МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
ІМЕНІ ІГОРЯ СІКОРСЬКОГО»
ІНСТИТУТ СПЕЦІАЛЬНОГО ЗВ'ЯЗКУ ТА ЗАХИСТУ ІНФОРМАЦІЇ

В.М. КУЛІКОВ, В.В. РЯБЦЕВ, С.С. ПАРШУКОВ

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ ДЛЯ ФАХІВЦІВ З КІБЕРБЕЗПЕКИ

Навчальний посібник

Рекомендовано Вченою радою ІСЗЗІ КПІ ім. Ігоря Сікорського
для використання у навчальному процесі з підготовки
фахівців першого (бакалаврського) рівня вищої освіти
за спеціальностями: 122 Комп'ютерні науки,
125 Кібербезпека та захист інформації,
172 Електронні комунікації та радіотехніка

Київ
ІСЗЗІ КПІ ім. Ігоря Сікорського
2023

УДК 004.432 (075.8)

К85

Розглянуто Вченою радою
ІСЗЗІ КПІ ім. Ігоря Сікорського
(Протокол № 8 від 26.01.2023 р.)

Рецензент: *В.В. Циганок, докт. тех. наук, с.н.с*

В.В. Соколов, канд. тех. наук, доц.

Куліков В.М., Рябцев В.В., Паршуков С.С.

К85 Об'єктно-орієнтоване програмування для фахівців з кібербезпеки: навч. посіб. / ІСЗЗІ КПІ ім. Ігоря Сікорського. Київ: КПІ ім. Ігоря Сікорського, 2023. 365 с.

ISBN 978-966-2577-19-8

Навчальний посібник містить матеріал, в який увійшли основи застосування об'єктно-орієнтованої технології для створення програм мовою програмування C++. Розробка програмних засобів, призначених для розв'язання завдань з кібербезпеки вимагає комплексного підходу. Тому в посібнику розглянуто засади об'єктно-орієнтованого аналізу і проектування мовою UML, основні можливості C++ та розширення мови за рахунок використання стандартної бібліотеки шаблонів STL. Матеріал супроводжується прикладами програм, які демонструють базові принципи об'єктно-орієнтованої технології і можуть бути основою для розробки слухачами власних програмних застосунків.

Навчальне видання призначене курсантам (студентам), які навчаються за спеціальностями 122 Комп'ютерні науки, 125 Кібербезпека та захист інформації, 172 Електронні комунікації та радіотехніка та вивчають нормативні навчальні дисципліни «Об'єктно-орієнтоване програмування», «Програмування» та «Інформатика». Викладені в посібнику загальні принципи об'єктно-орієнтованого підходу необхідні також при вивченні інших мов програмування, таких як Java, якій присвячена дисципліна «Крос-платформне програмування». Може бути також корисне всім, хто займається вивченням сучасних технологій розробки складних програмних проектів із застосуванням методів об'єктно-орієнтованого аналізу, проектування та програмування. Рекомендовано для використання при написанні дипломних та курсових проектів за відповідною тематикою.

УДК 004.432 (075.8)

ISBN 978-966-2577-19-8

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

ANSI – American National Standards Institute (американський національний інститут стандартів)
ISO – International Organization for Standardization (міжнародна організація по стандартизації)
LIFO – Last In, First Out (останнім прийшов – першим пішов)
FIFO – First In, First Out (першим прийшов – першим пішов)
OOA – Object-Oriented Analysis (об'єктно-орієнтований аналіз)
OOD – Object-Oriented Design (об'єктно-орієнтоване проектування)
OOP – Object-Oriented Programming (об'єктно-орієнтоване програмування)
STL – Standard Template Library (стандартна бібліотека шаблонів)
UML – Unified Modeling Language (уніфікована мова моделювання)
АТД – Абстрактний (користувальницький) тип даних
ООАП – Об'єктно-орієнтований аналіз та проектування
ООП – див. *ООР*

ПЕРЕДМОВА

Наукові дослідження в галузі удосконалення засобів та методів розробки програмного забезпечення часто спрямовані на намагання перевести працю програмістів із розряду творчості в розряд точних наук. Задача перетворення програмування з мистецтва в науку пов'язана, в першу чергу, з необхідністю створення технологічних процесів виготовлення програмних продуктів. Потреби розширення сфер застосування інформаційних технологій не можуть бути задоволені творчими здобутками лише програмістів-геніїв. Творче натхнення здатне породити шедеври програмістської культури, але не може завжди встигати за потребами науково-технічного прогресу.

Серед технологій створення програм значна увага приділяється технології об'єктно-орієнтованого програмування. Розповсюдження об'єктно-орієнтованих технологій пов'язане, перш за все, із намаганням подолати або суттєво зменшити існуючий і постійно зростаючий бар'єр складності програмного забезпечення. Об'єктна декомпозиція складної системи при її правильному розумінні і застосуванні дозволяє одному розробнику створювати більш складні програми і, одночасно, розширює можливості колективів програмістів спільно просувати великі проекти за рахунок переваг об'єктного моделювання і повторного використання коду.

Навчальний посібник призначений для підготовки та проведення занять з навчальних дисциплін «Об'єктно-орієнтоване програмування», «Програмування» та «Інформатика» для курсантів (студентів), які навчаються за спеціальностями 122 Комп'ютерні науки, 125 Кібербезпека та захист інформації та 172 Електронні комунікації та радіотехніка.

Предметом зазначених дисциплін є вивчення методології об'єктно-орієнтованого аналізу, проектування та програмування із реалізацією мовою Сі++.

Метою вивчення є формування у студентів (курсантів) здатностей:

– здійснювати проектування інформаційних систем (ІС) та інформаційних технологій (ІТ), включаючи формальний опис їх структури та проведення моделювання;

– використовувати сучасні комп'ютерні технології для системного, функціонального, конструкторського та технологічного проектування складних об'єктів і систем;

– застосовувати сучасні парадигми програмування під час програмної реалізації професійних задач;

– застосовувати професійні знання й уміння на практиці;

– вирішувати проблеми в професійній діяльності на основі аналізу й синтезу.

Згідно з вимогами освітньо-професійних програм студенти (курсанти) після засвоєння матеріалу дисциплін мають продемонструвати такі **знання**:

– інформаційних технологій, мов програмування, інструментарію програміста;

– сучасних технологій та інструментальних засобів розробки програмних систем;

– CASE-технологій проектування інформаційних та програмних систем;

– методів та стандартів оформлення документації;

– парадигм програмування, сучасних мов програмування, основних структур даних і алгоритмів.

Даний навчальний посібник відрізняється від інших аналогічних видань комплексним підходом до викладення матеріалу. Часто при вивченні об'єктно-орієнтованого програмування автори обмежуються вивченням мови програмування, такої як C++, Java або подібних їм. Автори даного видання вважають це недостатнім для фахівців з кібербезпеки. Надійний захист інформації може забезпечувати тільки комплексний підхід до створення і аналізу програмних засобів, призначених для розв'язання завдань з кібербезпеки. Тому, додатково в посібник включено матеріал з об'єктно-орієнтованого моделювання, мови UML та засобів стандартної бібліотеки шаблонів.

Позначення, які застосовуються в тексті:

Шрифти:

Bold – для заголовків та підзаголовків, а також елементів програмного коду, наприклад, **operator double()**;

Italic – для акцентуванні уваги на важливій інформації або ключових поняттях;

Italic Bold – для введення нових понять або визначень;

Arial – для представлення програмного коду.

Лапки:

« » – в українському тексті;

" " – в англійському тексті;

' ' – для позначення в тексті символів програмного коду, наприклад '**::**'.

1. ОСНОВИ ОБ’ЄКТНО-ОРІЄНТОВАНОГО АНАЛІЗУ, ПРОЕКТУВАННЯ ТА ПРОГРАМУВАННЯ

1.1. Поняття об'єктно-орієнтованого аналізу, проектування та програмування

1.1.1. Складність програмного забезпечення. Прості та складні програмні системи

Не всім програмним системам притаманна велика складність. Існує певна кількість програм, які було замислено і створено лише однією людиною. Як правило, такою людиною є програміст-початківець або професіонал, що працює ізольовано. Неправильно було б стверджувати, що всі такі програми створено погано, але майже завжди вони мають обмежену область застосування і нетривалий термін існування. Спільною рисою таких програм є те, що їх легше замінити новими, ніж намагатися переробити або покращити. Розробку таких програм точніше назвати стомливою ніж складною. Для їх створення частіше застосовуються традиційні процедурні (імперативні) мови програмування, такі як Бейсик, Паскаль, С.

Підходи, які розглядаються в даному посібнику, мають за мету підготовку студентів до створення програмних продуктів підвищеної складності. До них можна віднести:

– системи із зворотним зв’язком, які управляють або самі управляються подіями реального фізичного світу і для яких ресурси пам’яті та часу обмежені;

– задачі підтримки цілісності інформації об’ємом в сотні тисяч записів при паралельному доступі до неї для оновлення та вибірки;

– системи управління і контролю реальних процесів (наприклад, диспетчеризація повітряного або залізничного транспорту) та ін.

Подібні системи зазвичай мають великий час життя і значна кількість користувачів стає у залежність від їхнього правильного функціонування. Можна сказати, що складність таких програм перевищує можливості інтелекту людини. Один розробник практично неспроможний охопити всі аспекти такої системи.

Безумовно, і серед програмістів є генії, які здатні самотужки виконувати роботу групи пересічних розробників і досягти у цьому значних успіхів. Проте в програмуванні неможливо постійно сподіватися на надзвичайні можливості окремих обдарованих людей. Необхідно впроваджувати більш надійні способи конструювання складних програмних систем.

Наведені в навчальному посібнику приклади не можна назвати складними в даному розумінні. Вони показують лише окремі сторони вирішення складних задач і мають за мету дати уявлення про інструментарій, застосування якого дозволяє перетворити важку працю програміста у технологічний процес, що складається з окремих відносно простих і зрозумілих операцій.

Чим зумовлена складність програмного забезпечення? Складність програмного забезпечення не є випадковою. З одного боку її зростання слід розглядати, як об'єктивну закономірність прогресу в галузі інформаційних технологій (рисунок 1.1).

Рисунок 1.1 – Динаміка прогресу в галузі інформаційних технологій

Одночасно складність програмного забезпечення є наслідком таких причин:

- складність реальної предметної області (реального світу);
- складність управління процесом розробки;
- необхідність забезпечити достатню гнучкість програми;
- недосконаліть способів опису поведінки великих дискретних систем.

