibase * pob = new A;
Client obj;
obj.po = pob;      //injection
obj.func();
}
```

Рис. 15.10. Три способи реалізації ін'єкції

Техніка Dependency Injection часто використовується, зокрема у різноманітних класичних патернах ООП.

Патерн Bridge

Згідно книзі “Design Patterns”, призначенням шаблону Bridge (Міст) є відокремлення абстракції від реалізації так, щоб дозволити їм змінюватися незалежно

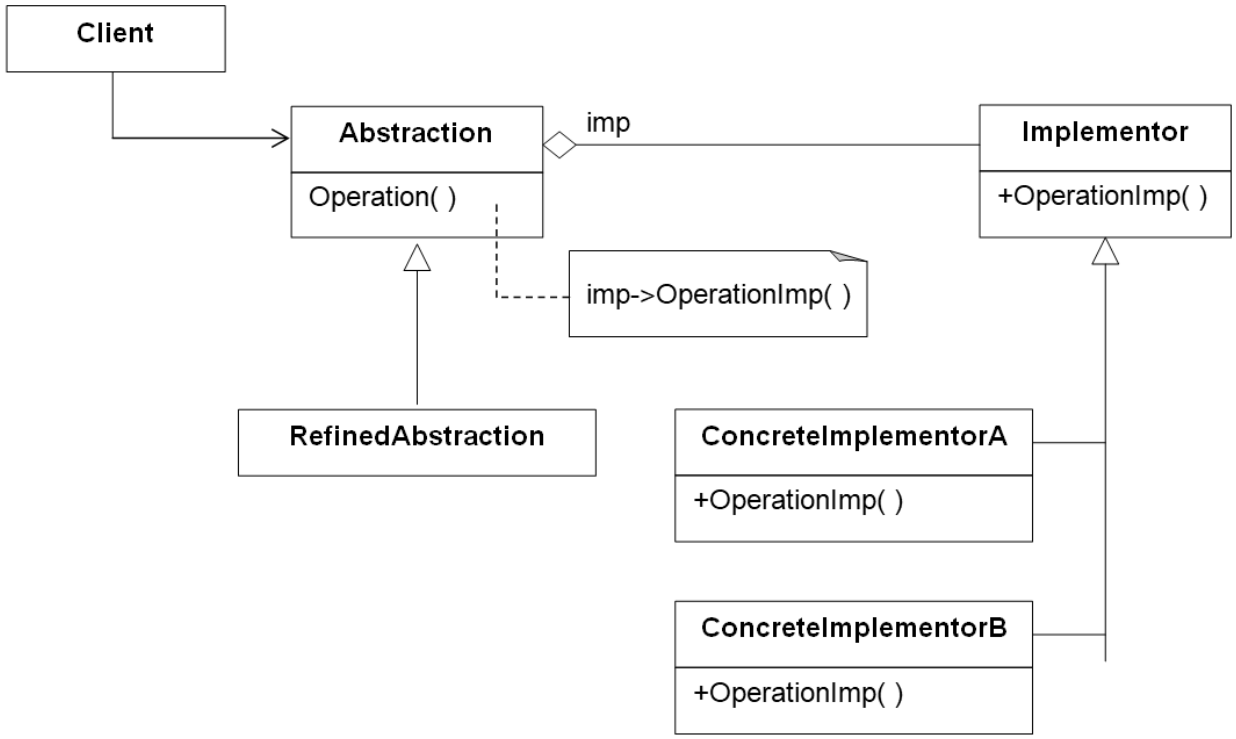


Рис. 15.11. Зображення патерну Bridge (джерело: книга ”Design Patterns”)

Клас Abstraction – це базовий клас для ієрархії типів RefinedAbstraction (уточнення, конкретизація). Клас Abstraction зберігає вказівник (посилання) на базовий клас (інтерфейс) ієрархіє реалізації (Implementor).

Відокремлення реалізації в окрему ієрархію від класу (інтерфейсу) Implementor призводить до того, що програмний код клієнта не залежіть від подробиць реалізації, схованих в похідних класах різновидів конкретних реалізацій деяких процесів. Крім того, вибір потрібної реалізації може здійснюватися прямо у ході виконання програми.

Таким чином, патерн означає доцільність виділення реалізації в окрему ієрархію класів. При цьому, зазвичай, передбачається, що операції реалізації (OerationImp) будуть деякими примітивами, які будуть використовуватися у методах більш високого рівня (Operation).

Якщо уявити собі, що кількість похідних, конкретизованих типів класів ієрархії RefinedAbstraction сягає n, а кількість варіантів реалізації – m, то без розділення на дві ієрархії загальна кількість похідних класів зростала б мультиплікативно, як $n \times m$. А розділення на дві незалежні ієрархії призводить до зменшення загальноє кількості класів до $n+m$.

Розглянемо наступний приклад

```
//--абстракція реалізації--
class Implementor
{
public:
    ~Implementor() {}
    virtual int open(string name) = 0;
    virtual void work() = 0;
    virtual void close() = 0;
};
```

```
//конкретна реалізація 1
class InternetImplementor : public Implementor
{
public:
    int open(string name) {
        cout << "Open internet service for " << name << "\n";
        return 1;
    }
    void work() {
        cout << "Working internet service" << "\n";
    }
    virtual void close() {
        cout << "Close internet service" << "\n\n";
    }
};
```

```
//конкретна реалізація 2
class SMSImplementor : public Implementor
{
public:
    int open(string name) {
        cout << "Open SMS service for " << name << "\n";
        return 1;
    }
    void work() {
        cout << "Sending SMS" << "\n";
    }
    void close() {
        cout << "Close SMS service" << "\n\n";
    }
};
```

```
//конкретна реалізація 3
class DBImplementor : public Implementor
{
public:
    int open(string name) {
        cout << "Open database for " << name << "\n";
        return 1;
    }
    void work() {
        cout << "Store to database" << "\n";
    }
    void close() {
        cout << "Close database" << "\n\n";
    }
};
```

```
//--абстракція об'єктів--
class Abstraction
{
private:
    Implementor* imp;
protected:
    string name;
    virtual void DoSomething() = 0;
public:
    virtual ~Abstraction() {}
    Abstraction(Implementor* p, string str)
    {
        imp = p;
        name = str;
    }
    void Processing()
    {
        DoSomething();
        if (imp->open(name))
        {
            imp->work();
            imp->close();
        }
    }
};
```

```
//об'єкт типу A
class ObjectA : public Abstraction
{
protected:
    void DoSomething() {
        cout << "Processing object typeA " << name << "\n";
    }
public:
    ObjectA(Implementor* p, string str) : Abstraction(p, str) {
        cout << "Create object typeA " << name << "\n";
    }
};
```

```
//об'єкт типу B
class ObjectB : public Abstraction
{
protected:
    virtual void DoSomething() {
        cout << "Processing object typeB " << name << "\n";
    }
public:
    ObjectB(Implementor* p, string str) : Abstraction(p, str)
    {
        cout << "Create object typeB " << name << "\n";
    }
};
```

У якості клієнтського коду буде наступна тестова програма

```
//--test--
int main(void)
{
    Abstraction* pa;
    Implementor* pimp;

    //--працюємо із об'єктами типу A--
    pimp = new InternetImplementor;
    pa = new ObjectA(pimp, "nameA1");
```

```
    pa->Processing();
    delete pimp;
    delete pa;

    pimp = new SMSImplementor;
    pa = new ObjectA(pimp, "nameA2");
    pa->Processing();
    delete pimp;
    delete pa;

    pimp = new DBImplementor;
    pa = new ObjectA(pimp, "nameA3");
    pa->Processing();
    delete pimp;
    delete pa;

    cout << "\n";
    //--працюємо із об'єктами типу B--
    pimp = new InternetImplementor;
    pa = new ObjectB(pimp, "nameB1");
    pa->Processing();
    delete pimp;
    delete pa;

    pimp = new SMSImplementor;
    pa = new ObjectB(pimp, "nameB2");
    pa->Processing();
    delete pimp;
    delete pa;

    pimp = new DBImplementor;
    pa = new ObjectB(pimp, "nameB3");
    pa->Processing();
    delete pimp;
    delete pa;
    return 0;
}
```

```
Microsoft Visual Studio Debug Console
Create object typeA nameA1
Processing object typeA nameA1
Open internet service for nameA1
Working internet service
Close internet service

Create object typeA nameA2
Processing object typeA nameA2
Open SMS service for nameA2
Sending SMS
Close SMS service

Create object typeA nameA3
Processing object typeA nameA3
Open database for nameA3
Store to database
Close database


Create object typeB nameB1
Processing object typeB nameB1
Open internet service for nameB1
Working internet service
Close internet service

Create object typeB nameB2
Processing object typeB nameB2
Open SMS service for nameB2
Sending SMS
Close SMS service

Create object typeB nameB3
Processing object typeB nameB3
Open database for nameB3
Store to database
Close database
```

Як бачимо, для клієнта таких 5 класів доступні два типи об’єктів з трьома варіантами реалізації, тобто, загалом 6 варіантів. Виграш по кількості незначний, але уявимо собі, що кількість типів об’єктів зросте, наприклад, до трьох. Тоді 3+3 = 6 класів дадуть різноманіття усіх 3×3 = 9 варіантів 3 типів об’єктів з 3 реалізаціями. Подібна техніка обмеження мультиплікативного зростання кількості класів використовується й в інших патернах, зокрема, Decorator.

Передавання об'єктам ієрархії Abstraction у якості параметру конструктора вказівників на об'єкти ієрархії Implementor можна вважати використанням техніки впровадження залежностей – від патерну Dependency Injection.

Контрольні запитання

1. У чому основна ідея патерну Фасад?
2. Навіщо ховати подробиці системи від клієнта?
3. Як патерн Фасад декларує розділення системи на підсистеми?
4. Для якого рівня (шару) програмної системи варто створювати фасад?
5. Чим обумовлена наявність двох основних варіантів патерну Адаптер?
6. Як можна впроваджувати залежності у патерні Dependency Injection?
7. Як трансформуються залежності при використанні патерну Dependency Injection?
8. Що означає відокремлення абстракції від реалізації в патерні Міст? Навіщо це потрібно?
9. Чи можливо застосувати декілька ієрархій реалізації відповідно патерну Міст?

Лекція 16

Тема 6.4. Патерни Decorator, Observer, Visitor

Патерн Decorator

Патерн Decorator відноситься до структурних патернів. Головна ідея цього патерну – розширення функціональності системи без створення множини похідних класів. Але, на перший погляд, це виглядає досить дивно: якщо ми хочемо додати новий функціонал, то як зробити це без написання нового коду і оформлення його у вигляді відповідного класу (оскільки згідно парадигми ООП усе має бути у певному класі)? Тут необхідно зробити уточнення – патерн Decorator призначений для вирішення проблеми мультиплікативного зростання похідних класів відповідно усім можливим варіантам комбінування властивостей (функціоналу) об’єктів.

Патерн Decorator означає певну схему щодо можливостей комбінування властивостей об’єктів шляхом організації ланцюжків багаторазового, рекурсивного обгортання завдяки єдиному інтерфейсу створення об’єктів. І таке комбінування властивостей може відбуватися динамічно на етапі виконання програми. Нижче наведена традиційне зображення діаграми UML патерну Decorator, взяте з книги ”Design Patterns”

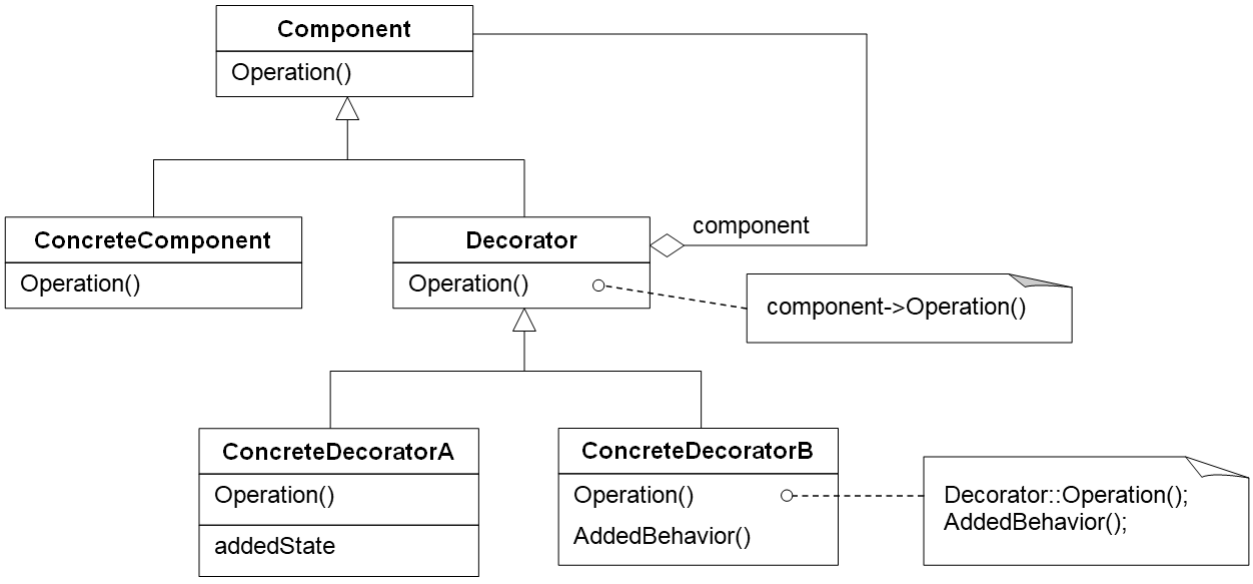


Рис. 16.1. Зображення патерну Decorator (джерело: [8])

Патерн Decorator також ще зветься Wrapper (Обгортка). Те, що об'єкт, який декорується (ConcreteComponent) та декоратори мають спільний інтерфейс (Component), дозволяє робити рекурсивне обгортання.

Розглянемо приклад на C++