Складність реального світу. Проблеми, для вирішення яких застосовується програмне забезпечення, дуже часто містять складні елементи і тому до відповідних програм висуваються складні вимоги, які у сукупності є взаємовиключними. Прикладом може бути система стільникового зв'язку, яка

складається з багатьох підсистем. Вимоги із зручності, вартості, продуктивності та надійності неможливо в такій системі виконати одночасно і в повному обсязі.

Складність управління процесом розробки. Складність існуючого програмного забезпечення оцінюється сотнями тисяч і навіть мільйонами рядків команд мови високого рівня. Жодна людина не спроможна повністю зрозуміти таку систему. Навіть, якщо таку систему розкласти на складові частини, буде отримано сотні або тисячі окремих модулів. Тому, такий об'єм робіт потребує залучення команди розробників. Чим більше розробників, тим складнішими є зв'язки між ними і тим складнішою є координація, особливо, коли учасники робіт географічно віддалені один від одного. Підтримка єдності та цілісності розробки є досить складною задачею для керівництва при колективному виконанні проекту.

Гнучкість програмного забезпечення. Створення нового програмного продукту може бути виконано із залученням наявних компонентів або із повною розробкою всього необхідного «з нуля». Гнучкість програмного забезпечення означає можливість програміста самому забезпечити себе всіма необхідними елементами на всіх рівнях абстракції. При цьому, розробник власними силами створює всі базові «будівельні блоки» майбутньої конструкції, з яких утворюються елементи більш високих рівнів абстракції. Ситуація, коли будівельник має сам виготовляти і цеглу, і бетон, і всі інші будівельні матеріали, стосовно праці програмістів не вважається дивною. Тому програмні розробки залишаються дуже трудомісткою справою.

Проблема опису поведінки великих дискретних систем. Виконання програми здійснюється на цифровому комп'ютері. Тому програміст має справу із системою з дискретними станами. Дискретні системи природно мають кінцеву кількість можливих станів, яка відповідно до законів комбінаторики дуже велика. Будь яка подія може перевести дискретну систему у новий стан і перехід з одного стану в інший не завжди детермінований. При небажаних умовах зовнішня подія може порушити поточний стан системи внаслідок того, що її розробники не змогли передбачити всі можливі варіанти. Це є причиною обов'язкового тестування програмних систем, але справа в тому, що всеосяжне тестування складних програм провести неможливо саме внаслідок надмірної кількості можливих станів і переходів. В умовах, коли не існує ані математичних інструментів, ані інтелектуальних можливостей для повного моделювання

поведінки великих дискретних систем, розробники примушені задовольнятися розумним рівнем впевненості у їх правильності.

П’ять ознак складної системи

Аналіз досвіду кращих програмістів дозволяє виділити загальні ознаки складних систем, на виявленні яких необхідно зосереджувати увагу при створенні їх програмних моделей.

1. Складні системи часто є ***ієрархічними*** *і складаються із взаємопов’язаних підсистем, які в свою чергу також можна розділити на підсистеми, і т.д., до самого нижчого рівня.*

Важливо зрозуміти, з яких компонентів і з яких ієрархічних відношень цих компонентів складається структура складних систем. Всі системи мають підсистеми та всі системи є частинами більш великих систем. Зрозуміти, а значить і створити програмні моделі можливо лише для тих систем, які мають ієрархічну структуру.

2. Вибір того, які компоненти в даній системі вважаються елементарними, відносно довільний і у великій мірі залишається на розсуд дослідника.

Це дозволяє давати відповідь на питання стосовно того, що слід вважати простішими елементами системи. Найнижчий рівень для певного спостерігача може виявитися досить високим для іншого.

3. Внутрішні зв’язки компонентів, як правило, більш важливі (потужні) ніж зв’язки між самими компонентами.

Така відмінність внутрішніх зв’язків компонентів від зв’язків між компонентами обумовлює розподіл функцій між частинами системи і дає можливість вивчати кожну частину окремо.

4. Ієрархічні системи зазвичай складаються з небагатьох типів підсистем, які по-різному скомбіновано та організовано.

Інакше кажучи, різні складні системи містять однакові структурні частини. Ці частини можуть використовувати спільні більш дрібні компоненти чи більш великі структури.

5. Будь-яка працююча складна система є результатом розвитку більш простої системи, яка вже працювала раніше. Складна система, яку

спроектовано «з нуля», ніколи не запрацює. Необхідно починати з працюючої простої системи.

Складні системи мають тенденцію розвитку з часом. Об'єкти, які спочатку розглядались як складні, стають елементарними, і з них будуються складні системи. Більш того, неможливо відразу правильно створити елементарні об'єкти: з ними слід спочатку попрацювати, щоб більше з'ясувати про реальну поведінку системи, і лише потім удосконалювати їх. Складні системи розвиватимуться значно швидше, якщо для них існують сталі проміжні форми.

Наслідки необмеженої складності

Чим складнішою є система, тим легше її зруйнувати. Будівельника важко примусити погодитися розширити фундамент вже збудованого 100-поверхового будинку. Це не тільки дорого: робити такі речі – просто небезпечно! Але дивно те, що користувачі програмних систем, не замислюючись, ставлять подібні завдання перед їх розробниками. Це, за їх твердженням, усього лише технічне питання для програмістів.

Наслідком невміння створювати складні програмні системи є проекти, які виходять за рамки встановлених строків і бюджетів і до того ж не відповідають початковим вимогам. Це часто називають кризою програмного забезпечення, що, на жаль, тягнеться вже довго і стає нормою. Криза приводить до розбазарювання людських ресурсів – найдорогоціннішого товару – і до істотного обмеження можливостей створення нових продуктів. Зараз просто не вистачає гарних програмістів, щоб забезпечити всіх користувачів потрібними програмами. Більше того, істотний відсоток персоналу, зайнятого розробками, часто повинен займатися супроводом і збереженням застарілих програм. З урахуванням прямого й непрямого внеску індустрії програмного забезпечення в розвиток економіки більшості провідних країн не можна дозволити, щоб існуюча ситуація залишалася без змін.

Як можна змінити стан справ? Оскільки проблема виникає в результаті складності структури програмних продуктів, доцільно розглянути способи роботи зі складними структурами в інших областях.

Можна привести безліч прикладів успішно функціонуючих складних систем. Деякі з них створені людиною, наприклад: космічний корабель Space Shuttle, тунель під Ла-Маншем, великі фірми типу Microsoft або General Electric.

У природі існують ще більш складні системи, наприклад система кровообігу людини або екосистема планети.

Приклади складних систем

Структура персонального комп'ютера. Персональний комп'ютер (ПК) – прилад помірної складності. Більшість ПК складається з одних і тих самих основних елементів: системної плати, монітора, клавіатури й пристрою зовнішньої пам'яті якого-небудь типу (частіше жорсткого диску). Можна взяти кожну із цих частин і розкласти її у свою чергу на складові. Системна плата, наприклад, містить оперативну пам'ять, центральний процесор (ЦП) і шину, до якої підключені периферійні пристрої. Кожну із цих частин можна також розкласти на складові: ЦП складається з регістрів і схем управління, які самі складаються із ще більш простих деталей: діодів, транзисторів тощо.

Це приклад складної ієрархічної системи. Персональний комп'ютер нормально працює завдяки чіткому спільному функціонуванню всіх його складових частин. Разом ці частини утворюють логічне ціле. Інженер може зрозуміти, як працює комп'ютер, тільки тому, що він може розглядати окремо кожну його складову. Таким чином, можна вивчати устрій монітора й жорсткого диска незалежно один від одного. Аналогічно можна вивчати арифметичну частину ЦП, не розглядаючи при цьому підсистему пам'яті.

Справа не тільки в тому, що складна система ПК є ієрархічною, але ще й в тому, що рівні цієї ієрархії представляють різні рівні абстракції, причому один є надбудовою над іншим і кожний може бути розглянутий (зрозумілий) окремо. На кожному рівні абстракції можна знайти пристрої, які спільно забезпечують деякі функції більше високого рівня, і обрати рівень абстракції, виходячи з конкретних специфічних потреб. Наприклад, намагаючись дослідити проблему синхронізації звернень до пам'яті, можна залишатися на рівні логічних елементів.

Структура рослин і тварин. Ботанік намагається виконати класифікацію рослин вивчаючи їхню морфологію, тобто форму й структуру. Рослини – це складні багатоклітинні організми. У результаті спільної діяльності різних органів рослин відбуваються такі складні типи поводження, як фотосинтез й споживання вологи.

Рослина складається із трьох основних частин: коріння, стебла й листя. Кожна з них має свою особливу структуру. Коріння, наприклад, складається з

кореневих відростків, кореневих волосків, верхівки кореня й т.д. Розглядаючи зріз листя ми бачимо його епідерміс, мезофіл і судинну тканину. Кожна із цих структур, у свою чергу, являє собою набір клітин. Усередині кожної клітини можна виділити наступний рівень, що включає хлоропласт, ядро й т.д. Так само, як і у комп'ютера, частини рослини утворюють ієрархію, кожен рівень якої має власну незалежну складність.

Всі частини на одному рівні абстракції взаємодіють цілком однозначним чином. Наприклад, на вищому рівні абстракції, коріння відповідає за поглинання з грунту води й мінеральних речовин. Коріння взаємодіє зі стеблом, яке передає ці речовини листям. Листя, у свою чергу, використовує воду й мінеральні речовини, що доставляє стебло, і виробляє за допомогою фотосинтезу необхідні елементи.

Для кожного рівня абстракції завжди чітко розмежоване «зовнішнє» й «внутрішнє». Наприклад, можна встановити, що частини листя спільно забезпечують функціонування листя в цілому й дуже слабко взаємодіють або взагалі прямо не взаємодіють із елементами коріння. Простіше говорячи, існує чіткий поділ функцій різних рівнів абстракції.

В усіх частинах рослини можна знайти велику кількість «уніфікованих елементів». Так Природа досягає економії речовини та енергії. Наприклад, клітини є основними будівельними блоками всіх структур рослини: коріння, стебло й листя рослини складаються із клітин. І хоча кожної із цих вихідних елементів дійсно є клітиною, існує величезна кількість різноманітних клітин. Є клітини, що містять і не містять хлоропласт, клітини з оболонкою, що забезпечує проникнення або утримання води, і навіть живі й померлі клітини.

При вивченні морфології рослини дослідник не виділяє в ній окремі частини, відповідальні за окремі фази єдиного процесу, наприклад, фотосинтезу. Фактично не існує централізованих частин, які безпосередньо координують діяльність більше низьких (підлеглих) рівнів. Замість цього можна знайти окремі частини, що діють як незалежні посередники, і кожний з них поводиться досить складно й при цьому узгоджено з більш високими рівнями. Тільки завдяки спільним діям великої кількості посередників утворюється більш високий рівень функціонування рослини. Наука про складність називає це виникаючим поводженням. Поводження цілого складніше ніж поводження суми його складових.

Певні аналогії можна знайти і в зоології. Багатоклітинні тварини, як і рослини, мають ієрархічну структуру: клітини формують тканини, тканини працюють разом як органи, групи органів визначають систему (наприклад, кровообігу) і так далі. Тут необхідно знову відзначити властиву для Природи ощадливість: основний будівельний блок всіх рослин і тварин – клітина. Природно, що між клітинами рослин і тварин існують розходження. Клітини рослини, наприклад, укладено у тверду целюлозну оболонку на відміну від клітин тварин. Але, незважаючи на ці розходження, обидві зазначені структури, безсумнівно, є клітинами. Це приклад спільності в різних сферах.

Життя рослин і тварин підтримується значною кількістю механізмів надклітинного рівня, тобто більш високого рівня абстракції. І рослини, і тварини використовують судинну систему для транспортування в організмі поживних речовин. І в тих, і в інших можуть існувати статеві розходження в межах одного виду.

Структура речовини. Дослідження в таких різних областях, як астрономія і ядерна фізика, дають безліч інших прикладів неймовірно складних систем. У цих двох дисциплінах можна знайти приклади ієрархічних структур. Астрономи вивчають галактики, які об'єднані в скупчення, а зірки, планети й інші небесні тіла утворюють галактику. Ядерна фізика має справу зі структурною ієрархією фізичних тіл зовсім іншого масштабу. Атоми складаються з електронів, протонів і нейтронів; електрони, можливо, є елементарними частинками, але протони і нейтрони формуються з ще більш дрібних компонентів - кварків.

Знову виявляється спільність форм механізмів у цих складних ієрархіях. Насправді сучасна наука вважає, що у Всесвіті працюють усього чотири типи сил: гравітаційна, електромагнітна, сильна й слабка взаємодії. Багато законів фізики універсальні, наприклад, закон збереження енергії й імпульсу можна застосувати й до галактик, і до кварків.

Структура соціальних інститутів. Як останній приклад складних систем можна розглянути структуру суспільних інститутів. Люди об’єднуються в групи для виконання завдань, які вони не можуть виконати індивідуально. Деякі організації швидко розпадаються, інші функціонують протягом багатьох поколінь. Чим більша організація, тим чіткіше проглядається в ній ієрархічна структура. Транснаціональні корпорації складаються з компаній, які у свою чергу складаються з відділень, що містять різні філії. Останнім належать окремі офіси

й т.д. Границі між частинами організації можуть змінюватися і з часом може виникнути нова, більше стабільна ієрархія.

Відношення між різними частинами великої організації подібні до відношень між компонентами комп'ютера, рослини або галактики. Характерно, що ступінь взаємодії між співробітниками однієї установи значно вищий, ніж між співробітниками двох різних установ. Клерк, наприклад, звичайно не спілкується з виконавчим директором компанії, а в основному обслуговує відвідувачів. Але й тут різні рівні мають єдині механізми функціонування. Робота й клерка, й директора оплачується однією фінансовою організацією, і обидва вони для своїх цілей використовують загальну апаратуру, зокрема, телефонну систему компанії.

Канонічна форма складної системи

Виявлення загальних абстракцій і механізмів значно полегшує розуміння складних систем. Наприклад, досвідчений пілот, зорієнтувавшись усього за кілька хвилин, може взяти на себе керування багатомоторним реактивним літаком, на якому він раніше ніколи не літав, і спокійно його вести. Визначивши елементи, загальні для всіх подібних літаків (такі, як кермо керування, елерони й дросельний клапан), пілот потім знайде відмінності цього конкретного літака від інших. Якщо пілот уже знає, як управляти одним літаком певного типу, йому набагато легше навчитися управляти іншим схожим літаком.

Наведені приклади пояснюють термін «**ієрархія»** лише в досить приблизному розумінні. Найцікавіші складні системи містять багато різних ієрархій. У літаку, наприклад, можна виділити кілька систем: двигун, живлення, керування польотом, зв’язку, навігації тощо. Таке розділення дає *структурну ієрархію* типу **"the-part-of"** (бути частиною).

В той же час систему «двигун» можна розкласти зовсім в інший спосіб. Наприклад, турбореактивний двигун – особливий тип реактивного двигуна, а «РД-36-51» – особливий тип турбореактивного двигуна. З іншого боку, поняття «реактивний двигун» узагальнює властивості притаманні всім реактивним двигунам; «турбореактивний двигун» – це просто особливий тип реактивного двигуна із властивостями, які відрізняють його, наприклад, від прямоточного. Ця друга ієрархія являє собою ієрархію типу **"is-a"** (бути чимось).

Подібний підхід можна застосувати до розгляду будь-якої складної системи. Опис структури системи завжди залежить від точки зору, з якої ця

система розглядається. Виходячи з досвіду, визнано за необхідне розглядати систему із двох точок зору: як ієрархії першого й другого типу. Ієрархію типу **"is-a"** прийнято називати ***структурою (ієрархією) класів***, а тип **"the-part-of"** називають ***структурою (ієрархією) об'єктів***.

Треба відзначити, що складні програмні системи включають також й інші типи ієрархії. Особливе значення мають їхня *модульна структура*, що описує відношення між фізичними компонентами системи, і *ієрархія процесів*, що описує відношення між динамічними компонентами.

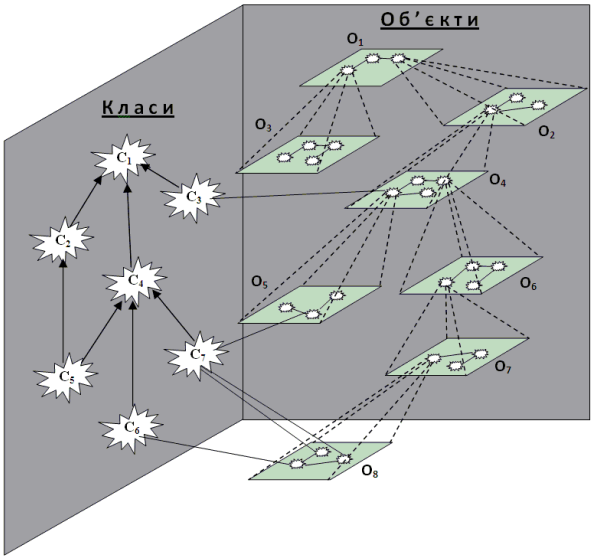


Рисунок 1.2 – Канонічна форма складної системи

Поєднуючи поняття *структури класів* і *структури об'єктів* з п'ятьма ознаками складних систем, можна дійти висновку, що фактично всі складні системи можна представити однією й тією ж (канонічної) формою, що показана на рис. 1.2. Тут наведені дві ортогональні ієрархії однієї системи: *класів* й *об'єктів*. Кожна ієрархія є багаторівневою, крім того в обох випадках класи й об'єкти більш високого рівня побудовано з більш простих. Який клас або об'єкт обрано у якості елементарного, залежить від конкретної задачі. Компоненти одного рівня мають чітко виражені зв'язки, особливо це стосується компонентів структури об'єктів. В межах будь-якого розглянутого рівня перебуває наступний рівень складності. Відзначимо також, що структури класів й об'єктів не є незалежними: кожен елемент структури об'єктів представляє специфічний екземпляр певного класу. Як можна бачити на рис. 1.2, об'єктів у складній системі

зазвичай набагато більше, ніж класів. Наявність двох ієрархій дозволяє уникнути *надлишковості* структури складної системи. Якщо ієрархія класів системи відсутня, то довелося б повторювати одні й ті самі відомості для кожного екземпляру класу (об’єкту). Із введенням структури класів в ній розміщуються загальні властивості екземплярів (об’єктів).

Існуючий досвід показує, що найбільш успішними є ті програмні системи, у які закладено ретельно продумані структури класів й об'єктів і яким властиві п'ять ознак складних систем. Іншими словами, оцінюючи важливість цього спостереження можна висловитися більш категорично: дуже рідко можна зустріти програмну систему, розроблену точно за графіком, із дотриманням бюджету, із задоволенням усіх вимог замовника і у якій би не було враховано міркування, що викладені вище.

Структури класів й об'єктів системи разом утворюють *архітектуру складної системи*. Розробка архітектури системи у формі ієрархії класів та об’єктів виконується в рамках ***об’єктного підходу*** до створення складних програмних систем. Застосування об’єктного підходу дозволяє значно підвищити поріг складності реальних систем, для яких можна побудувати адекватні програмні моделі.

Людські можливості й складні системи

Якщо відомо, як проектувати складні програмні системи, то чому на шляху створення таких систем розробників чекають серйозні перешкоди? Відносно новий *об'єктний підхід*, як доведено досвідом, не може вирішити всіх проблем. Існує ще одна, очевидно, головна причина: фізична обмеженість можливостей людини при роботі зі складними системами.

Аналізуючи складну програмну систему, у ній можна виявити багато складових частин, які взаємодіють різними способами. При цьому, ні самі частини системи, ні способи їхньої взаємодії не виявляють ніякої подібності. Це приклад неорганізованої складності. Коли розробник починає організовувати систему в процесі її проектування, йому необхідно думати відразу багато про що. Наприклад, у системі керування рухом літаків доводиться одночасно контролювати стан багатьох літальних апаратів, з огляду на такі їхні параметри, як місце розташування, швидкість і курс. При аналізі дискретних систем необхідно розглядати великі, складні й не завжди детерміновані простори станів.