```
//абстрактний клас-інтерфейс
//у патерні він позначений як Component
class Ibase {
public:
    virtual ~Ibase() {}
    virtual void run() = 0;
};
```

```
//клас об'єкта, який буде декоруватися
//у патерні він позначений як ConcreteComponent
class O : public Ibase {
public:
    ~O() {  }
    void run() {
        cout << 'O';
    }
};
```

```
//базовий клас для декораторів
class Decorator : public Ibase {
private:
    Ibase* wrappee;
public:
    Decorator(Ibase* inner) {
        cout << "Decorator constructor" << ", ";
        wrappee = inner;
    }

    ~Decorator() { delete wrappee; }

    void run() {
        wrappee->run();
    }
};

//конкретний декоратор A
class DecoratorA : public Decorator {
public:
    //у цьому конструкторі спочатку робиться виклик конструктора базового
    //класу Decorator з параметром core, а потім вже допрацьовує код
    //у фігурних дужках
    DecoratorA(Ibase* core) : Decorator(core) {
        cout << "core = " << typeid(*core).name() << ", ";
        cout << "A constructor" << endl;
    }

    ~DecoratorA() {  }

    void run() {
        Decorator::run();   //спочатку від core
        cout << 'A';        //а потім А-декоруємо
    }
};

//конкретний декоратор B
class DecoratorB : public Decorator {
public:
    DecoratorB(Ibase* core) : Decorator(core) {
```

```
        cout << "core = " << typeid(*core).name() << ", ";
        cout << "B constructor" << endl;
    }
    ~DecoratorB() {  }

    void run() {
        Decorator::run();   //спочатку від core
        cout << 'B';        //а потім B-декоруємо
    }
};

//конкретний декоратор C
class DecoratorC : public Decorator {
public:
    DecoratorC(Ibase* core) : Decorator(core) {
        cout << "core = " << typeid(*core).name() << ", ";
        cout << "C constructor" << endl;
    }
    ~DecoratorC() {  }

    void run() {
        Decorator::run();   //спочатку від core
        cout << 'C';        //а потім C-декоруємо
    }
};
```

Примітка. Для того, щоб у конструкторах декораторів A, B, C першим викликався конструктор базового класу декоратора з параметром (core), його виклик записано у список ініціалізації відповідних конструкторів.

```
//--test--
int main(void)
{
    Ibase* od = new DecoratorA(new O);
    cout << "Result of the run(): ";
    od->run();
    delete od;
    cout << "\n\n";

    od = new DecoratorB(new DecoratorA(new O));
    cout << "Result of the run(): ";
    od->run();
    delete od;
    cout << "\n\n";

    od = new DecoratorA(new DecoratorC(new O));
    cout << "Result of the run(): ";
    od->run();
    delete od;
    cout << "\n\n";

    od = new DecoratorC(new DecoratorB(new DecoratorA(new O)));
    cout << "Result of the run(): ";
    od->run();
    delete od;
    return 0;
}
```

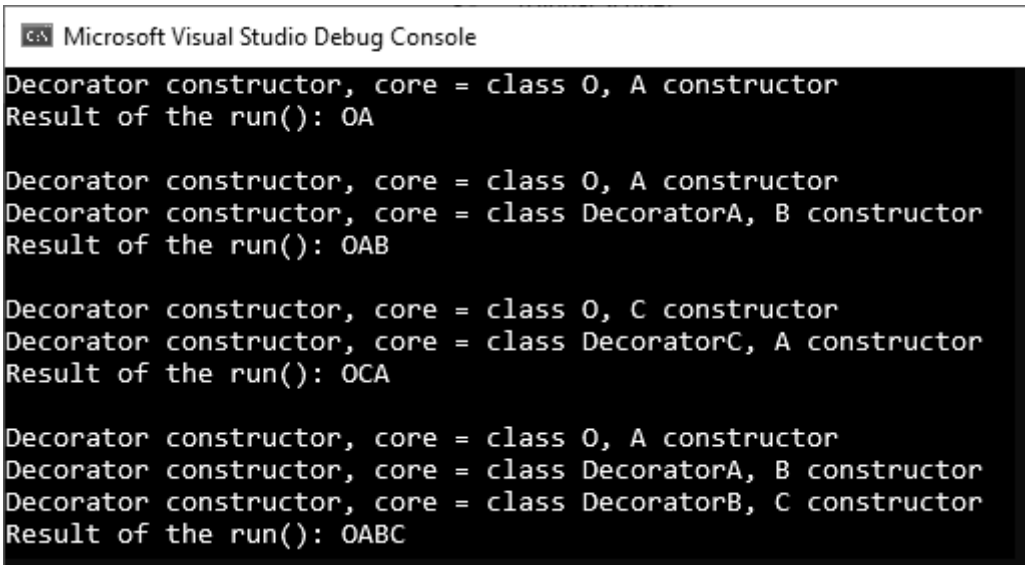


Рис. 16.2. Ілюстрація роботи тесту декорування на C++

Для того, щоб краще зрозуміти особливості декорування, розглянемо реалізацію попереднього прикладу також мовою Java

```
//Base interface for decorators and decorated objects
public interface Ibase {
    void run();
}
```

```
//class for decorated object O
private class O implements Ibase {
    public O() {
        myResult.s += "Class O constructor\n";
    }

    @Override
    public void run() {
        myResult.s += "O";
    }
}
```

```
//Base Decorator class
public class Decorator implements Ibase {
    private Ibase wrapee;

    public Decorator(Ibase inner) {
        this.wrapee = inner;
        myResult.s += "Base Decorator constructor\n";
    }

    @Override
    public void run() {
        wrapee.run();
    }
}
```

```
//Concrete Decorator A
public class DecoratorA extends Decorator {
    public DecoratorA(Ibase core) {
        super(core);
        myResult.s += "DecoratorA constructor\n";
    }

    @Override
    public void run() {
        super.run();
        myResult.s += "A";
    }
}
//Concrete Decorator B
public class DecoratorB extends Decorator {
    public DecoratorB(Ibase core) {
        super(core);
        myResult.s += "DecoratorB constructor\n";
    }

    @Override
    public void run() {
        super.run();
        myResult.s += "B";
    }
}
//Concrete Decorator C
public class DecoratorC extends Decorator {
    public DecoratorC(Ibase core) {
        super(core);
        myResult.s += "DecoratorC constructor\n";
    }

    @Override
    public void run() {
        super.run();
        myResult.s += "C";
    }
}
```

Фрагмент коду програми – створення двох тестових варіантів комбінацій декорування

```
private void onMenuExample1(String title) {
    myResult.s = title + "\n\n";
    myResult.s += "Test1:\n";
    Ibase od = new DecoratorA(new O());
    myResult.s += "Result of the run(): ";
    od.run();

    myResult.s += "\n\n";
    myResult.s += "Test2:\n";
    od = new DecoratorC(new DecoratorB(new DecoratorA(new O())));
    myResult.s += "Result of the run(): ";
    od.run();

    TextView tv = (TextView)findViewById(R.id.textView);
    tv.setText(myResult.s);
}
```

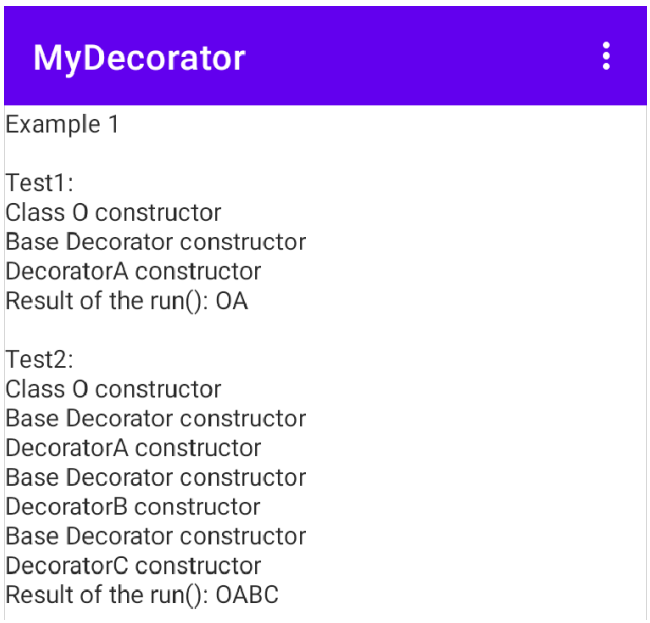


Рис. 16.3. Робота тесту декорування Java на Android

Завдяки наявності в Java ключового слова “super” можна наочно записувати виклик членів базових класів. Так, зокрема, у конструкторах похідних класів декораторів явно викликати конструктор базового класу з параметром-посиланням (core) на об'єкт, який обгортається. У інших аспектах код декорування на Java є подібним до коду на C++.

Підсумки. Таким чином, Декоратор — це структурний патерн, який дозволяє додавати об’єктам нові властивості, розташовуючи їх у об’єкти-обгортки. Причому це можна робити досить гнучко ”на льоту” (а інколи навіть вже і на стадії виконання зкомпільованої програми).

Ознакою Декоратора є ланцюжкове створення об'єктів (обгортка обгортки), причому першим завжди створюється об'єкт, який декорується (O)

```
IBase = new DecoratorC(new DecoratorB(new DecoratorA(new O())));
```

Декоратор дозволяє робити обгортування необмежену кількість разів та у різноманітних комбінаціях завдяки тому, що обгортки та об'єкти, що обгортаються, мають спільний інтерфейс. Це дозволяє уникнути мультиплікативного зростання кількості похідних класів конкретних декораторів. Так, три класи декораторів A,B,C дозволяють робити 7 варіантів декорування: OA, OB, OC, OAB, OAC, OBC, OABC (якщо не враховувати порядок застосування декораторів, наприклад, OAB або OBA). Так, наприклад, щоб створювати 7 різновидів вікна, яке може мати горизонтальний та (або) вертикальний скролінг та (або) товсту границю, то достатньо мати три похідних класа замість семи.

Патерн Observer

Уявимо собі, що у системі потрібно надсилати оповіщення якимось компонентам (спостерігачам) щодо зміни стану деякого об’єкта

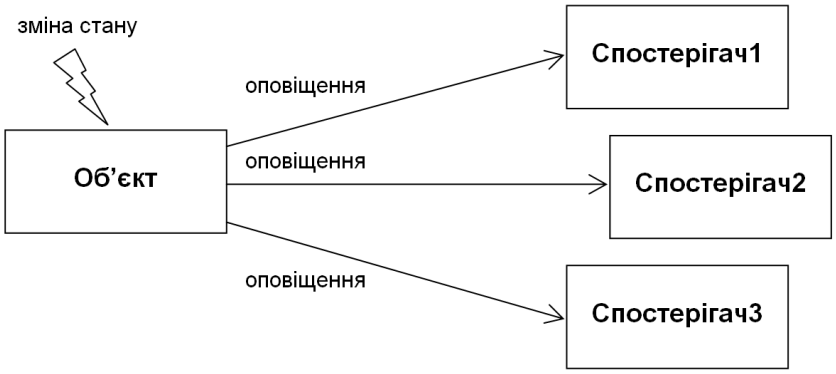


Рис. 16.4. Об'єкт нагляду і спостерігачі

Патерн Observer (Спостерігач) описує схему побудови класів спостерігачів та об’єкта спостереження так, щоб по можливості, уникнути занадто сильної взаємозалежності класів об’єкта та спостерігачів.

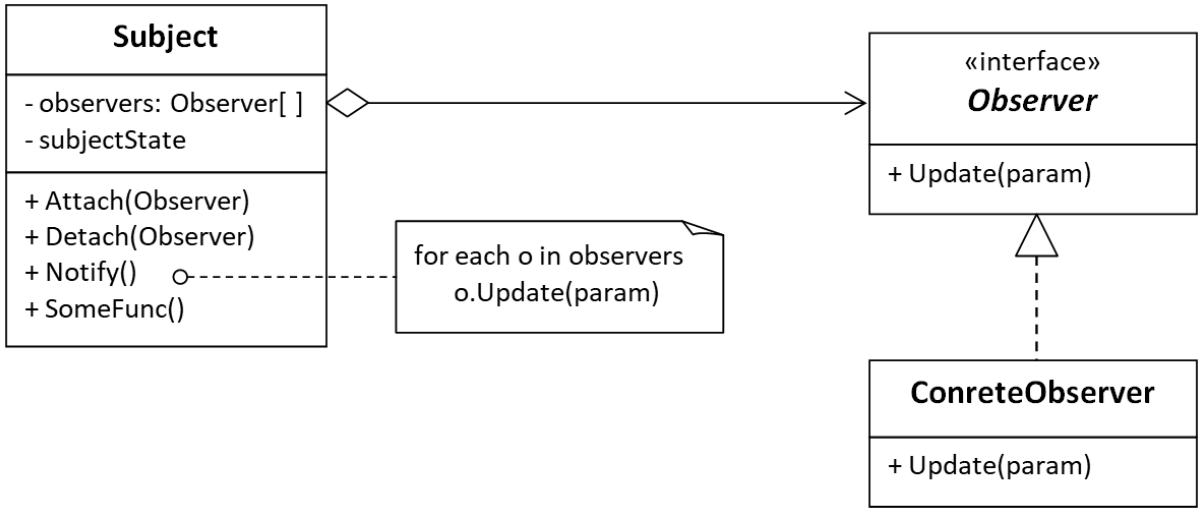


Рис. 16.5. Структура патерну Observer

Observer – інтерфейс з методом Update, який буде викликатися при оповіщенні усіх спостерігачів. Вміст оповіщення може передаватися через параметр метода Update.

ConcreteObserver – один або декілька різновидів конкретних спостерігачів, які будуть реагувати на оповіщення – виклики методів Update.

Subject – об'єкт спостереження. Виконує деяку змістовну роботу, бізнес-логіку (SomeFunc), містить якісь дані, які описують поточний стан об’єкта (subjectState) а також забезпечує підключення–реєстрацію спостерігачів (Attach), їхнє від’єднання (Detach), а також оповіщення усіх зареєстрованих спостерігачів (Notify).

Основні властивості патерну Observer у наведеній вище конфігупації:

- Об'єкт спостереження не залежіть від конкретних спостерігачів. Він використовує лише оголошення інтерфесу **Observer** для підтримки списку спостерігачів.
- Конкретні спостерігачі, які реалізують інтерфейс **Observer,** не залежать від об’єкту спостереження.
- Під’єднання та від’єднання спостерігачів може відбуватися у довільному порядку в ході виконання програми. Виклики методів Attach та Detach можна записувати у клієнтському коді.
- Незалежність спостерігачів і об’єкту спостереження дозволяє побудувати систему взаємодії програмних компонентів зі слабкими зв’язками окремих модулів (класів). Це створює передумови для подальшого розширення та нарощування системи. Можна додавати нові класи спостерігачів без зміни коду класів вже існуючих спостерігачів та об'єкту спостереження.

Розглянемо приклад програмного коду на C++