На жаль, одна людина не може стежити за всім цим одночасно. Експерименти психологів показують, що максимальна кількість структурних одиниць інформації, за яких людський мозок може одночасно стежити, приблизно дорівнює семи плюс-мінус два. Ймовірно, це пов'язано з обсягом короткострокової пам'яті в людини. Додатковим обмежуючим фактором є швидкість обробки мозком людини вхідної інформації: на сприйняття кожної нової одиниці інформації йому потрібно близько 5 секунд.

Таким чином, виявляється серйозна дилема. Складність програмних систем зростає, але здатність мозку людини впоратися із цією складністю обмежена. Сучасна теорія і практика програмування не стоїть на місці і постійно розвивається в напрямку вирішення цієї проблеми. В даному посібнику розглянуто *об'єктний підхід*, як сукупність методологій *аналізу*, *проектування* та *програмування* при розробці складних програмних систем.

Висновки

Програмам властива складність, яка часто перевершує можливості людського розуму.

Складні структури часто приймають форму ієрархій: класів і об'єктів.

Складні системи звичайно створюються на основі стійких проміжних форм.

Пізнавальні здатності людини обмежені, але їхні рамки можна розсунути, використовуючи декомпозицію, виділення абстракцій і створення ієрархій.

Складні системи можна досліджувати, концентруючи основну увагу або на об'єктах, або на процесах; існують вагомі підстави використовувати *об'єктно-орієнтовану декомпозицію*, при якій світ розглядається як упорядкована сукупність об'єктів, взаємодія яких один з одним визначає поведінку системи.

Об'єктно-орієнтований підхід – метод, що використовує об'єктну декомпозицію; об'єктно-орієнтований підхід пропонує багатий набір логічних і фізичних моделей, за допомогою яких можна отримати уяву про різні аспекти досліджуваної системи.

1.1.2. Абстрактні моделі, що лежать в основі мов програмування

Інженерна справа як наука та мистецтво

Будь-яка інженерна дисципліна, чи то будівництво, механіка, хімія, електроніка або програмування, містить у собі елементи як науки, так і мистецтва. В процесі розробки абсолютно нової системи важливість інженера, як митця, висувається на перший план. При проектуванні програм це відбувається практично завжди. Тим більше при роботі з системами із зворотним зв'язком, і особливо, коли це стосується систем управління і контролю, коли інженерові доводиться розробляти програмне забезпечення, вимоги до якого не є стандартними. В інших випадках вимоги до системи можуть бути достатньо повно і точно визначені, але часто вони визначені таким чином, що відповідний їм технічний рівень розробки виходить за межі існуючих технологій. Прикладом цього є створення засобів для проведення наукових досліджень або інструментарію для досліджень в галузі штучного інтелекту. Цілком природним вважається вимагати створити таку систему, яка б мала набагато кращу швидкодію, ємність або функціональні можливості в порівнянні із уже існуючими.

В усіх цих випадках слід намагатися використовувати знайомі абстракції й механізми («стійкі проміжні форми»), як основу нової системи. При наявності великої бібліотеки повторно використовуваних програмних компонентів, інженер-програміст повинен їх по-новому скомпонувати, щоб задовольнити всім явним і неявним вимогам до системи, так само, як музикант знаходить нові форми застосування можливостей свого інструмента. Але на практиці подібних багатих бібліотек не існує, програміст звичайно може використати, на жаль, лише відносно невелику кількість готових модулів.

Зміст проектування

У будь-якій інженерно-технічній дисципліні під проектуванням звичайно розуміється деякий уніфікований підхід, за допомогою якого можна знайти шляхи вирішення певної проблеми, забезпечуючи виконання поставленого завдання. У контексті інженерного проектування можна визначити мету проектування, як створення системи, що:

– задовольняє заданим (можливо, неформальним) функціональним специфікаціям;

– узгоджена із обмеженнями з боку технічного устаткування;

– задовольняє явним і неявним вимогам до експлуатаційних якостей системи і споживання нею ресурсів;

– задовольняє явним і неявним критеріям дизайну;

– задовольняє вимогам до самого процесу розробки, таким, наприклад, як тривалість і вартість, а також залучення додаткових інструментальних засобів.

Згідно із припущенням Страуструпа (розробника мови C++): «Мета проектування – виявлення ясної й відносно простої внутрішньої структури, яку іноді називають архітектурою... Проект є остаточним продуктом процесу проектування». Проектування передбачає врахування суперечливих вимог. Його продуктами є ***моделі***, що дозволяють нам зрозуміти структуру майбутньої системи, збалансувати вимоги й намітити схему реалізації.

Важливість побудови моделі

Моделювання розповсюджене в усіх інженерних дисциплінах, у значній мірі через те, що воно реалізує принципи *декомпозиції*, *абстракції* й *ієрархії*. Кожна модель описує певну частину розглянутої системи, що дає можливість будувати нові моделі на базі старих, вже перевірених практикою і часом. Моделі дозволяють запобігати невдач. Для цього треба оцінити поводження кожної моделі у звичайних і незвичайних ситуаціях, а потім провести необхідні доробки.

Як вже було сказано вище, щоб зрозуміти у всіх тонкощах поводження складної системи, доводиться використати не одну модель. Наприклад, проектуючи комп'ютер на одній платі, інженер-електронник повинен розглядати систему як на рівні її окремих елементів (мікросхем), так і на рівні всієї схеми. Схема допомагає інженерові розібратися в спільному поводженні мікросхем. Схема являє собою план фізичної реалізації системи мікросхем, у якому враховано розмір плати, споживана потужність і типи наявних інтегральних мікросхем. Із цього погляду інженер може незалежно оцінювати такі параметри системи, як температурний режим і технологічність виготовлення. Проектувальник плати може також розглядати *динамічні* й *статичні* особливості системи. Аналогічно, інженер-електронник використовує діаграми, що ілюструють статичні зв'язки між різними мікросхемами, і часові діаграми, що відображають поводження елементів у часі. Потім інженер може застосувати

осцилограф або цифровий аналізатор для перевірки правильності і статичної, і динамічної моделей.

Елементи програмного проектування

Звичайно, не існує такого універсального методу, який би дозволив інженеру-програмісту виконати всі вимоги до складної програмної системи. Проектування складної програмної системи аж ніяк не зводиться до сліпого виконання якогось набору стандартних рецептів. Скоріше це поступовий та ітеративний процес. Проте, використання певної *методології проектування* дозволяє вносити у процес розробки елементи організованості. Інженери-програмісти розробили десятки різних методів, які можна класифікувати за трьома категоріями:

– *умовні позначки* – мова для опису кожної моделі;

– *процес* – правила проектування моделі;

– *інструменти* – засоби, які прискорюють процес створення моделей, і в які вже втілені закони функціонування моделей. Інструменти допомагають виявляти помилки в процесі розробки.

Сучасні методи проектування базуються на міцній теоретичній основі й при цьому залишають програмістові значну свободу для вільного самовираження.

Об’єктно-орієнтовані моделі

Чи існує найкращий метод проектування? На це питання немає однозначної відповіді. По суті воно аналогічне питанню: «Чи існує кращий спосіб декомпозиції складної системи?» Якщо й існує, то поки він нікому не відомий. Це питання можна поставити і в такий спосіб: «Як найкращим способом розділити складну систему на підсистеми?» Практика розробки моделей складних систем свідчить, що корисніше всього створювати такі моделі, які фокусують увагу на об'єктах, знайдених у самій предметній області, і утворюють так звану *об’єктно-орієнтовану декомпозицію*.

Об’єктно-орієнтований аналіз і *проектування* – це метод, що логічно приводить до *об’єктно-орієнтованої декомпозиції*. Застосовуючи об’єктно-орієнтоване проектування, можна створити гнучкі програми із заощадженням часу та інших ресурсів. При розумному розподілі простору станів можна досягти

більшої впевненості в правильності розробленої програми. У підсумку, зменшується ризик невиконання поставлених вимог при розробці складних програмних систем.

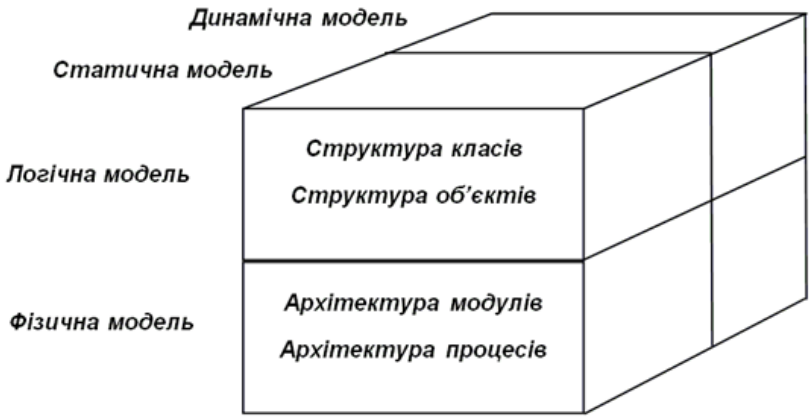


Рисунок 1.3 – Об’єктно-орієнтовані моделі

Об’єктно-орієнтоване проектування пропонує багатий вибір моделей, які представлені на рис. 1.3. Об’єктно-орієнтовані моделі проектування відображають ієрархію і класів, і об'єктів системи. Ці моделі охоплюють весь спектр найважливіших конструкторських рішень, які необхідно розглядати при розробці складної системи, і в такий спосіб полегшують створення проектів, яким властиві всі п'ять атрибутів добре організованих складних систем.

Абстрактні моделі, що лежать в основі мов програмування

Мови програмування – це завжди деякі моделі (віртуальні обчислювальні машини), що дозволяють найефективніше використовувати можливості обчислювальних засобів, важливі для конкретних областей застосування. Програміст при написанні програм орієнтується не на обчислювальну машину, а на деяку абстрактну модель обчислювального пристрою. Якість абстрактної моделі, її відповідність змісту вирішуваних задач багато в чому визначає якість і ефективність проектування з використанням цієї моделі.

Складність завдань, які можливо вирішити із застосуванням тих або інших концепцій проектування (тобто, мов, що виражають їх), безпосередньо пов'язана з рівнем абстракції, як при постановці задачі, так і в ході її вирішення. Абстракція є, ймовірно, наймогутнішим інтелектуальним засобом пізнання з тих, що є у розпорядженні людини. Із здатністю до абстрактного мислення пов'язана і

здібність людини до творчості. Крім того, виявлення загальних абстракцій і механізмів значно полегшує розуміння складних систем.

Так, мови асемблеру дозволяють уникнути необхідності програмування в машинному коді, і в такому розумінні асемблери є абстракцією обчислювальної машини, для якої їх було спроектовано. Доречі, і сама обчислювальна машина, яка є ієрархічною системою, може розглядатися на різних рівнях абстракції. Продовжуючи цю тезу, можна стверджувати, що мови, засновані на методології процедурного програмування, можна вважати абстракціями асемблерів.

Таким чином, мова програмування – це не тільки виразник концепцій проектування. Мова програмування є також виразником рівня абстракції технічних засобів, використовуваних для вирішення завдань. Розвиток мов програмування – це, перш за все, розвиток абстрактних моделей, що полегшують і систематизують проектування.

Абстрагування в мовах імперативного програмування

Імперативне програмування є першою методологією програмування, яка підтримується на апаратному рівні й орієнтована на ЕОМ із архітектурою фон Неймана. Методологія структурного імперативного програмування втілює підхід, що характеризується принципом послідовно-покрокової зміни стану комп'ютера, і який, крім цього, підтримує концепцію структурного програмування.

Прикладами мов виразників концепцій структурного імперативного програмування можуть служити мови Fortran, Pascal, C, PL/1. Всі ці мови базуються на парадигмі процедурного програмування, заснованої на поданні програми у вигляді ієрархії процедур і функцій.

Слід зазначити, що поняття процедурного й структурного програмування не слід ототожнювати, оскільки використання процедурної моделі сумісно й з об'єктно-орієнтованим програмуванням, і з функціональним програмуванням, і з паралельним програмуванням. Розвиток структурного програмування дав можливість реалізувати або удосконалити низку важливих методів проектування:

- функціонально-ієрархічна декомпозиція;
- структурна організація даних;
- повторне використання проектних рішень;
- технологія тестування програмного забезпечення.

Основна проблема процедурних мов полягає в тому, що рівень досягнутої ними абстракції завжди вимагає від програміста мислення в більшій мірі в термінах обчислювальної машини (нехай навіть і віртуальної), ніж у термінах задачі, яку треба вирішувати. Якість проектування визначається в підсумку тим, наскільки вдало програмістові вдалося встановити відповідність між поняттями, характерними для розв'язуваного завдання, і наявними засобами мови програмування. В результаті не завжди вдається адекватно відобразити ці поняття в рамках машинної моделі. Абстрагування, що досягається за допомогою використання процедур, добре підходить для опису абстрактних дій, але не надає адекватних мовних засобів для опису абстрактних об'єктів. До інших недоліків процедурного підходу можна віднести:

– архітектура програм непристосована до внесення змін, які виникають при еволюціонуванні складної системи;

– розроблені програми досить важко узагальнювати для багаторазового використання.

Висновок: засновані на процедурній парадигмі мови імперативного програмування реалізують систематичний підхід до розробки програмного забезпечення, який у багатьох випадках заснований на концепції структурного програмування. При цьому, завдання програміста – перейти від уявлення про розв'язуване завдання в термінах, властивих цьому завданню, до термінів, властивих моделі обчислювальної машини.

Інші абстрактні моделі

В історичному плані розвиток абстрактних моделей не завжди означав перехід на більш високий рівень абстракції подання обчислювальної машини, як виконавчого пристрою. Специфічні особливості деяких задач сприяли появі альтернативних моделей.

Прикладами альтернатив моделюванню обчислювальної машини можуть служити концепції, реалізовані в мовах *функціонального* й *логічного* програмування. Прикладом мови функціонального програмування може служити мова LISP, яка заснована на моделі, що у компактній формі характеризується постулатом: «всі завдання, в підсумку, можуть бути зведені до роботи зі списками». Класичним прикладом мови логічного програмування є мова Prolog, що ґрунтується на принципі: «всі завдання можуть бути зведені до ланцюжка

логічних висновків». Найчастіше ці мови знаходять застосування в задачах штучного інтелекту і прикладної математики.

Кожний з подібних підходів є прийнятним або навіть кращим тільки для певного класу задач. У багатьох інших випадках більш успішними виявлялися мови загального призначення. Навіть в області штучного інтелекту багато комерційних зразків систем, заснованих на знаннях, було розроблено із використанням універсальних мов.

Для вирішення завдань логічного управління активно розвивається підхід, заснований на використанні теорії автоматів – «автоматне програмування». Моделі представлення даних і поводження системи, які покладено в основу цього підходу, зосереджено на абстракціях *стану* (який описує статичні аспекти об'єкта управління) і *події* (яка визначає динаміку процесу управління). Автоматне програмування знаходить застосування не тільки при вирішенні завдань управління, але і в таких областях, як синтаксичний аналіз і трансляція, моделювання систем управління, проектування програмованих логічних контролерів та ін.

Наведені приклади показують, що розвиток абстрактних моделей пов'язаний не тільки з переходом на вищі рівні абстракції обчислювальної машини, а також і з розвитком інших концепцій абстрагування, орієнтованих на різні класи задач.

Парадигми програмування

У більшості випадків програмісти використовують в роботі тільки одну мову програмування і слідують тільки одному *стилю*. Вони програмують в парадигмі, яку нав'язано ним використовуваною мовою. Часто вони залишають осторонь альтернативні підходи до мети, а отже, їм важко побачити переваги стилю, який у більшій мірі відповідає умовам розв'язуваного завдання.

Стиль програмування – це спосіб побудови програм, заснований на певних принципах програмування, і виборі відповідної мови, яка робить зрозумілими програми, написані в цьому стилі. Відомо п'ять основних різновидів стилів програмування. Їх наведено нижче разом з властивими для них видами абстракцій:

– процедурно-орієнтований – алгоритми;
– об'єктно-орієнтований – класи і об'єкти;

– логіко-орієнтований – цілі, часто виражені в термінах числення предикатів;

– орієнтований на правила – правила «якщо-то»;

– орієнтований на обмеження – інваріантні співвідношення.

Неможливо визнати будь-який стиль програмування найкращим у всіх областях практичного застосування. Наприклад, для проектування баз знань більш придатним є стиль, орієнтований на правила, а для розрахункових задач – процедурно-орієнтований. Проте досвід показує, що об'єктно-орієнтований стиль є найбільш прийнятним для найширшого кола додатків; дійсно, ця парадигма часто служить архітектурним фундаментом, на якому засновано інші парадигми.

Кожен стиль програмування має свою концептуальну базу. Кожен стиль вимагає свого умонастрою й способу сприйняття розв'язуваного завдання. Для об’єктно-орієнтованого стилю концептуальна база – це *об'єктна модель*.

1.1.3. Сутність об'єктно-орієнтованого підходу

Сутність об’єктно-орієнтованого підходу полягає, насамперед, у двох моментах. З одного боку, він надає розробникові інструмент, що дозволяє описати завдання й істотну частину реалізації проекту в термінах, що характеризують проблемну (предметну) область, а не комп'ютерну модель. Об’єктно-орієнтований аналіз починається з дослідження предметів реального світу, що є частиною розв'язуваного завдання. Хоча застосування об’єктно-орієнтованої парадигми саме по собі не забезпечує якості проектування, вона сприяє ясному поданню даних і притаманних їм властивостей і дій (поводження) на рівні програмного коду. При цьому програмні структури, які представлено формальною мовою, відповідають моделі опису завдання й способів її рішення мовою, природною для даної предметної області. Ця особливість відрізняє об’єктно-орієнтовані мови від мов імперативного (процедурного) програмування, націлених на моделювання дій, виконуваних обчислювальною машиною (див. рис. 1.4).

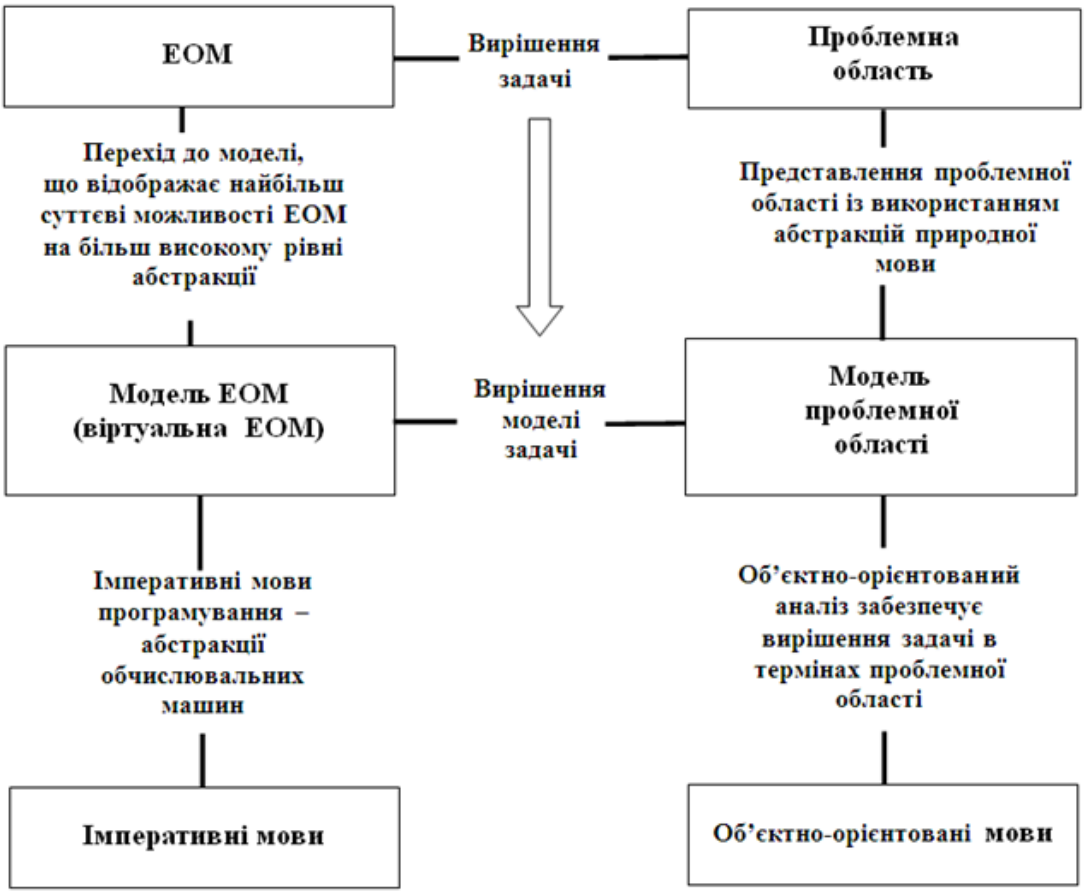


Рисунок 1.4 – Імперативні та об’єктно-орієнтовані мови

З іншого боку, об’єктно-орієнтоване проектування надає розробникам гнучкий потужний універсальний інструмент, не пов'язаний з якимось певним класом задач. По зауваженню Буча (розробника UML), «об'єктний підхід зарекомендував себе як уніфікована ідея всієї комп'ютерної науки, що може бути застосована не тільки в програмуванні, але також у проектуванні інтерфейсу користувача, баз даних і навіть архітектури комп'ютерів».