```
#include <iostream>
#include <string>
#include <list>
using namespace std;
```

```
//--Інтерфейс Observer--
class IObserver
{
public:
    virtual ~IObserver() {}
    virtual void Update(string) = 0;
};
```

```
//--Реалізація конкретних спостерігачів--
class ConrceteObserver1 : public IObserver
{
public:
    void Update(string message)
    {
        cout << "I'm the Observer1. " << message << endl;
    }
};

class ConrceteObserver2 : public IObserver
{
public:
    void Update(string message)
    {
        cout << "I'm the Observer2. " << message << endl;
    }
};
```

```
//--Клас об'єкта спостереження--
class Subject
{
private:
    int subjectState = 0;
    list <IObserver*> observers;
public:
    void Attach(IObserver* o)
    {
        cout << "Attach observer" << endl;
        observers.push_back(o);
    }

    void Detach(IObserver* o)
    {
        cout << "Detach observer" << endl;
        observers.remove(o);
    }

    void Notify()
    {
        cout << "Begin Notify" << endl;
        string message = "The Subject state is " + to_string(subjectState);
        for (const auto& o : observers)     //for all attached observers
            o->Update(message);
        cout << "End Notify" << endl << endl;
    }

    void SomeFunc()
    {
        cout << "Subject.SomeFunc() call" << endl;
        subjectState++;
        Notify();
    }
};
```

```
//--test--
int main(void)
{
    Subject subject;

    ConrceteObserver1* obs1 = new ConrceteObserver1;
    subject.Attach(obs1);
    subject.SomeFunc();

    ConrceteObserver2* obs2 = new ConrceteObserver2;
    subject.Attach(obs2);
    subject.SomeFunc();

    subject.Detach(obs1);
    subject.SomeFunc();

    delete obs1;
    delete obs2;
    return 0;
}
```

Таким чином, в ході роботи такої тестової програми послідовно створюються та під’єднуються два спостерігача. Потім перший з них від’єднується від об’єкта спостереження.

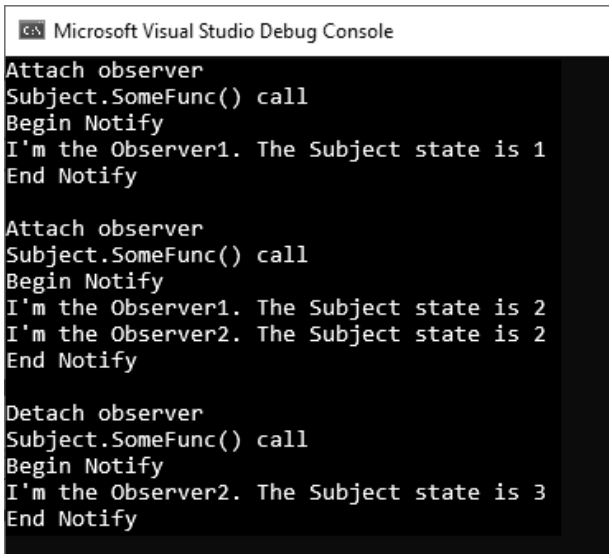


Рис. 16.6. Ілюстрація взаємодії спостерігачів і об’єкта спостереження

Можна уявити собі ситуацію, коли існував деякий клас, наприклад, з ім’ям Subject, але для нього спочатку не передбачалося оповіщення спостерігачам – вочевидь, у нього немає методів Attach, Detach та Notify. А тепер розробник хоче побудувати механізм оповіщення для спостереження за об’єктом класу Subject на основі патерну Observer. Якщо розробник класу Subject не бажає вносити весь програмний код методів Attach, Detach, Notify та структуру даних для зберігання списку спостерігачів в клас Subject, то можливо винести це у окремий клас, наприклад, з іменем EventManager. А клас Subject зробити похідним від EventManager. У випадку, якщо клас Subject і до цього був похідним, але для якогось іншого класу, і робити множинне спадкуваня розробник не бажає, тоді можна зробити композицію Subject та EventManager.

Якщо передбачається декілька різновидів типів об'єктів для оповіщення, тоді доцільно їх реалізувати в класах ConcreteSubject, які будуть похідними від Subject. А у базовому класі Subject залишити лише засоби менеджменту спостерігачів – методи Attach, Detach та Notify, щоб не копіювати багаторазово код цих методів у декілька класів конкретних об’єктів спостереження. Крім того, можливо базовий клас Subject зробити класом-інтерфейсом.

Загалом відомо багато різновидів цього патерну, починаючи з класичної книги ”Design Patterns” [8].

Взаємодія зі спостерігачами може бути організована по-різному. Можливі, як мінімум, два варіанти:

1. Об'єкт спостереження надсилає кожному спостерігачу усю наявну інформацію без урахування того, чи потрібна вона у повному обсязі, чи ні.

2. Об'єкт спостереження може лише інформувати, що відбулися зміни, і після цього кожний спостерігач робить запит на отримання тої порції інформації, яка потрібна саме йому. Запит робиться викликом відповідного метода у об’єкта спостереження, тому це означатиме, що клас спостерігача має залежати від класу об’єкта спостереження. Для зручності роботи спостерігача йому може передаватися вказівник або посилання на об'єкт спостереження, наприклад, об'єкт спостереження може надсилати свій вказівник this через параметр метода Update. Це також має сенс і при спостереженні за багатьма об’єктами.

Можна сказати, що другий варіант взаємодії є більш узагальненим, і саме для нього була наведена діаграма класів у книзі ”Design Patterns” [8]

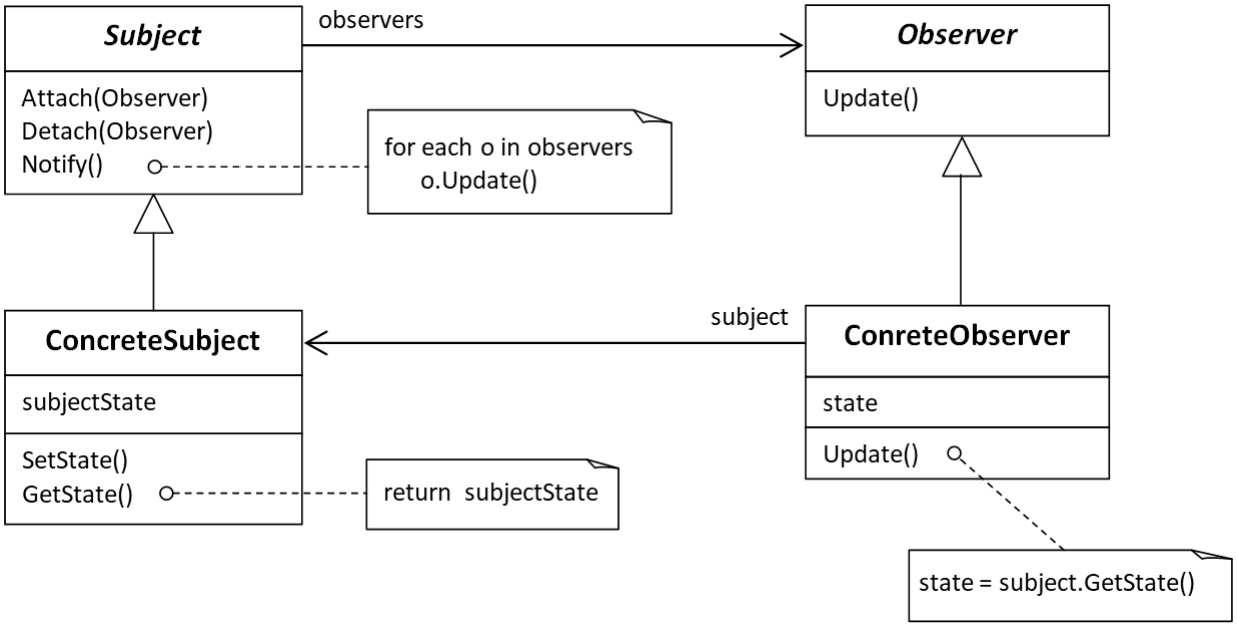


Рис. 16.7. Структура патерну Observer з книги [8]

У даній версії патерну Observer класи ConcreteObserver залежать від ConcreteSubject, тому що можуть зберігати посилання або вказівник (subject) на об’єкти ConcreteSubject, а також можуть викликати методи (GetState) для отримання потрібної інформації від ConcreteSubject при виконанні метода Update.

Функція Notify не обов’язково повинна викликатися об’єктом спостереження – з метою оптимізації потоку оповіщень це може бути покладено на інших учасників.

Для патерну Observer відомі ще інші назви: Видавець-Підписник, Слухач (Listener).

Патерн Visitor

Патерн Visitor (Відвідувач) дозволяє додавати нові операції у об'єкти класів, не змінюючи ці класи.

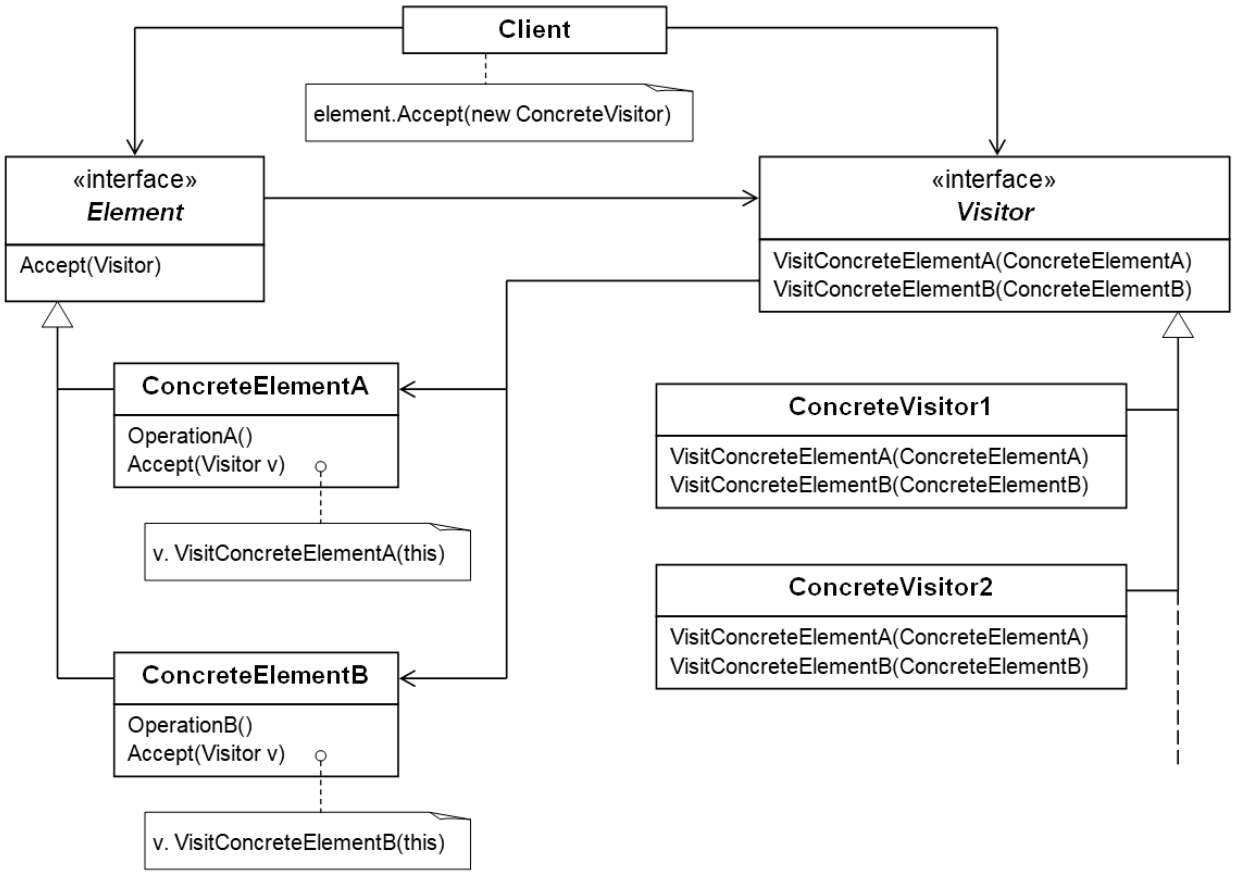


Рис. 16.8. Структура патерну Visitor

Visitor – клас-інтерфейс, або абстрактний базовий клас, у якому оголошуються імена та сигнатури операцій конкретних класів елементів, які планується відвідувати для наділення їх потрібною функціональністю.

ConcreteVisitor – класи конкретних відвідувачів, які реалізують (доповнюють) операції для класів конкретних елементів. Множина класів конкретних відвідувачів може розширюватися шляхом додавання нових класів, які будуть реалізовувати відповідний потрібний функціонал для класів конкретних елементів.

Element – базовий клас, або клас-інтерфейс, у якому для усіх класів елементів оголошується операція Accept – прийняти відвідувача

ConcreteElement – класи конкретний елементів. Множина цих класів є фіксованою, незмінною. Конкретні елементи реалізують операцію Accept з урахуванням операцій від відвідувачів. Також конкретні елементи мають деякі власні операції.

Розглянемо приклад