При цьому використання об’єктно-орієнтованого підходу виправдовує себе не для всіх класів задач. Наприклад, для реалізації основних обчислювальних алгоритмів застосування об’єктно-орієнтованого програмування навряд чи дозволяє одержати більше наочний і більше ефективний код, хоча, зрозуміло, ці завдання можна використати як приклади, що ілюструє відмінності в керуванні даними в процедурних й об’єктно-орієнтованих мовах.

У багатьох випадках найкращі результати проектування забезпечуються спільним застосуванням декількох моделей програмування. Так мова C++ і створювалася як мова, що підтримує декілька парадигм проектування, надаючи

програмістам потужний і гнучкий механізм для конструювання абстрактних типів, що підходить для більшості сучасних обчислювальних машин. C++ підтримує багато стилів програмування й поєднує ефективність і гнучкість мови C для системного програмування з можливостями об’єктно-орієнтованої організації коду й даних. Таким чином, багато фахівців погоджуються з тим, що переваги об’єктно-орієнтованої технології можна реалізувати швидше й у більше повному обсязі, якщо використовувати її у сполученні із традиційними методами.

Як складова багатьох відомих технологій проектування, об’єктно-орієнтована модель програмування підтримується багатьма сучасними мовами. До найбільш відомих об’єктно-орієнтованих мов програмування можна віднести мови Simula-67, Ada, Smalltalk, Object Pascal, C++, Eiffel, Java, C#, та ін.

Можливість працювати з об’єктами та класами підтримується навіть такими традиційно «необ’єктними» мовами, як Prolog та Lisp.

Роль декомпозиції

Спосіб керування складними системами був відомий ще в стародавності – "divide et impera" (розділяй і пануй). При проектуванні складної програмної системи необхідно розділяти її на все менші й менші підсистеми, кожну з яких можна вдосконалювати окремо. У цьому випадку не буде перевищено пропускної здатності людського мозку: для розуміння будь-якого рівня системи необхідно одночасно утримувати у розумі інформацію лише про деякі її частини (аж ніяк не про всі).

Алгоритмічна декомпозиція. Більшість з програмістів формально навчено структурному проектуванню «зверху вниз», і вони сприймають декомпозицію як звичайний розподіл алгоритмів, де кожен модуль системи виконує один з етапів загального процесу.

Об’єктно-орієнтована декомпозиція. Проте існує альтернативний спосіб декомпозиції. У якості критерію декомпозиції можна обрати належність елементів системи до різних абстракцій даної проблемної області. Хоча обидві схеми декомпозиції вирішують одне завдання, вони роблять це різними способами. У другій (об’єктно-орієнтованій) декомпозиції світ представлений сукупністю автономних діючих об’єктів, які взаємодіють один з одним, щоб забезпечити поводження системи, що відповідає більше високому рівню абстракції. Таким чином, кожен об'єкт має своє власне поводження, і кожний з

них моделює деякий об'єкт реального світу. Об'єкти вміють щось робити, і можна, пославши їм повідомлення, примусити їх виконати щось конкретне. Така декомпозиція заснована на об'єктах, а не на алгоритмах, і вона називається ***об'єктно-орієнтованою декомпозицією***.

Декомпозиція: алгоритмічна або об'єктно-орієнтована? Яка декомпозиція складної системи є більш правильною – по алгоритмах чи по об'єктах? Правильна відповідь на це питання: важливі обидва аспекти. Поділ по алгоритмах зосереджує увагу на порядку подій, що відбуваються, а поділ по об'єктах надає особливого значення агентам, які є або об'єктами, або суб'єктами дії. Однак вважається, що неможливо сконструювати складну систему одночасно двома способами, тим більше, що ці способи по суті є ортогональними (ця ортогональність вивчалася ще з давніх часів стародавніми греками. Пасивний погляд пропонувався Демокрітом, який стверджував, що світ складається з атомів. Ця позиція Демокрита ставила в центр усього матерію. Класичним представником іншої сторони – активного погляду – був Геракліт, що виділяв поняття процесу). Поділ системи необхідно починати або по алгоритмах, або по об'єктах, а потім, використовуючи отриману структуру, спробувати розглянути проблему з іншого погляду.

Досвід свідчить, що доцільніше розпочинати з об'єктної декомпозиції. Такий початок допоможе краще впоратися з доданням організованості складності програмних систем. Цей об'єктний підхід допомагає при описі таких несхожих систем, як комп'ютери, рослини, галактики й суспільні інститути. Об'єктна декомпозиція зменшує розмір програмних систем за рахунок повторного використання загальних механізмів, що призводить до істотної економії виразних засобів. Об'єктно-орієнтовані системи більше гнучкі й простіше еволюціонують з часом, тому що їхні схеми базуються на стійких проміжних формах. Об'єктна декомпозиція істотно знижує ризик при створенні складної програмної системи, тому що вона розвивається з менших систем, які вже перевірено. Більше того, об'єктна декомпозиція допомагає розібратися в складній програмній системі, пропонуючи розумні рішення щодо вибору підпростору великого простору станів.

Основні положення об’єктно-орієнтованого підходу

Методи структурного (алгоритмічного, процедурного, імперативного) проектування допомагають спростити процес розробки складних систем за рахунок використання алгоритмів як готових будівельних блоків. Аналогічно, методи об’єктно-орієнтованого проектування створені, щоб допомагати розробникам застосовувати потужні виразні засоби об’єктно-орієнтованого програмування, що використовують у якості таких блоків класи і об'єкти.

Але в об'єктній моделі відображається також множина інших факторів. Об'єктний підхід зарекомендував себе як уніфікована ідея всієї комп'ютерної науки і знаходить застосування не тільки в програмуванні, але також у проектуванні інтерфейсу користувача, баз даних і навіть архітектури комп'ютерів. Причина такого поширення в тому, що орієнтація на об'єкти дозволяє впоратися зі складністю систем різноманітної природи.

Об’єктно-орієнтований аналіз і проектування відбивають еволюційний, а не революційний розвиток проектування; нова методологія не пориває з колишніми методами, а будується з урахуванням попереднього досвіду. На жаль, більшість програмістів у цей час формально й неформально натреновані на застосування тільки методів структурного проектування. Зрозуміло, багато гарних проектувальників створили й продовжують удосконалювати велику кількість програмних систем на основі цієї методології. Однак алгоритмічна декомпозиція допомагає тільки до певної межі, і звертання до об’єктно-орієнтованої декомпозиції є необхідним. Більше того, при спробах використання таких мов, як C++ або Ada, у якості традиційних, алгоритмічно (процедурно) орієнтованих, не тільки губиться їхній внутрішній потенціал – скоріше за все результат буде навіть гіршим, ніж при використанні звичайних мов C і Pascal. Дати електродриль теслі, який і не чув про неї, означатиме використовувати її, як молоток. Він зігне кілька цвяхів і розіб'є собі пальці, тому що електродриль мало придатний для заміни молотка.

OOP, OOD і OOA

Успадкувавши від багатьох попередників, об'єктний підхід, на жаль, перейняв і заплутану термінологію. Програміст в Smalltalk користується терміном *метод*, в C++ – терміном *віртуальна функція*, в CLOS – *узагальнена функція*.

В Object Pascal використовується термін *приведення типів*, а в мові Ada те ж саме називається *перетворенням типів*. Щоб зменшити плутанину, варто визначити, що є об’єктно-орієнтованим, а що – ні.

Поняття «об'єкт» є центральним у всьому, що стосується об’єктно-орієнтованої методології. Поняття «об'єкт» можна визначити як окрему сутність, що чітко проявляє своє поводження. Або як сутності, що поєднують процедури й дані, тому що вони роблять обчислення й зберігають свій локальний стан. Визначення об'єкта як *сутності* якоюсь мірою відповідає дійсності, але все ж-таки головним у даному понятті є об'єднання ідей *абстракції даних* й *алгоритмів*. В об'єктному підході акцент перенесено на конкретні характеристики фізичної або абстрактної системи, яка є предметом програмного моделювання. Об'єкти мають цілісність, що не повинна – а, насправді, – не може бути порушена. Об'єкт може тільки міняти стан, поводитися, управлятися, або ставатися в певне відношення до інших об'єктів. Інакше кажучи, властивості, які характеризують об'єкт і його поводження, залишаються незмінними. Наприклад, ліфт характеризується такими незмінними властивостями: він може рухатися вгору й вниз, залишаючись у межах шахти. Будь-яка модель повинна враховувати ці властивості ліфта, тому що вони входять у його визначення.

Об’єктно-орієнтоване програмування

Що ж таке об’єктно-орієнтоване програмування (object-oriented programming, ООР)? Його можна визначити в такий спосіб:

об’єктно-орієнтоване програмування *– це методологія програмування, заснована на поданні програми у вигляді сукупності об'єктів, кожний з яких є екземпляром певного класу, а класи утворюють ієрархію успадкування.*

У даному визначенні можна виділити три частини:

1) ООР використає як базові елементи об'єкти, а не алгоритми;
2) кожен об'єкт є екземпляром якого-небудь певного класу;
3) класи організовані ієрархічно.

Програма буде об’єктно-орієнтованою тільки при дотриманні всіх трьох зазначених вимог. Зокрема, програмування не засноване на ієрархічних відношеннях, не належить до ООР, а називається програмуванням на основі *абстрактних типів даних*.

Відповідно до цього визначення не всі мови програмування є об’єктно-орієнтованими. Страуструп це визначив так: «якщо термін *об’єктно-орієнтована мова* взагалі що-небудь означає, то він повинен означати мову, що має засоби гарної підтримки об’єктно-орієнтованого стилю програмування... Забезпечення такого стилю у свою чергу означає, що в мові зручно користуватися цим стилем. Якщо написання програм у стилі ООР вимагає спеціальних зусиль або воно неможливо зовсім, то ця мова не відповідає вимогам ООР». Теоретично можлива імітація об’єктно-орієнтованого програмування на звичайних мовах, таких, як Pascal і навіть COBOL, або мова асемблеру, але це вкрай важко.

Підтримка успадкування в *об’єктно-орієнтованих мовах* означає можливість встановлення відношення "is-a" («є», «це є», «це»), наприклад, червона троянда – це квітка, а квітка – це рослина. Мови, що не мають таких механізмів, не можна віднести до об’єктно-орієнтованих. Такі мови називають *об'єктними*, а не об’єктно-орієнтованими. Відповідно до цього визначення об’єктно-орієнтованими мовами є Smalltalk, Object Pascal, C++ й CLOS, а Ada – об'єктна мова.

Об’єктно-орієнтоване проектування (OOD, object-oriented design). Програмування насамперед передбачає правильне й ефективне використання механізмів конкретних мов програмування. В проектуванні, навпаки, основна увага приділяється правильному й ефективному структуруванню складних систем. Таким чином, ***об’єктно-орієнтоване проектування*** – це методологія проектування, що поєднує в собі процес об'єктної декомпозиції та прийоми подання логічної й фізичної, а також статичної й динамічної моделей проектованої системи.

У даному визначенні є два ключових аспекти. Об’єктно-орієнтоване проектування:

1) ґрунтується на об’єктно-орієнтованій декомпозиції;
2) використовує різноманітні прийоми подання моделей, що відбивають логічну (класи й об'єкти) і фізичну (модулі й процеси) структури системи, а також її статичні й динамічні аспекти.

Саме об’єктно-орієнтована декомпозиція відрізняє об’єктно-орієнтоване проектування від структурного (алгоритмічного); у першому випадку логічна структура системи відображається абстракціями у вигляді класів й об'єктів, у другому – алгоритмами. В подальшому абревіатуру OOD, object-oriented design,

може бути використано для позначення методу об’єктно-орієнтованого проектування, визначеного в цьому посібнику.

Об’єктно-орієнтований аналіз. На об'єктну модель вплинула більш рання модель життєвого циклу програмного забезпечення. Традиційна техніка структурного аналізу заснована на потоках даних у системі. Об’єктно-орієнтований аналіз (або OOA, object-oriented analysis) спрямований на створення абстрактних моделей реального світу на основі об’єктно-орієнтованого світогляду.

Об’єктно-орієнтований **аналіз** – це методологія, при якій вимоги до системи сприймаються з погляду класів та об'єктів, виявлених у предметній області.

Як співвідносяться OOA, OOD й OOP? За результатами OOA формуються абстрактні моделі, на яких ґрунтується OOD; OOD, у свою чергу, створює фундамент для остаточної реалізації системи з використанням методології OOP.

Висновки

Об’єктно-орієнтоване програмування поліпшує проектування за рахунок фокусування на даних, як найбільш стабільному елементі обчислювальної системи. Об’єктно-орієнтований підхід концентрується на розробці коду, націленого на повторне використання. Об’єктно-орієнтована модель забезпечує кращу масштабованість проектів. Саме тому більшість успішних сучасних технологій проектування програмних систем передбачає переважне або виняткове використання об’єктно-орієнтованої парадигми.

1.1.4. Об'єктна модель предметного середовища, принципи її побудови

Для об'єктно-орієнтованого стилю концептуальна база – це об'єктна модель. Об'єктну модель складають чотири головні елементи:

- абстрагування (abstraction);
- інкапсуляція (encapsulation);
- модульність (modularity);
- ієрархія (hierarchy).

Абстрагування дозволяє виділити такі істотні характеристики деякого об'єкта, які відрізняють його від всіх інших типів об'єктів.

Інкапсуляція – це відокремлення один від одного елементів об'єкта, що визначають його устрій і поведінку; інкапсуляція також служить для того, щоб ізолювати зовнішнє поводження об'єкта від його внутрішнього устрою.

Ієрархія – це впорядкування абстракцій, засіб класифікації об'єктів, систематизація зв'язків між об'єктами.

Модульність – це представлення системи у вигляді сукупності відокремлених сегментів, зв'язок між якими забезпечується за допомогою зв'язків між класами, визначеними в цих сегментах.

Тепер розглянемо призначення і взаємодію цих елементів більш докладно.

Зміст абстрагування, як елементу об’єктної моделі

Абстрагування є одним з основних прийомів, використовуваних при вирішенні складних задач. ***Абстракція*** виділяє такі істотні характеристики кожного об'єкта, які відрізняють його від всіх інших видів об'єктів і, таким чином, чітко визначає його концептуальні границі з погляду спостерігача. Наприклад, будь-який автомобіль має корпус і колеса, причому корпуси автомобілів можуть дуже істотно відрізнятися один від одного. Проте навряд чи можна сплутати автомобіль з возом, незважаючи на те що віз також має корпус і колеса. Тобто абстракції автомобіля і возу відрізняються один від одного, незважаючи на те, що вони мають деякі спільні властивості.

Під абстрактними моделями, що використовуються в об’єктно-орієнтованому проектуванні, розуміється не щось уявне чи таке що не має відношення до дійсності, а моделі, для яких характерні найбільш загальні властивості, актуальні для практичних випадків.

Об’єктно-орієнтоване програмування характеризується наявністю двох основних видів абстракцій:

– тип даних об'єктної природи (***клас***) – обумовлене програмістом розширення вихідних типів мови.

– екземпляр класу (***об'єкт***) – змінна класу. Об'єктів у складній системі звичайно набагато більше, ніж класів. Об'єкт характеризується *станом*, *поводженням* і *ідентичністю*.

Стан об'єкта характеризується набором його *властивостей* (*атрибутів*) і поточними значеннями кожної з цих властивостей (ПЕОМ працює, диск відформатовано).

Стан об'єкта – результат його ***поводження***, тобто виконання у визначеній послідовності характерних для нього дій (при включенні дисплею його екран починає світитися).

Ідентичність – це така властивість (чи набір властивостей) об'єкта, що дозволяє відрізнити його від всіх інших об'єктів того ж типу. Ідентичність не обов'язково пов'язана з назвою або зі станом об'єкта: дві зовсім однакові ПЕОМ, що розміщуються в одному приміщенні, все ж таки є різними об'єктами. Властивістю, що відповідає за ідентичність у цьому випадку, може вважатися серійний (заводський) номер системного блоку ПЕОМ.

У якості простішого та інтуїтивно зрозумілого всім прикладу абстракції можна навести електричну лампочку, для якої визначені, як мінімум, два властивих їй стани: включена (світить) і виключена (не світить) (рисунок 1.5).

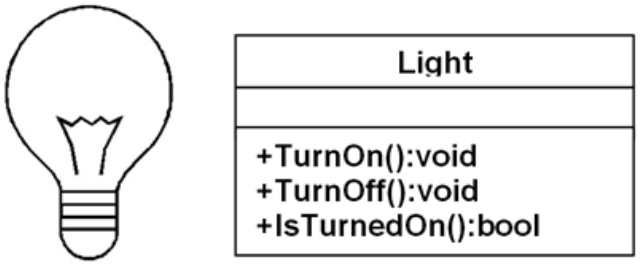


Рисунок 1.5 – Абстракція електричної лампи

Процес вмикання й вимикання лампочки описує поводження об'єкта. Вмиканню лампи на рівні абстрактної моделі відповідає функція **TurnOn()**, вимкненню – функція **TurnOff()**. Для того щоб довідатися, світиться лампа чи ні, людині достатньо подивитися на неї. На рівні абстрактної моделі цей процес представлено спеціальною функцією **isTurnedOn()**, що дозволяє встановити поточний стан модельного об'єкту. На діаграмі класу знак «плюс» поруч із ім'ям кожної функції показує, що ці функції становлять відкритий інтерфейс класу.

Всі лампочки мають приблизно однакове поводження, хоча їхні реалізації можуть відрізнятися. Принцип роботи вольфрамових, галогенних, газових ламп різний, однак кожна з них розпізнається нами як освітлювальний пристрій, який можна включити й виключити. Які фізичні й хімічні процеси беруть участь при переході лампи з вимкнутого стану у включене конкретного споживача може не цікавити. Йому може бути достатньо знати, де і як правильно використати цей пристрій. Конструюючи абстрактну модель, програміст намагається відобразити в ній, у першу чергу, найбільш загальні властивості реальних об'єктів.

Оголошення мовою C++, що відповідає даній моделі, може виглядати наступним чином:

```
class Light {  // Клас Light
public:
        void TurnOn();
        void TurnOff();
        bool IsTurnedOn();  };
```

На програмному рівні організацію роботи можна представити у такий спосіб: для того, щоб скористатися функціональністю відкритого інтерфейсу класу, необхідно створити об'єкт даного класу (який і є моделлю лампи). Для управління об'єктом у програмі визначається відповідна змінна:

```
// Фрагмент коду на C++
Light   light1;                 // Об'єкт класу Light
light1.TurnOn();                // "Включити"
light1.TurnOff();               // "Виключити"
```

Зміст інкапсуляції, як елементу об’єктної моделі

Інкапсуляція щільно пов'язана з абстракцією. Якщо абстрагування спрямоване на зовнішнє поводження об'єкта, що спостерігається, то інкапсуляція займається внутрішнім устроєм і тому визначає чіткі границі між різними абстракціями. Механічна коробка передач для автомобіля Audi не підходить до «Жигулів» через розходження в реалізації, незважаючи на загальний зовнішній інтерфейс.

Інша важлива властивість інкапсуляції – можливість приховувати реалізацію. Інкапсуляція завжди передбачає можливість обмеження доступу до даних класу. З одного боку, це дозволяє спростити інтерфейс класу, показавши найбільш істотні для зовнішнього користувача дані і методи. З іншого боку, приховування реалізації забезпечує можливість внесення змін у реалізацію класу без зміни інших класів. Даний аспект дуже важливий з погляду подальшого супроводження і модернізації програмного коду.