```
// Попереднє оголошення класів Element
// задля можливості використання їхніх імен у класі Visitor
class Element;
class ElementA;
class ElementB;
```

```
// Базовий клас-інтерфейс Відвідувача
class Visitor
{
public:
    virtual ~Visitor() { };
    virtual void VisitElementA(ElementA*) = 0;
    virtual void VisitElementB(ElementB*) = 0;
};
```

```
// Базовий клас-інтерфейс Елемент
class Element
{
public:
    virtual ~Element() { };
    virtual void Accept(Visitor&) = 0;
};
```

```
// Конкретний елемент A
class ElementA : public Element
{
public:
    virtual void Accept(Visitor& v)
    {
        v.VisitElementA(this);
    }
    void OperationA()
    {
        cout << "OperationA Element A" << endl;
    }
};
```

```
// Конкретний елемент B
class ElementB : public Element
{
public:
    virtual void Accept(Visitor& v)
    {
        v.VisitElementB(this);
    }
    void OperationB()
    {
        cout << "OperationB Element B" << endl;
    }
};
```

```
// Конкретний Відвідувач 1
class ConcreteVisitor1 : public Visitor
{
public:
    void VisitElementA(ElementA* e)
    {
        cout << "ConcreteVisitor1 for Element A" << endl;
        e->OperationA();
    }
    void VisitElementB(ElementB* e)
```

```
    {
        cout << "ConcreteVisitor1 for Element B" << endl;
        e->OperationB();
    }
};
```

```
// Конкретний Відвідувач 2
class ConcreteVisitor2 : public Visitor
{
public:
    void VisitElementA(ElementA* e)
    {
        cout << "ConcreteVisitor2 for Element A" << endl;
        e->OperationA();
    }
    void VisitElementB(ElementB* e)
    {
        cout << "ConcreteVisitor2 for Element B" << endl;
        e->OperationB();
    }
};
```

```
//--test--
int main(void)
{
    ElementA eA;
    ElementB eB;
    Element* elements[4] = { &eA, &eB, &eB, &eA };

    ConcreteVisitor1 v1;
    // працює Конкретний Відвідувач 1
    for (int i = 0; i < 4; i++)
        elements[i]->Accept(v1);

    cout << endl;

    ConcreteVisitor2 v2;
    // а тепер працює Конкретний Відвідувач 2
    for (int i = 0; i < 4; i++)
        elements[i]->Accept(v2);

    return 0;
}
```

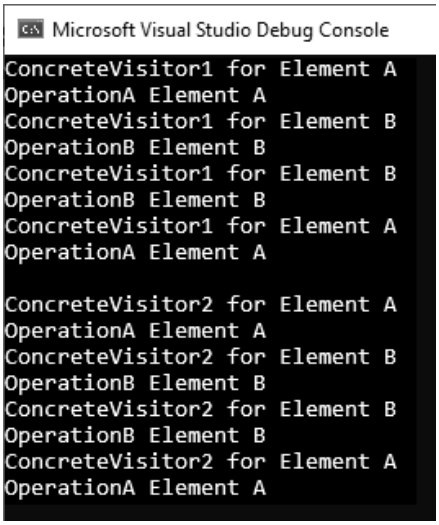


Рис. 16.9. Результати тестів для елементів та відвідувачів

У класі Visitor можна записувати функцію з одним іменем, але багаторазово перевантажену для відповідних типів конкретних елементів.

```
class Visitor
{
public:
    virtual void Visit(ElementA*) = 0;
    virtual void Visit(ElementB*) = 0;
    // і так далі для усіх типів конкретних елементів
};
```

Відповідно спрощується запис методів Accept конкретних елементів

```
class ElementA : public Element
{
public:
    virtual void Accept(Visitor& v)
    {
        v.Visit(this);
    }
    //.. інші члени
};

class ElementB : public Element
{
public:
    virtual void Accept(Visitor& v)
    {
        v.Visit(this);
    }
    //.. інші члени
};
```

Такий підхід зветься «подвійним диспетчеруванням» – автоматичний вибір конкретного метода Visit виконується не тільки поліморфно через посилання (або вказівник) на Visitor, а й через передавання типу конкретного елемента (this) через параметр метода Visit.

Підсумки. Таким чином, патерн Visitor дає змогу, не змінюючи класи деякої ієрархії Element, додавати в них нові операції. Значною мірою функціональність переноситься з самих класів Element в ієрархію спадкування Visitor. Якщо у класах Element це не було передбачено, то у базовий клас Element вноситься метод Accept і далі створюються класи ієрархії Visitor. Реалізація конкретних відвідувачів може ускладнюватися тим, що для виконання певних операції необхідно мати доступ до закритих членів класів Element.

Патерн Visitor доцільно використовувати тільки в тому випадку, якщо ієрархія класів від Element залишається стабільною, незмінною – в усіх класах ієрархії Visitor методи Visit записуються відповідно наявним класам з ієрархії Element. В іншому випадку, потрібно докласти значних зусиль на оновлення всієї ієрархії – починаючи з інтерфейсу Visitor.

Контрольні запитання

1. У чому рекурсивність декорування згідно патерну Декоратор?
2. Як реалізувати патерн декоратор на C++?
3. Які відміінності реалізації декораторів на Java від C++?
4. Навіщо одній частині програми може бути потрібно спостерігати за іншою частиною програми? Наведіть приклади?
5. Як патерн Observer забезпечує незалежність об'єкта спостереження від конкретних спостерігачів?
6. Чи може клас конкретного спостерігача не залежати від класу конкретного об'єкта спостереження?
7. Яку інформацію і кому можна передавати через параметр(и) метода Update в патерні Спостерігач?
8. Яка основна ідея патерну Visitor?
9. Скільки методів має бути в інтерфейсі класів відвідувачів?
10. Які наслідки матиме поява нового похідного конкретного класу елементів відповідно патерну Visitor?

Тема 7.1. Об'єктно-орієнтований підхід vs функціонально-процедурний

Приклад функціонально-процедурного підходу

Нехай потрібно запрограмувати показ чотирьох типів фігур та обчислювати їхню площу. Можна зробити дві функції: ShowShape(type) та ShapeArea(type) і реалізувати це, наприклад, у такий спосіб

```
//символічні константи ідентифікаторів типів фігур
#define SHAPE_POINT 1
#define SHAPE_LINE 2
#define SHAPE_RECT 3
#define SHAPE_ELLIPSE 4

//...   десь зберігаються параметри фігур
```

```
//функція показу фігури типу type
void ShowShape(int type)
{
switch(type)
   {
   case SHAPE_POINT:
      //...  показ точки
      break;
   case SHAPE_LINE:
      //...  показ лінії
      break;
   case SHAPE_RECT:
      //...  показ прямокутника
      break;
   case SHAPE_ELLIPSE:
      //...  показ еліпса
      break;
   default : break;
   }
}
```

```
//функція обчислення площі фігури типу type
double ShapeArea(int type)
{
switch(type)
   {
   case SHAPE_RECT:
      return    //...  площа прямокутника
   case SHAPE_ELLIPSE:
      return    //...  площа еліпса
   default : return 0.0;
   }
return 0.0;
```

```
//клієнтський код виклику функцій для різноманітних фігур
ShowShape(SHAPE_ELLIPSE);
area = ShapeArea(SHAPE_ELLIPSE);

ShowShape(SHAPE_LINE);
area = ShapeArea(SHAPE_LINE);

ShowShape(SHAPE_RECT);
area = ShapeArea(SHAPE_RECT);
. . .
```

А потім з'явилися нові вимоги – у програмі треба обчислювати периметри фігур. Можна додати у програмний код нову функцію

```
//функція обчислення периметру фігури type
double ShapePerimeter (int type)
{
switch(type)
   {
   case SHAPE_LINE:
      return    //...  довжина лінії
   case SHAPE_RECT:
      return    //...  периметр прямокутника
   case SHAPE_ELLIPSE:
      return    //...  периметр еліпса
   default : return 0.0;
   }
return 0.0;
}
```

Старі функції – ShowShape(type) та ShapeArea(type) та структури даних змінювати не потрібно. Тобто, можна казати про можливість повторного використання програмного коду при додаванні нових функціональних можливостей.

Але, вимоги до програми можуть знову змінитися. Уявимо, що стало потрібно, щоб програма працювала б із ще одним типом фігур, наприклад, трикутниками. Тоді для трикутника треба додати ідентифікатор типу

```
//символічні константи ідентифікаторів типів фігур
#define SHAPE_POINT 1
#define SHAPE_LINE 2
#define SHAPE_RECT 3
#define SHAPE_ELLIPSE 4
#define SHAPE_TRIANGLE 5
```

І далі треба переписувати вихідні тексти усіх функцій: ShowShape(type), ShapeArea(type) та ShapePerimeter (type) – в усі switch(type) додавати case SHAPE_TRIANGLE з відповідним програмним кодом зазначених вище функцій тепер і для трикутника.

Приклад об’єктно-орієнтованого підходу

Уявимо, що розроблення програми знаходиться на етапі, коли з вимог відомо лише те, що треба показувати точки, лінії, прямокутники, еліпси та обчислювати їхню площу.

Назвемо фігуру об’єктом і запрограмуємо тип кожної фігури у вигляді класу, починаючи з базового класу Shape. Інтерфейсними методами будуть Show та Area.

```
class Shape
{
//...
public:
   virtual void Show(void) = 0;
   virtual double Area(void) { return 0.0; }
};
```

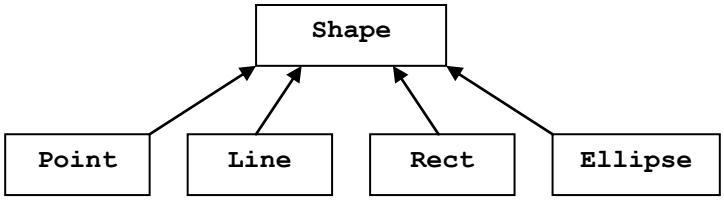


Рис. 16.10. Класи для типів фігур

Конкретні типи фігур будуть представлятися похідними класами. У кожному похідному класі відповідно визначаються функції Show та Area. У клієнтському коді створюються екземпляри обєктів

```
Ellipse el;
el.Show();
area = el.Area();

Line li;
li.Show();
area = li.Area();

Rect re;
re.Show();
area = re.Area();
. . .
```

А що буде, якщо згідно новим вимогам потрібно додати ще якійсь тип фігур, наприклад, трикутник? Тоді для трикутників можна записати новий клас – Triangle, зробивши його похідним від Shape. Відповідно треба визначити функції Show та Area для класу Triangle.

```
class Triangle : public Shape
{
//..
public:
   void Show(void)
   {
   //.. показ трикутника
   }

   double Area(void)
   {
   return  //.. площа трикутника
   }
};
```

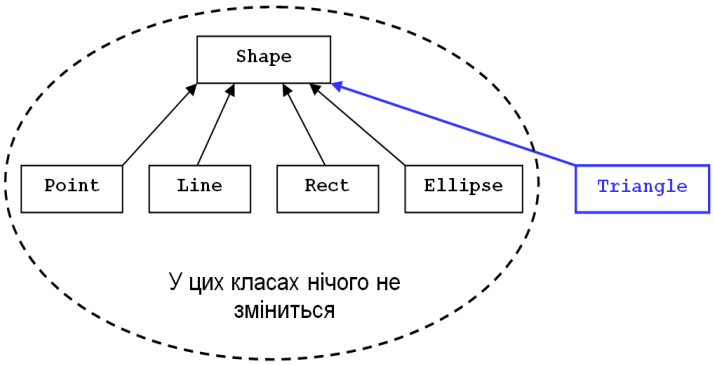


Рис. 16.11. Додавання класу для нового типу фігур

При додаванні класу трикутника попередній програмний код для класів, створених раніше, не змінюється. Таким чином, має місце повторне використання коду при додаванні нових типів об'єктів.

А якщо потрібно додати ще одну функцію, наприклад, обчислення периметру кожного об’єкту? Для того, щоб забезпечувати поліморфізм – треба вносити зміни у базовий клас і далі в усі похідні від нього.

```
class Shape
{
//...
public:
   virtual void Show(void) = 0;
   virtual double Area(void) { return 0.0; }
   virtual double Perimeter(void) { return 0.0; }
};
```

Примітка. Якщо виникає потреба змінювати базовий клас, то це зветься "*крихкість базового класу*".

Загалом, необхідно переписувати програмний код для усіх класів (можливо, за винятком класу Point, оскільки для нього буде використано визначення периметру за замовчуванням у базовому класі).

Таким чином, додавання нових операцій (нової функціональності) є вельми небажаним для ієрархій класів, які орієнтовані на типи об'єктів. Варто сказати, що такий об'єктно-орінтований підхід можна було б назвати «типо-орієнтованим підходом». Для вирішення проблеми з додаванням нових операцій свого часу був запропонований патерн Visitor, який ми вище вже розглядали у цій лекції, проте реалізація цього патерну породжує нові проблеми. Так, зокрема якщо для виконання усіх функцій класам Visitor потрібен буде доступ до закритих даних класів Shape, то це, окрім порушення інкапсуляції, може призвести до суттєвого ускладнення коду.

Expression Problem

Порівнюючі функціонально-процедурний (ФП) та об'єктно-орієнтований підходи, можна сказати, що значною мірою, один підхід є інверсією іншого.

Функціонально-процедурний підхід Програмування від функцій Декомпозиція на процедури по функціям	**Об’єктно-орієнтований підхід** Програмування від типів даних Декомпозиція по типам даних
Функції ☺ **Достатньо легко можна додавати нові функції**, не переписуючі старі, якщо не змінювати типи даних і якщо не виникає проблем із структурами даних	**Функції** ☹ **Важко додавати нові функції.** Якщо вони закріплені у ієрархії класів, то доводиться переписувати усі класи, починаючи з базового
Типи даних ☹ **Важко додавати нові типи** даних – зазвичай доводиться переписувати усі функції, які обробляють такі дані	**Типи даних** ☺ **Легко додавати нові типи,** особливо при розумному використанні спадкування та поліморфізму
Рекомендації На початковій стадії ретельно вибирати та проектувати **структури даних та набір типів даних**, щоб зменшити вірогідність їхніх змін у майбутньому	**Рекомендації** На початковій стадії ретельно проектувати **інтерфейси об'єктів та взаємини класів**, щоб зменшити вірогідність їхніх змін у майбутньому

Рис. 16.12. Інверсність функціонально-процедурного підходу та ООП

Примітка. Не плутати функціонально-процедурний підхід з *функціональним*.

Ще досить давно було помічено, що при розробці програм у процедурному стилі від функцій важко додавати нові типи об'єктів, а при розробці від типів об’єктів важко додавати нові функції. Таке досліджував та описав у 1975 John Raynolds, назвавши це ортогональністю підходів. Пізніше Philip Wadler це назвав ”Expression Problem”. Різні автори висловлювали думку, що, мабуть в межах якогось одного підходу або мови програмування не можна легко забезпечити одночасно нарощування функціоналу і додавання нових типів для об’єктів [23].

Контрольні запитання

1. У чому суть функціонально-процедурного підходу?
2. Наскільки стабільним є код у функціонально-процедурному стилі при додаванні нових операцій?
3. Наскільки стабільним є код у функціонально-процедурному стилі при додаванні нових типів?
4. Наскільки стабільним є об'єктно-орієнтований код при додаванні нових операцій? Наскільки стабільним є об'єктно-орієнтований код при додаванні нових типів?
5. Як співвідносяться функціонально-процедурний підхід та ООП?
6. Що таке Expression Problem?
7. Яку роль у вирішенні Expression Problem може відігравати патерн Visitor?

Лекція 17

Тема 7.2. Принципи SOLID та інші принципи об'єктно-орієнтованого дизайну

Принципи SOLID визначають основні рекомендації щодо розумного об'єктно-орієнтованого підходу побудови програмного забезпечення. Ці принципи в «комплекті» були сформульовані у 2000 р. Робертом Мартіном в статті «Design Principles and Design Patterns» [20]. Але у цій статті не був записаний принцип єдиної відповідальності (SRP) – його було додано у «комплект» базових принципів цим самим автором трохи пізніше – у 2002 році в книзі ”Agile Software Development, Principles, Patterns, and Practices” [19]. Перелік принципів SOLID наведено у таблиці нижче

Таблиця 17.1

Перелік принципів SOLID

Буква	Назва в оригіналі	*Визначення (англ.),* [джерело], коментар
S	Single-Responsibility Principle (SRP)	*A class should have one and only one reason to change* [19]. Якщо клас матиме єдину відповідальність – тоді буде лише одна причина його змінювати.
O	Open-Closed Principle (OCP)	*Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.* [19]. Програмні сутності мають бути відкритим для розширення, але закритим для модифікацій
L	Liskov Substitution Principle (LSP)	*Subclasses should be substitutable for their base classes* [20]. Якщо користувач класів може працювати з об’єктами базових класів, то він так само може працювати і з об’єктами похідних класів.
I	Interface Segregation Principle (ISP)	*Many client specific interfaces are better than one general purpose interface.* [20]. Багато спеціалізованих інтерфейсів краще одного універсального. Один великий інтерфейс варто розділити на декілька дрібних інтерфейсів.
D	Dependency Inversion Principle (DIP)	*High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.* [19]. Модулі верхнього рівня не повинні залежати від модулів нижнього рівня – обидва мають залежати від абстракцій. Абстракції не мають залежати від чогось конкретного. Конкретне має залежати від абстракцій.

Дотримання вказаних принципів на етапі дизайну програмної системи може стати запорукою успішності проекту, який буде можливо без великих проблем підтримувати, вдосконалювати, розвивати, розширювати.

Принцип єдиної відповідальності

Принцип єдиної відповідальності (*Single-Responsibility Principle* – SRP) означає, що сфера відповідальності кожного класу має бути обмеженою. Чим більше обов'язків покладається на клас, тим більша вірогідність можливих змін цього класу, особливо при появі нових вимог – такий клас має низьку стабільність, є крихким.

Найбільшою мірою порушення принципу SRP можна проілюструвати у вигляді так званого антипатерну "God Object". Розглянемо приклад класу

```
class Shape
{
    ...  //інші члени
public:
    void move(double, double);      //пересування фігури
    void setColor(int);             //визначення кольору фігури
    double getArea();               //обчислення площі фігури
    virtual void show(Context);     //показ фігури
    void print();                   //друк фігури
    void writeXML(string);          //запис опису фігури у форматі XML
    void writeFile(FILE);           //запис фігури у файл
    int readFile(FILE);             //читання фігури з файлу
    void writeDbase();              //запис фігури у базу даних
    void readDbase();               //читання фігури з бази даних
    void sendSMS();                 //надсилання фігури по SMS
    void download(Url);             //завантаження фігури з мережі
    virtual Shape* getInstance();   //створення клону об'єкта
};
```

Як здається, цей клас може усе. В усякому випадку, його розробник намагався передбачити усе, що взагалі можливо зробити з об'єктом типу Shape, основна суть якого – опис властивостей просторового об'єкту, який має певну геометричну форму (яку можливо, заплановано конкретно визначати у похідних класах). Чим більше обов'язків і, відповідно, методів класу – тим більша вірогідність потреби змінити цей клас у разі змінення вимог щодо використання таких об'єктів. І це навіть у тому випадку, якщо такий клас використовується лише в одному модулі. А якщо клієнтами цього класу будуть декілька модулів, а тим більше, декілька різних проектів, то вірогідність змін стрімко зростатиме.

Може здатися, що замість такого класу варто оголошувати абстрактний клас або інтерфейс в розрахунку на те, що усі можливі аспекти застосування такого "God Object" будуть враховані у реалізаціях. Але крихкість базового класу залишається – занадто високою буде вірогідність змінювати сигнатури методів (а можливо і склад методів) такого базового класу.

А яким може бути вирішення цієї проблеми? Треба розділювати такий клас на декілька класів відповідно сферам застосування його об'єктів.

Можливим рішенням, може бути, наприклад, таке

```
class Shape
{
public:
    void move(double, double);
    void setColor(int);
    double getArea();
    virtual void show(Context);
    virtual Shape* getInstance();
};

class ShapeFileManager
{
public:
    void writeXML(Shape*);
    void writeFile(Shape*);
    Shape* readFile();
};

class ShapeDbaseManager
{
public:
    void writeDbase();
    Shape* readDbase();
};

//... інші класи
```

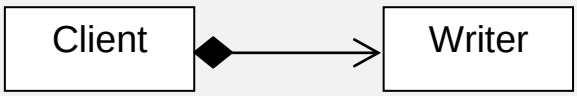
Якщо аналізувати подібне рішення, то воно може бути стабільнішим, оскільки тепер класи розподілені по сферам відповідальності і кожний з них містить небагато методів. Тепер клас Shape з описом фігури, як такої, містить методи які, як здається розробнику класу, можна вважати суттєвими і невід'ємними саме для цього класу. Але, якщо аналізувати доцільність співіснування методів показу та обчислення площі фігур у одному класі, то може виявитися, що це спричиняє певні проблеми.

Подібний приклад було розглянуто у книзі Роберта Мартіна [19], і там вказувалося щодо цього наступне. Уявимо собі, що такий клас (Shape) потрібно використовувати в програмі, яка виконує лише математичні розрахунки з площами об'єктів, і туди не треба тягти код підтримки відображення фігур для метода show. І навпаки, якщо іншій програмі потрібно лише відображати фігури, то програмний код метода getArea тоді буде зайвим. А якщо при зміні вимог до одної з цих програм треба буде у класі щось змінювати стосовно графіки, то такі зміни призведуть до проблем з використанням цього класу в іншій програмі, хоча там графіка можливо, і взагалі не потрібна. І так само може бути з обчисленням площ.

Принцип відкритості-закритості

У якості прикладу розглянемо клас Client, у якому при виконанні якихось дій згідно його бізнес-логіці, має записуватися рядок звіту про кожну дію, наприклад, у файл. Для цього клієнт викликає метод store класу Writer. Клас Client залежіть від класу Writer.

```
class Client
{
private:
    Writer* wr;
public:
    Client() { wr = new Writer; }
    ~Client() { delete wr; }

    void func(string request)
    {
        //.. some busines code
        wr->store("Processed " + request);   //save history
    }
};

class Writer
{
public:
    void store(string report)
    {
        //... write report to file
    }
};
```



А тепер уявимо, що стало потрібно записувати рядки звіту не у файл, а у базу даних. Для цього треба створити клас

```
class DbaseWriter
{
public:
    void store(string report)
    {
        //... write report to database
    }
};
```

І далі потрібно відповідно переписати клас клієнта

```
class Client
{
private:
    DbaseWriter* wr;
public:
    Client() { wr = new DbaseWriter; }
    //... other code
};
```

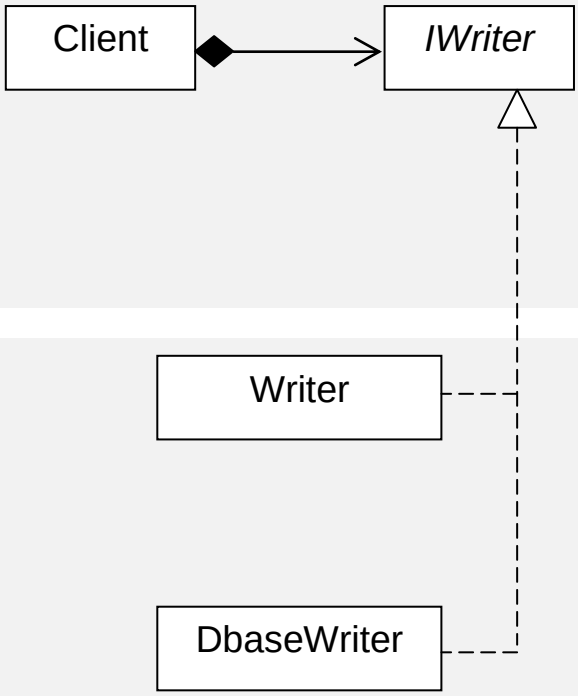
Хоча клас Client не повинен відповідати за подробиці зберігання звіту, але його клас Client довелося змінювати. Тобто, клас Client не закритий для модифікацій.

Для того, щоб стабілізувати програмний код класу Client, треба щоб він залежав не від конкретного класу, а від абстракції. У якості такої абстракції доцільно обирати клас-інтерфейс, наприклад

```
__interface IWriter
{
    void store(string report);
};
```

Запишемо код класу Client, у якому зробимо передавання вказівника на об'єкт записувача через параметр конструктора.

```
class Client
{
private:
    IWriter* wr;
public:
    Client(IWriter* w) { wr = w; }
   ~Client() { delete wr; }
    void func(string request) {
        //.. some busines code
        wr->store("Processed " + request);
    }
};
```




```
class Writer : public IWriter
{
public:
    void store(string report)
    {
        //... write report to file
    }
};

class DbaseWriter : public IWriter
{
public:
    void store(string report)
    {
        //... write report to database
    }
};
```

Конкретні класи записувачів реалізують інтерфейс IWriter. Кожний об’єкт цих класів може бути використаний в об’єкті класу Client.

```
Client* cl = new Client(new Writer);
//або
Client* cl = new Client(new DbaseWriter);
```

Таким чином, програмний код придатний для розширення (класи ієрархії Writer), а клас Client захищений від модифікацій.

Примітка. Якщо в системі щось планується змінювати-розширювати, то така змінна частина має описуватися класами, починаючи від абстракції.

Принцип Лісков

Принцип щодо можливості заміщення базового класу похідним класом був сформульовано Барбарою Лісков у 1988 наступним чином: “What is wanted here is something like the following substitution property: If for each object of type S there is an object of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when is substituted for then S is a subtype of T” [4].

Коли цей принцип Роберт Мартін у 2000-2002 роках включав до складу базових принципів, назвавши його Liskov Substitution Principle (LSP), то він перефразував його наступним чином (у перекладі з англ.): ”Припустимо, що у нас є функція f, яка приймає як аргумент вказівник або посилання на деякий базовий клас B. Припустимо також, що існує деякий похідний D від B, який, переданий f під виглядом B, спричиняє неправильну поведінку f. Тоді D порушує LSP. Очевидно, що D є крихким у присутності f” [19]. Іншими словами, поведінка об’єктів похідного класу не повинна суперечити поведінці, визначеній у базовому класі.

Розглянемо наступний приклад. Уявимо, що є якійсь пристрій, назвемо його DeviceA, який спочатку треба увімкнути (викликом метода turnOn) і після цього викликати метод adjust для настроювання потрібного параметру режиму роботи пристрою.

```
class DeviceA
{
public:
    virtual bool turnOn()
    {
        //... some code
        return true;
    }
    virtual bool adjust(int value)
    {
        //... some code
        return true;
    }
};

class Client
{
public:
    void func(DeviceA *d, int value)
    {
        if (d->turnOn())          //спочатку вмикаємо пристрій
            d->adjust(value);     //а потім налаштовуємо робочий режим
        //... some client code
        delete d;
    }
};
```

А тепер уявимо собі, що з’явився пристрій DeviceB, який описується похідним класом DeviceB. Доцільність робити класс DeviceB похідним від класу DeviceA обумовлюється повторним використанням деякого коду класу DeviceA. Але об'єкт DeviceB працює трохи по-іншому: спочатку треба встановити робочий параметр, а потім підключити пристрій до лінії. Тобто, код функції клієнта має бути вже таким

```
   if (d->adjust(value))          //спочатку налаштовуємо робочий режим
       d->turnOn();               // а потім вмикаємо - підключаємо до лінії
```

Код користувача, записаний для використання класу DeviceA

```
Client cl;
cl.func(new DeviceA, 25);
```

буде працювати некоректно у випадку заміни DeviceA на DeviceB

```
Client cl;
cl.func(new DeviceB, 25);          //буде некоректно працювати
```

Таким чином, замість об'єкту базового класу DeviceA у функцію клієнта не можна підставляти об'єкт похідного класу DeviceB. Відповідно, це є порушенням принципу LSP. Клієнта не варто примушувати розбиратися у відмінностях поведінки об’єктів для кожного класу з певної ієрархії типів.

Виправити становище можна, наприклад, наступним чином: сховати подробиці у закриті члени, а зробити інтерфейсом об’єкта метод run, який буде єдиною точкою доступу до пристроїв з боку клієнта

```
class DeviceA
{
protected:
    virtual bool turnOn()
    {
        //... some code
        return true;
    }
    virtual bool adjust(int value)
    {
        //... some code
        return true;
    }
public:
    void run(int value)
    {
        if (turnOn())
            adjust(value);
    }
};
```

```
class DeviceB : public DeviceA
{
//... some code for DeviceB
public:
    void run(int value)
    {
        if (adjust(value))
            turnOn();
    }

};
```

```
class Client
{
public:
    void func(DeviceA *d, int value)
    {
        d->run(value);
        //... some client code
        delete d;
    }
};
```

Тепер код клієнта буде коректно виконуватися як для DeviceA так і для DeviceB

```
Client cl;
cl.func(new DeviceA, 25);
```

```
Client cl;
cl.func(new DeviceB, 25);          //також коректно працює
```

Незважаючи на те, що вимоги щодо можливості вільної підстановки будь-яким з об'єктів певної ієрархії класів можуть здаватися дещо жорсткими, але дотримання принципу LSP дозволяє покращити побудову ієрархії класів, а також зменшити проблеми тестування коду.

Принцип розділення інтерфейсу

Якщо є деякий інтерфейс з великою кількістю різноманітних методів, то зазвичай потім виникають складнощі, особливо у випадку якщо цей інтерфейс мають реалізовувати декілька класів у різних частинах програми. Можна сказати, що проблеми, пов'язані з наявнітю «жирних» інтерфейсів, подібні до тих, які аналізувалися вище для принципу єдиної відповідальності (SRP). Значною мірою «жирний» інтерфейс подібен антипатерну "God Object".

Якщо інтерфейс містить декілька груп методів і для одного класу-реалізатора потрібна лише одна група, другому класу – інша група методів,

але оскільки усі методи зібрані в одному інтерфейсі, то кожний клас має якось реалізовувати усі його методи.

Основна ідея принципу розділення інтерфейсу (ISP): клієнти не повинні бути змушені залежати від методів, які вони не використовують. Якщо мають бути зроблені якісь зміни, то такі зміни повинні носити локальний характер, і не розповсюджуватися на усі класи клієнтів. «Жирні» інтерфейси треба роздіювати по сферам застосування.

Розглянемо наступний приклад

```
__interface IMyDatabase
{
    bool attach();              //під’єднатися до бази
    void detach();              //від’єднатися від бази, закрити канал зв'язку
    bool isAttached();          //чи під'єднано до бази?
    void store(string);         //зберегти дані
    string get(what);           //прочитати дані
    void remove(what);          //вилучити дані
    void sort(context);         //сортувати контекст
    string request(string);     //виконати запит
    bool isChanged(context);    //чи є зміни?
};
```

Такий інтерфейс варто розділити на декілька інтерфейсів, наприклад

```
__interface IAttach
{
    bool attach();              //під’єднатися до бази
    void detach();              //від’єднатися від бази, закрити канал зв'язку
};
```

```
__interface IAttachControl
{
    bool isAttached();          //чи під'єднано до бази?
};
```

```
__interface IStore
{
    void store(string);         //зберегти дані
    string get(what);           //прочитати дані
};
```

```
__interface IRemovee
{
    void remove(what);          //вилучити дані
};
```

```
__interface IMyDatabase
{
    void sort(context);         //сортувати контекст
};
```

```
__interface IMyDatabase
{
    string request(string);     //виконати запит
};
```

```
__interface IChanged
{
    bool isChanged(context);   //чи є зміни?
};
```

Реалізацію таких спеціалізованих інтерфейсів варто доручити окремим класам, наприклад

```
class AttachManager
{
    bool attach();              //під’єднатися до бази
    void detach();              //від’єднатися від бази, закрити канал зв'язку
};
```

```
class AttachControlManager
{
bool isAttached();         //чи під'єднано до бази?
};
```

Таке розділення доцільно у випадку, якщо при виконанні операцій з базою доведеться перевіряти: чи є у зараз під’єднання до бази даних? Але при цьому не потрібен об'єкт, який містить програмний код встановлення зв’язку з базою та від'єднанням від неї.

Аналогічно можна пояснити доцільність окремого інтерфейсу Ichanged з єдиним методом isChanged() – відстеження наявності змін у певному контексті є сферою відповідальності окремого класу-менеджера.

Принцип інверсії залежностей

Роберт Мартин у книзі “Agile Software Development, Principles, Patterns, and Practices” 2002 року сформулював цей принцип наступним чином: ” Модулі верхнього рівня не повинні залежати від модулів нижнього рівня – обидва мають залежати від абстракцій. Абстракції не мають залежати від чогось конкретного. Конкретне має залежати від абстракцій.” (у перекладі з англ. з [19]).

Як розуміти те, що модулі верхнього рівня не повинні залежати від модулів нижнього рівня? Якщо відображати схему залежностей модулів чи класів, то якщо клас А якось використовує клас В, то значить, клас А залежіть від класу В, і клас А можна розглядати як вищий рівень, а клас В – нижній рівень у ієрархії залежностей. Аналогічно і для модулів – особливо для тих мов програмування, які примушують записувати кожний клас в окремий файл-модуль. Виходить, що на діаграмі залежностей елементи верхнього рівня залежать від елементів нижнього рівня – саме за такими ознаками вони і розташовуються на різних рівнях ієрархії залежностей. Що тут не так? Безумовно, можна на діаграмі класів розташувати базовий клас

зверху а похідні – знизу, і тоді вийде що похідні (залежні від базового) розташуються внизу. Але мова трохи не про це.

Роберт Мартін пояснював, що він мав на увазі модулі верхнього рівня – це політика (Policy), нижче – рівень механізмів (Mechanism), на самому нижньому рівні – утіліти (Utility). Якщо проектувати і програмувати систему у традиційному процедурному стилі тоді й вийде, що модулі (процедури, підпрограми) вищого рівня буде залежати–використовувати модулі, процедури, підпрограми нижчого рівня

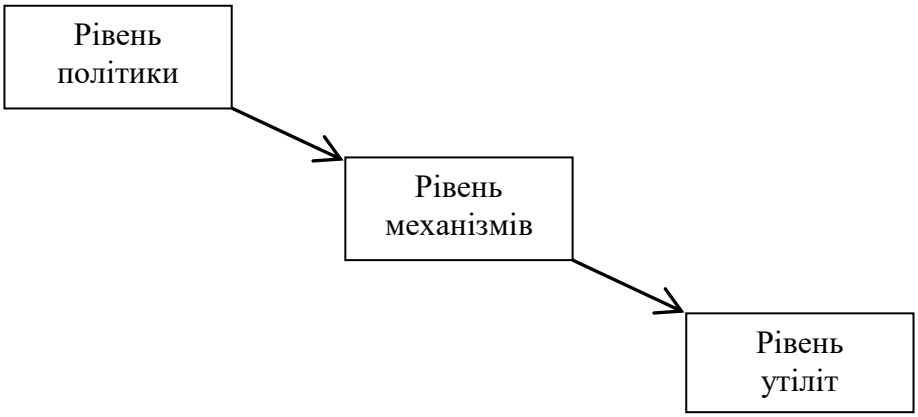


Рис. 17.1. Залежності рівнів модулів системи, спроектованої у процедурному стилі. Джерело: [19]

Але, як відзначав Роберт Мартін, саме модулі вищого рівня мають впливати на модулі нижніх рівнів. Модулі політики мають впливати на модулі реалізації політики. Модулі бізнес-правил високого рівня повинні мати пріоритет над деталями реалізації і не залежати від цих деталей.

А як реалізувати такі побажання? Вище ми розглядали патерни проектування, які реалізовували відокремлення потрібних класів від деталей реалізації. Основна ідея – зробити для потрібного класу залежність не від конкретних класів, а від абстракції – від інтерфейсу. А потім буде можливість інжектувати потрібні конкретні залежності, що створює передумови для досягнення гнучкості рішення. У цій лекції вище ми вже розглядали застосування такої техніки для реалізації принципу відкритості-закритості (OCP).

Можна сказати, що основою для такого дизайну залежностей та взаємодії є техніка callback – «не викликайте нас, ми викличемо вас».

Таким чином, згідно описаній Р.Мартіном методології проектування, модуль політики оголошує інтерфейс сервісів політики, який повинні реалізовувати класи нижнього рівня – механізмів (бізнес-правил). На рівні

цих механізмів оголошується інтерфейс відповідних сервісів механізмів, який має реалізовуватися в модулях ще нижчого рівня.

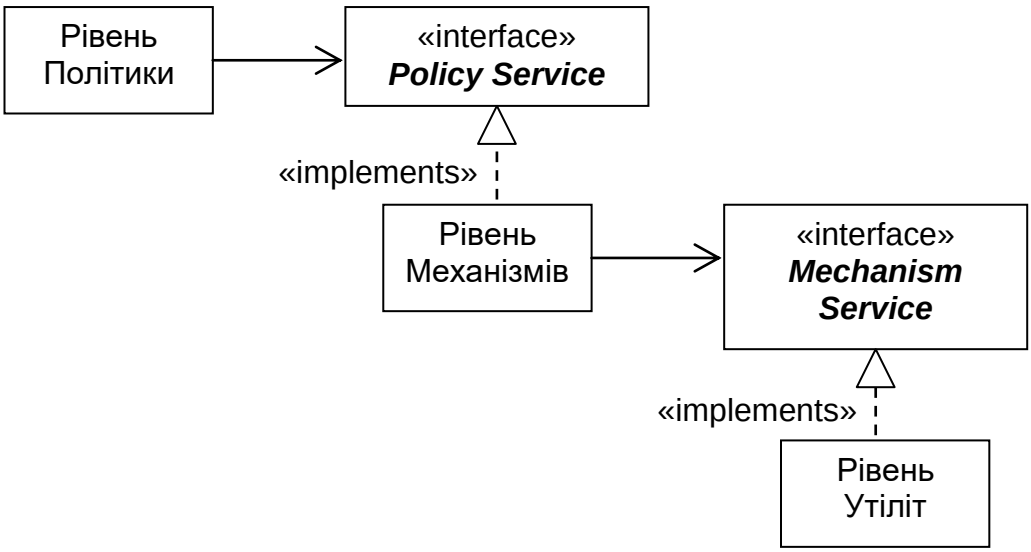


Рис. 17.2. Інверсія залежностей рівнів згідно Роберту Мартіну [19]

Таким чином, можна стверджувати, що повноцінному об'єктно-орієнтованому дизайну притаманне інвертування залежностей рівнів системи порівняно з дизайном у процедурному стилі.

Щодо організації залежностей лише від абстракцій. Роберт Мартін запропонував наступні правила:

- Жодна змінна не має оголошуватися як вказівник або посилання на конкретний клас
- Жоден клас не повинен бути похідним від конкретного класу
- Жоден метод не повинен перевизначати вже реалізований метод в базовому класі [19].

Закон Деметри в ООП

Принцип проектування програмних систем, відомий як «Закон Деметри» має на меті зменшення залежностей у системі. Був проголошений у 1987 році групою дослідників Northeastern University і може бути сформульований наступним чином: кожний модуль не повинен знати про внутрішні подробиці та зв’язки іншого модуля. Дотримання цього принципу, як здається, може допомогти зменшити зв’язність компонентів та забезпечить можливості повторного використання коду, зручне тестування та супроводження програмного забезпечення.

Припустимо, що для деякого класу, який має метод M, створити об'єкт з ім’ям O. Тоді, згідно закону Деметри, мають місце наступні обмеження, зокрема, метод M може викликати лише:

- той самий об’єкт, тобто сам об’єкт O;
- об’єкти, передані як аргументи методу M;
- локальні об’єкти, тобто об’єкти, створені всередині методу M;
- глобальні об’єкти, які доступні об’єкту O;
- безпосередні складові об'єкта O.

Одним з наслідків цих заборон є те, що у випадку, якщо використовується об'єкт деякого классу і цей об'єкт може повертати посилання на об'єкт іншого классу, то не можуть напряму викликатися методи цього іншого классу – правило «не більше одної крапки»

```
a.Method();         //дозволено
a.b.Method();       //а це суперечить закону Деметри
```

Використане джерело: https://uk.wikipedia.org/wiki/Закон_Деметри.

Розглянемо наступний приклад

```
class classC
{
public:
   void MethodC(void) {
      cout << "classC.MethodC()  works!\n\n";
   }
};
```

```
class classB      //клас classB залежіть від класу classC
{
private:
   classC * pc;
public:
   classB(classC * p) {
      pc = p;
   }

   classC * getC(void) {     //надає усім бажаючим доступ до об'єкта класу classC
      return pc;
   }
};
```

```
class classA       //клас classA залежіть від класу classB
{
public:
   void Method(classB * pb)  {
      pb->getC()->MethodC();     //безпосереднє звернення до метода класу classC
                                 //це є порушенням Закону Деметри
   }
};
```

```
int main(void)
{
   classC c;
   classB b(&C);    //об'єкту класу classB передаємо адресу об'єкта класу classC
   classA a;

   a.Method(&B);    //методу класу classA передаємо адресу об'єкта класу classB
   return 0;
}
```

І хоча ця програма успішно працює – в результаті виклику a.Method(&B) виведе на консоль рядок: ”classC.MethodC() works!”, але фактично безпосередньої виклик з класу classA метода класу classC означає, що така програма розроблена всупереч закону Деметри. Якщо з класу в клас передаються вказівники на об'єкти, то правило «не більше одної крапки» можна також доповнити правилом заборони двох і більше ”->”, наприклад

```
pb->getC()->MethodC();      //порушує закон Деметри
```

Багаторівнева архітектура і вимоги закону Деметри

Якщо програмна система побудована з декількох шарів (рівнів), то необхідно, щоб об'єкти (класи) певного *i*-го рівня мали доступ тільки до об'єктів (класів) суміжнього (*i*-1)-го рівня. Об'єкти *i*-го рівня можуть використовувати тільки сервіс від об'єктів (*i*-1)-го рівня. У грамотно спроєктованій системі забороняється безпосередній доступ об'єктів *i*-го рівня до сервісів (функцій або структур даних) (*i*-2)-го рівня.

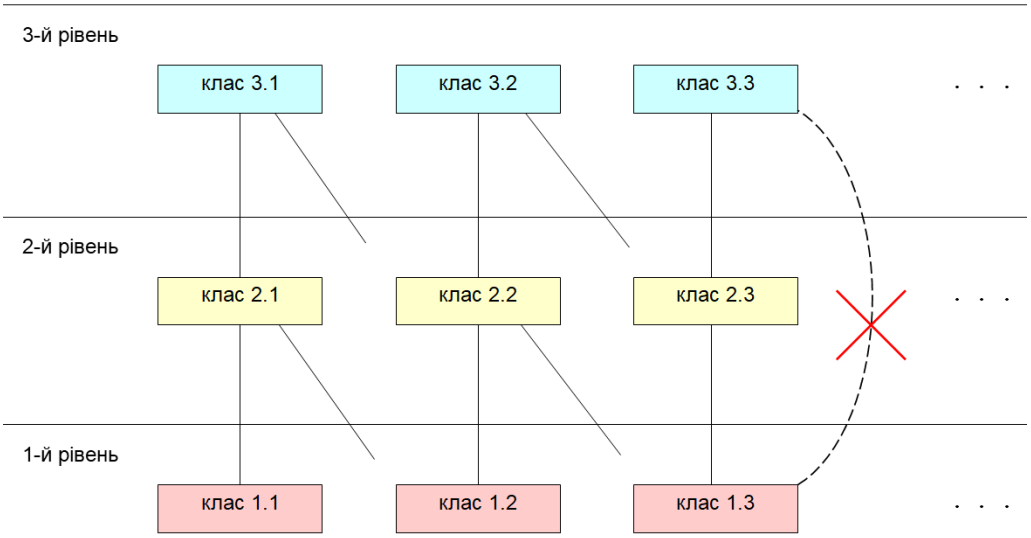


Рис. 17.3. Заборона «дальніх» зв’язків у багаторівневій структурі

Контрольні запитання

1. Що означає абревіатура SOLID?
2. Хто і коли зформулював базовий "комплект" принципів ООП?
3. Що означає принцип єдиної відповідальності?
4. Що таке “God Object”?
5. Як принцип відкритості-закритості пов'язаний з абстракціями?
6. Які обмеження накладає принцип Лісков?
7. Чому інтерфейс з великою кількістю різноманітних методів краще розлілити на декілька менших інтерфейсів?
8. Що означає інверсія залежностей?
9. У чому суть закону Деметри для ООП? Поясніть правило "не більше одної крапки"

Лекція 18

Тема 7.3. Рефакторинг

Поняття рефакторингу

Рефакторинг – це процес зміни системи програмного забезпечення таким чином, щоб він не змінював зовнішню поведінку коду, але покращував його внутрішню структуру.

Якщо розглядати процес розробки програмного забезпечення, то можна сказати, що людині-програмісту майже неможливо за один раз написати такий програмний код, який і буде успішним кінцевим результатом. Завжди знайдеться щось, що потрібно виправити або покращити.

Як виявляється, при використанні об'єктно-орієнтованого підходу для розробки програмних систем необхідність виконувати рефакторинг виникає достатньо часто. Практика використання ООП у багатьох проектах свідчить про це. У 2000 році Мартин Фаулер зі співавторами написав книгу “Refactoring: Improving the Design of Existing Code” – Рефакторинг: покращення дизайну існуючого коду [11]. У цій книзі були зібрані та систематизовані методи та прийоми рефакторингу для ООП.

Про поняття рефакторингу Мартин Фаулер пише наступне: «Це дисциплінований спосіб очищення коду, який мінімізує ймовірність появи помилок. По суті, коли ви виконуєте рефакторинг, ви покращуєте дизайн коду після його написання. «Покращення дизайну після його написання». Це дивний поворот фрази. У нашому поточному розумінні розробки програмного забезпечення ми вважаємо, що ми проектуємо, а потім кодуємо. Хороший дизайн на першому місці, а кодування – на другому. З часом код буде змінюватися, і цілісність системи, її структура відповідно до цього дизайну поступово зникає. Код повільно переходить від розробки до хакерства» [11].

Можна виділити такі важливі аспекти рефакторингу:

- Рефакторинг складається з дрібних простих кроків. Проте, сумарний ефект таких невеличких змін може радикально покращити проект.

- Проектування безперервно відбувається впродовж розробки, а не завершується цілком на початку. При реалізації системи стає зрозуміло, як можна покращити проект. В результаті такої взаємодії якість проекту буде високою.

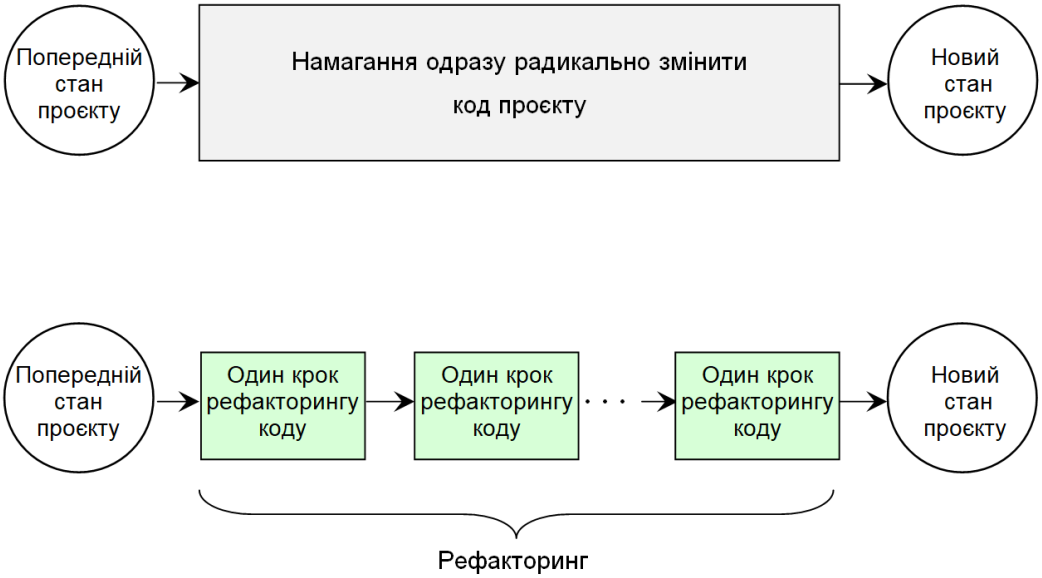


Рис. 18.1. Відмінності можливих підходів покращення коду

Можна уявити собі, що покращення коду можна було б організувати так: спочатку всеохоплюючий аналіз з метою знаходження кращого рішення, а потім тотальне переписування коду проєкту – можливо, одночасно багатьма програмістами команди. Але намагання одразу с першої спроби написати кращий код може бути хибним і така робота може виявитися занадто складною, оскільки наперед треба врахувати забагато факторів, які можуть бути невизначеними.

На відміну цього рефакторинг є послідовним цілеспрямованим виконанням дрібних кроків покращення коду. Чому саме дрібними кроками? Тому що часто необхідність виконання певного кроку не є очевидною від початку, і тільки у процесі змін може стати зрозуміло, що трансформувати у коді далі. Також може бути з'ясовано, що у результаті певного кроку виникають деякі непередбачувані проблеми – тоді варто повернутися на крок назад. Такий підхід по суті є процесом послідовного наближення до оптимального рішення.

Коли потрібно виконувати рефакторинг? У книзі [11] наведено перелік ознак того, що з программним кодом вже щось негаразд (мовою оригіналу "*Bad Smells in Code*").

Ознаки проблемного коду, які можуть стати причиною рефакторингу:

- Дублювання коду
- Довгий метод
- Великий клас

- Довгий список параметрів
- Дивергентність змін. Клас часто змінюється різними способами з різних причин
- «Заздрість» методів – коли метод більше звертається до іншого класу, відмінному від того, членом якого він є оголошеним
- Неструктуровані дані
- Оператор switch. Наявність цього оператора може вказувати на те, що його треба замінити поліморфізмом
- Паралельні ієрархії успадкування
- «Ледачий клас» – клас, який робить недостатньо, щоб виправдати своє існування, слід усунути
- Умоглядна, спекулятивна загальність. Якщо є абстрактні класи, які не роблять багато. Методи з невикористаними параметрами повинні підлягати видаленню параметра. Спекулятивну загальність можна помітити, коли єдиними користувачами методу чи класу є тестові випадки
- Клас даних. Це класи, які мають поля, методи отримання та встановлення для полів (функції get та set) і нічого більше. Такі класи є лише примітивними власниками даних, якими маніпулюють зазвичай занадто часто
- «Відмова від заповіту». Підкласи успадковують усі методи та дані від базових класів. Але, якщо вони не потребують усього від них – то це означає, що ієрархія успадкування неправильна
- Інші ознаки.

Варто відзначити, що деякою мірою має місто парність категорій. Так, зокрема, з одного боку потрібно орієнтуватися на абстракції, але з іншого боку, якщо оголошено забагато абстрактних класів, які не роблять багато – то це теж погано. Неструктуровані дані є ознакою поганого коду, але наявність класів даних, у яких є тільки поля, а у якості методів присутні тільки сетери та гетери – є теж сумнівним дизайном.

Далі розглянемо докладніше деякі з методів рефакторингу.

Заміна switch поліморфізмом

Якщо ми у програмному коді десь бачимо switch(**щось**), то це “**щось**” треба зробити класом з віртуальним методом і оголосити похідні класи відповідно варіантам вибору. У похідних класах визначити програмний код метода для кожного з варіантів, а замість switch записати поліморфний виклик метода.

І усе це не тому, що оператор switch є поганим – зовсім ні, він працює дуже швидко і у багатьох випадках є незамінним. Не треба вважати, що будь що треба обходитися без нього (подібно до того як колись закликали зовсім не використовувати оператор goto). Але з точки зору ООП, якщо у деякому програмному коді switch записано для вибору альтернатив для різних типів варіантів, то якщо множина варіантів може з часом розширюватися, змінюватися, то замість переписування у клієнтському коді тіла оператора switch краще представити змінну множину варіантів класами.

Нехай маємо наступний фрагмент програмного коду

```
int type;     //type = ID1,ID2,ID3, ...
. . .
type = ID2;
. . .
switch (type)
  {
  case ID1:
    DoSomething1();
    break;
  case ID2:
    DoSomething2();
    break;
  case ID3:
    DoSomething3();
    break;
  . . .
  }
```

Для того, щоб прибрати конструкцію switch(type), спочатку замість “type” оголосимо базовий клас ObjType і низку похідних класів

```
class ObjType
{
public:
  virtual void DoSomething1(void) = 0;
};

class ObjType1 : public ObjType
{
public:
  void DoSomething(void);
};

class ObjType2 : public ObjType
{
public:
  void DoSomething(void);
};

class ObjType3 : public ObjType
{
public:
  void DoSomething(void);
};
```

А тепер запишемо клієнтський код – вже без switch

```
ObjType* p;
. . .
p = new ObjType2;
. . .
p->DoSomething();        //поліморфізм замість switch
```

Приклад виконання рефакторингу

Нехай маємо наступний програмний код.
Клієнтська частина – використовуємо об'єкти класу MyClass:

```
MyClass mc;
char tmpstr[256];
//. . .
mc.readstring(tmpstr, "oldfile.log");
mc.method(tmpstr, OLD);
//. . .
mc.readstring(tmpstr, "currentfile.log");
mc.method(tmpstr, PRESENT);
//. . .
mc.readstring(tmpstr, "futurefile.log");
mc.method(tmpstr, FUTURE);
//. . .
```

Клас MyClass оголошується та визначається у файлах модуля module1
Файл module1.h

```
enum TimeType
{
   OLD = 1,
   PRESENT = 2,
   FUTURE = 3
};

class MyClass
{
private:
   int value;
public:
   int getvalue();
   void readstring(char *dest, char *fname);
   int method(char *src, int vartype);
};
```

Файл module1.cpp

```
#include "module1.h"

int MyClass::getvalue()
{
   return value;
}

void MyClass::readstring(char *dest, char *fname)
```

```
{
   //. . .
   strcpy(dest, "start\tmessage\tend");
   //. . .
}

int MyClass::method(char *src, int vartype)
{
   int res = 0;
   char *pos;

   if (src == 0) return 0;
   pos = strchr(src, '\t');
   if (pos)
      value = strlen(pos);
   //. . .
   switch (vartype)
      {
      case OLD:
         res = 10;
         break;
      case PRESENT:
         res = 100 + value;
         break;
      case FUTURE:
        res = 1000 + value;
        break;
      default : break;
      }
   //. . .
   return res;
}
```

Як здається, у метода **method** занадто великий текст. Крім того, конструкція на основі **switch** викликає сумніви у плані можливого розширення типів.

1-й крок рефакторингу: виділення метода (Extract Method) у програмному коді **method.** Частину коду виділимо в окрему функцію **count()**

```
class MyClass
{
private:
   int value;
   int count(int vartype);
public:
   int getvalue();
   void readstring(char *dest,char *fname);
   int method(char *src, int vartype);
};
```

```
int MyClass::count(int vartype)
{
   int res = 0;
   switch (vartype)
      {
      case OLD:
         res = 10;
         break;
   case PRESENT:
      res = 100 + value;
      break;
```

```
   case FUTURE:
      res = 1000 + value;
      break;
   default : break;
   }
return res;
}
```

```
int MyClass::method(char *src, int vartype)
{
int res;
char *pos;

if (src == 0) return 0;
pos = strchr(src, '\t');
if (pos)
   value = strlen(pos);
//. . .
res = count(vartype);
//. . .
return res;
}
```

2-й крок рефакторингу: заміна switch поліморфізмом. Оголосимо базовий клас

```
class TimeBase
{
public:
   virtual int count(int value) = 0;
};
```

Похідні класи

```
class OldTime : public TimeBase
{
public:
   int count(int value)
   {
      return 10;
   }
};
```

```
class PresentTime : public TimeBase
{
public:
   int count(int value)
   {
      return 100 + value;
   }
};
```

```
class FutureTime : public TimeBase
{
public:
   int count(int value)
   {
      return 1000 + value;
   }
};
```

Відповідно змінюємо **method** класу MyClass

```
class MyClass
{
private:
   int value;
public:
   int getvalue();
   void readstring(char *dest, char *fname);
   int method(char *src, TimeBase *tm);
};
```

```
int MyClass::method(char *src, TimeBase *tm)
{
   int res;
   char *pos;

   if (src == 0) return 0;
   pos = strchr(src, '\t');
   if (pos)
      value = strlen(pos);
   //. . .
   res = tm->count(value);
   delete tm;
   //. . .
   return res;
}
```

У коді клієнта класів **MyClass** і **XXXTime** буде використовуватися композиція цих класів

```
MyClass mc;
char tmpstr[256];
//. . .
mc.readstring(tmpstr, "oldfile.log");
mc.method(tmpstr, new OldTime);
//. . .
mc.readstring(tmpstr, "currentfile.log");
mc.method(tmpstr, new PresentTime);
//. . .
mc.readstring(tmpstr, "futurefile.log");
mc.method(tmpstr, new FutureTime);
//. . .
```

Як здається, рефакторинг можна продовжити. Так, наприклад, виглядає доцільним додаткове структурування з метою уникнення фактично дублювання коду для ланцюжків викликів пар методів readstring та method. Зокрема, якщо ці методи викликаються тільки парою, то можна їх перенести у закриті члени, а публічним методом класу MyClas зробити такий:

```
int MyClass::func(char *fname, TimeBase *tm)
{
   char tmpstr[256];
   mc.readstring(tmpstr, fname);
   return mc.method(tmpstr, tm);
}
```

Місце рефакторингу у процесі розробки програмного забезпечення

Типовий процес розробки програмного забезпечення можна представити як послідовність багатьох ітерацій, які можуть повторюватися

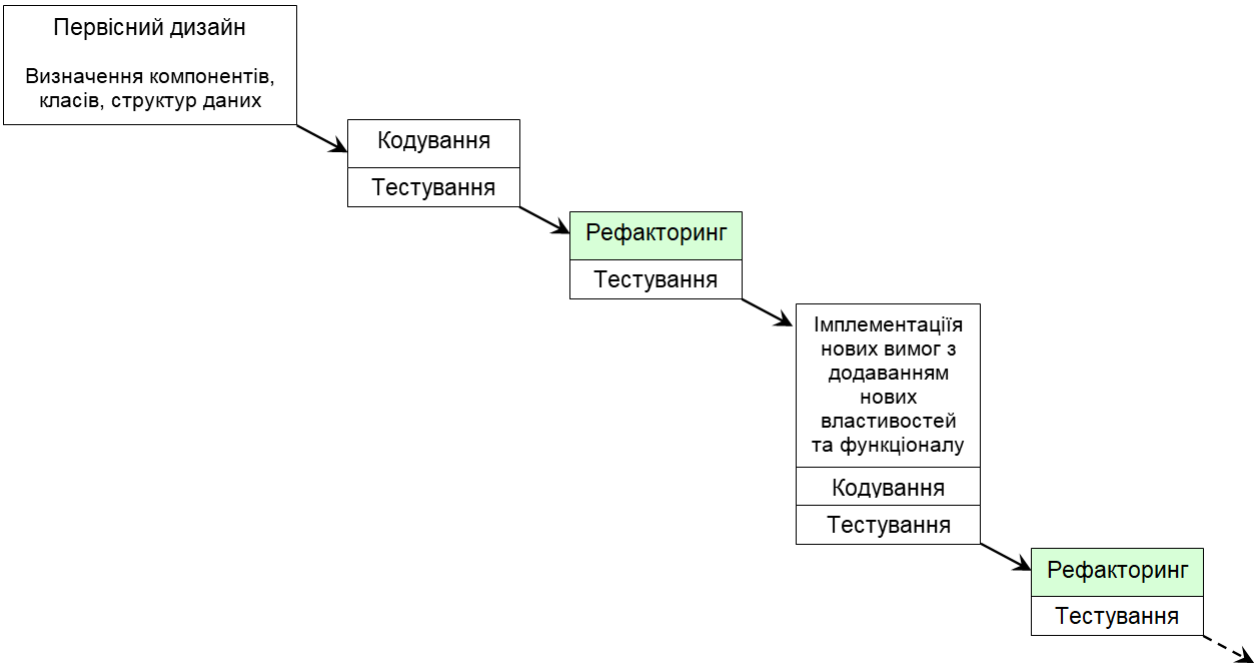


Рис. 18.2. Місце рефакторингу у процесі розробки програмного забезпечення

Необхідно відзначити, в процесі розробки програмного забезпечення на основі об'єктно-орієнтованого підходу зазвичай доволі часто доводиться виконувати рефакторинг, що ускладнює процес і може вважатися відчутним недоліком такого підходу до створення програмних систем. Також можна відзначити, що рефакторинг може виконуватися у різний час декілька разів, незважаючи на те, що, за визначенням рефакторинг означає покращення коду при незмінності вимог щодо властивостей та функціоналу системи.

Те, що існує потреба у частому рефакторингу при використанні ООП, може обумовлюватися не тільки особливостями власне об'єктно-орієнтованого підходу, а й значною мірою це може визначатися людським фактором в інженерії програмного забезпечення.

Контрольні запитання

1. Що таке рефакторинг?
2. Які ознаки поганого коду?
3. Які ви знаєте методи рефакторингу?
4. Коли треба замінювати оператор switch поліморфізмом?
5. Яким чином та як часто треба виконувати рефакторинг?

ПЕРЕЛІК ПОСИЛАНЬ

1. Об'єктно-орієнтоване програмування. Комп'ютерний практикум: Навч. посібник. [Електронний ресурс] / Уклад.: В.М. Порєв. – Електронні текстові дані (1 файл: 2,5 МБ). – Київ : КПІ ім. Ігоря Сікорського, 2022. – 105 с. Гриф надано Методичною радою КПІ ім. Ігоря Сікорського (протокол №1 від 02.09.2022р.) за поданням Вченої ради факультету інформатики та обчислювальної техніки (протокол №11 від 11.07.2022 р.)

2. Патерни проектування. REFACTORING GURU [Online] Available from: https://refactoring.guru/uk/design-patterns

3. Alan Shalloway, James R. Trott. Design Patterns Explained: A New Perspective on Object-Oriented Design. Addison-Wesley, 2nd edition, 2004, 468p.

4. Barbara Liskov. Data Abstraction and Hierarchy. SIGPLAN Notices, 23, 5 (May 1988)

5. Bjarne Stroustrup. Programming: Principles and Practice Using C++ (2nd Edition) [Online] Available from:https://dl.icdst.org/pdfs/files3/fef0590f02fa06bb42cba558fbc9e51c.pdf

6. C++ Standard Library including C++ 14 & C++ 17 [Online] Available from: https://www.educative.io/courses/cpp-standard-library-including-cpp-14-and-cpp-17

7. Charles Petzold. Programming Windows [Online] Available from: https://docs.google.com/file/d/0B73JwvIHVHaiSFdpekJCOUdoeE0/view?resourcekey=0-JFez95uS9jETJQdFGKY_1w

8. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, 1995, 368 p.

9. Gerbert Schildt. C++: A Beginner's Guide. McGrawHil, 2012, 542 p.

10. Gerbert Schildt. Java: A Beginner’s Guide. McGrawHil (Eighth Edition), 2018, 814 p.

11. Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts Refactoring: Improving the Design of Existing Code. Addison Wesley Professional, Second Edition, 2018, 448 p.

12. Martin Fowler. Inversion of Control Containers and the Dependency Injection pattern [Online] Available from: https://www.martinfowler.com/articles/injection.html#FormsOfDependencyInjection

13. Martin Fowler. UML Distilled: A Brief Guide to the Standard Object Modeling Language, 3rd Edition, 2003, 208 p.

14. Microsoft. Creating and Managing Visual C++ Projects [Online] Available from: https://msdn.microsoft.com/en-us/library/4457htyc.aspx

15. Microsoft. Local functions (C# Programming Guide) [Online] Available from: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/local-functions

16. Microsoft. Nested Class Declarations [Online] Available from: https://learn.microsoft.com/en-us/cpp/cpp/nested-class-declarations?view=msvc-170

17. Microsoft. Overview of Windows Programming in C++ [Online] Available from: https://docs.microsoft.com/en-us/cpp/windows/

18. OMG Standards Development Organization. Unified Modeling Language [Online] Available from: https://www.omg.org/spec/UML/

19. Robert C. Martin. Agile Software Development, Principles, Patterns, and Practices. 1st ed., Pearson, 2002 [Online] Available from: https://dl.ebooksworld.ir/motoman/Pearson.Agile.Software.Development.Principles.Patterns.and.Practices.www.EBooksWorld.ir.pdf

20. Robert C. Martin. Design Principles and Design Patterns. 2000 [Online] Available from: http://staff.cs.utu.fi/~jounsmed/doos_06/material/DesignPrinciplesAndPatterns.pdf

21. The Unified Modeling Language [Online] Available from: https://www.uml-diagrams.org/

22. Wikipedia. Common User Access [Online] Available from: https://en.wikipedia.org/wiki/IBM_Common_User_Access

23. Wikipedia. Expression problem [Online] Available from: https://en.wikipedia.org/wiki/Expression_problem

24. Working Draft, Standard for Programming Language C++. Document Number: N4910 Date: 2022-03-17 [Online] Available from: https://github.com/cplusplus/draft/releases/tag/n4910