Для найпростішої абстракції електричної лампи рівень реалізації пов'язаний з тим, яким чином забезпечується представлення стану лампи. Наприклад, це можна зробити з використанням логічної (**bool**) змінної (див. діаграму на рис. 1.6). Знак «мінус» в атрибуті **turnedOn** на діаграмі показує, що доступ до нього ззовні закрито. Це означає, що змінити значення змінної

turnedOn можна тільки використовуючи функції, що утворюють зовнішній інтерфейс (у нашому випадку – функції **TurnOn()** і **TurnOff()**). Таким чином, якщо спробувати знайти на діаграмі найбільш точний графічний еквівалент інкапсуляції, то, імовірно, варто вважати їм лінію, що відокремлює закритий для доступу ззовні атрибут **turnedOn** від відкритих методів класу.

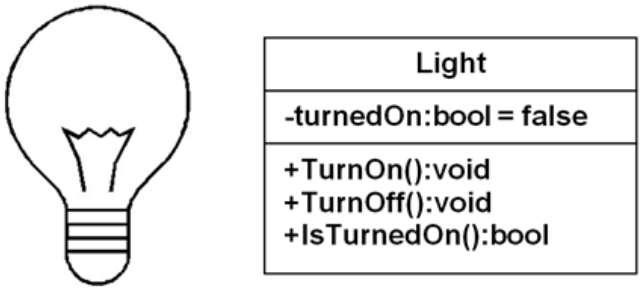


Рисунок 1.6 – Інкапсуляція: відокремлення інтерфейсу від реалізації

Уточнене визначення класу **Light** на C++ може мати наступний вигляд:

```
class Light {
private:                // специфікатор, що вказує на недоступність ззовні
    bool turnedOn;  // недоступне ззовні
public:                 // специфікатор, що вказує на відкритість елементів класу
    void TurnOn();
    void TurnOff();
    bool isTurnedOn() ;   };
```

Реалізація функцій-членів класу Light може мати наступний вигляд:

```
void Light::TurnOn() { turnedOn = true; }
void Light::TurnOff() { turnedOn = false; }
bool Light::IsTurnedOn() { return turnedOn; }
```

Таким чином, закритий атрибут класу **turnedOn** доступний тільки для функцій-членів класу. Немає легальної можливості змінити змінну **turnedOn** ззовні. У даному прикладі функції-члени класу працюють із однією змінною-членом класу, що і відповідає за стан об'єкта, а об'єкт класу **Light** зберігає єдине значення логічного типу.

Відношення класів та ієрархія класів

Якщо розглядати об’єктно-орієнтоване програмування (ООП), як програмування, що сфокусовано на дані, а обчислювальний процес, як моделювання поводження, то об'єктом вивчення повинні, насамперед, стати форми організації структур даних, властивих об’єктно-орієнтованому програмуванню, і форми взаємодії об'єктів усередині програмної системи, побудованої з використанням ООП.

Основні відношення класів

Орієнтуючись на специфікацію UML, можна визначити чотири базових види відношень між об’єктно-орієнтованими абстракціями (більш детально мова UML розглядається в наступних розділах даного посібника).

Відношення ***узагальнення*** (generalization) описує відношення між загальною сутністю і її конкретним втіленням (наприклад, один клас є спеціалізацією іншого класу).

Відношення ***асоціації*** (association) описує структурне відношення, що показує, що об'єкти одного типу деяким чином пов'язані з об'єктами іншого типу (наприклад, перебувають відносно типу «частина / ціле»).

Відношення ***залежності*** (dependency) описує відношення використання, при якому зміна в специфікації одного класу може вплинути на клас, що його використовує (наприклад, об'єкти деякого класу передаються в якості аргументів функціям-членам іншого класу).

Відношення ***реалізації*** (realization) описує семантичне відношення, при якому клас гарантує виконання контракту, обумовленого деяким інтерфейсом. Як правило, визначається стосовно до *інтерфейсів* і класів, які *реалізують* (implement) інтерфейси.

Далі наведено пояснення змісту цих визначень із використанням прикладу конструювання об'єктної моделі для гірлянди кольорових лампочок, подібної тим, що вішають на ялинку перед новорічним святом. Цей досить простий і разом з тим наочний приклад дозволяє проілюструвати всі основні види відношень між класами.

Відношення узагальнення

Гірлянда повинна складатися не зі звичайних лампочок, а з кольорових. Зрозуміло, що кольорова лампочка може вважатися різновидом звичайної, однак при цьому їй притаманна, принаймні, одна додаткова властивість. Абстрактну модель такої лампочки представлено у відношенні «***узагальнення***» до звичайної лампочки (рис. 1.7).

Клас **ColoredLight** розширює визначення **Light**, *успадковуючи* його елементи й додаючи до цих елементів нові. При цьому говорять, що клас **ColoredLight** *успадковує* клас **Light**. Відношення успадкування є, ймовірно, найпоширенішим типом відношення узагальнення. У відповідності зі специфікацією UML, відношення узагальнення зображується на діаграмі класів у вигляді суцільної лінії з незафарбованою стрілкою, спрямованої від дочірнього (більше спеціалізованого) елемента до батьківського (більше загального).

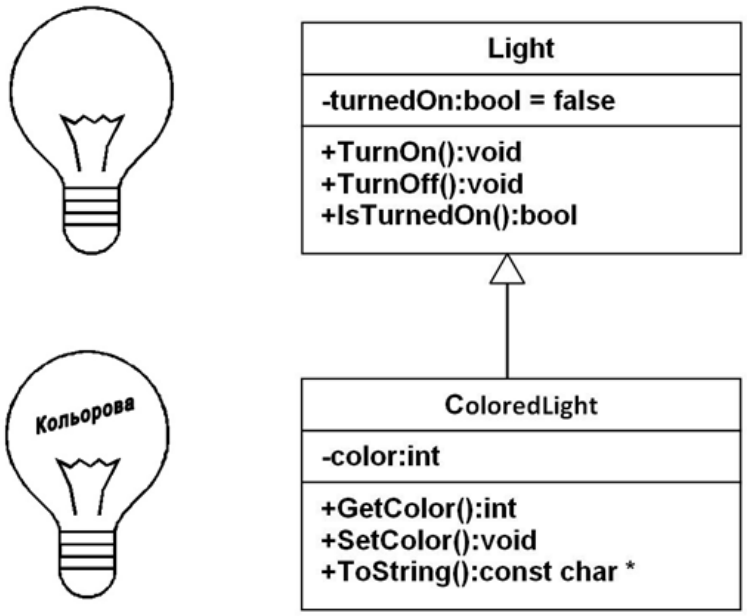


Рисунок 1.7 – Клас **ColoredLight** розширює визначення класу **Light**

Клас **ColoredLight** додає до методів, успадкованих із класу **Light**, методи, що дозволяють встановити або довідатися про колір лампочки, а також – функцію, що представляє колір лампочки у вигляді строкового літерала.

Для спрощення реалізації в даному прикладі колір подається у вигляді цілого числа.

Фрагмент вихідного коду, побудованого на основі даного відношення, може виглядати в так:

```
class Light             // Базовий клас
{
  private:
        bool  turnedOn;
  public:
        void TurnOn() { turnedOn = true; }
        void TurnOff() { turnedOn = false; }
        bool IsTurnedOn() { return turnedOn; }
};

class ColoredLight : public Light  // Похідний клас: успадковує
                                   // властивості базового класу
{
  private:
        int color;
  public:
        void SetColor(int);
        int GetColor();
        const char *ToString();
};
```

Використовуючи об'єкт похідного класу, можна звертатися до відкритих елементів базового класу:

```
ColoredLight cl;              // Об'єкт – змінна, экземпляр класу
cl.SetColor(0);               // Звертання до методу класу ColoredLight
cl.TurnOn();                  // Звертання до методу класу Light
ColoredLight * pCl = new ColoredLight; // Покажчик на об'єкт класу ColoredLight
pCl->TurnOn();                // Звертання до методу класу Light
```

Відношення асоціації

Звичайна гірлянда – це набір кольорових лампочок. Для об’єктної моделі гірлянди це означає, що об'єкт класу, втіленням якої виступає абстракція гірлянди (**Garland**), повинен надавати можливість працювати з множиною об'єктів класу **ColoredLight**. Діаграма на рис. 1.8 показує можливу організацію класу **Garland**.

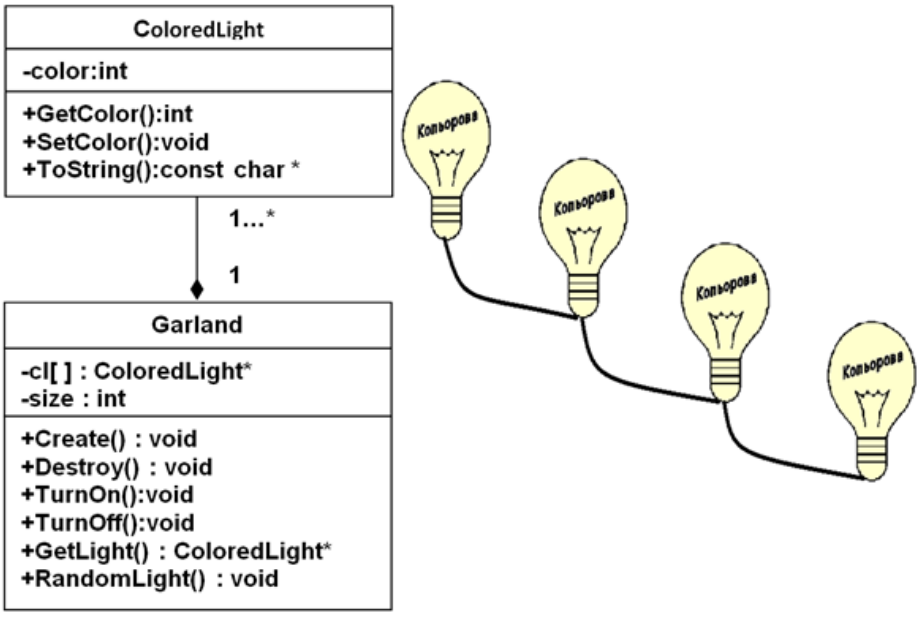


Рисунок 1.8 – Гірлянда складається з кольорових лампочок

Як можна бачити з діаграми, у якості одного з елементів класу **Garland** виступає масив покажчиків на об'єкти типу **ColoredLight**. При цьому говорять, що клас **Garland** перебуває у відношенні ***асоціації*** до класу **ColoredLight**. Зовнішній інтерфейс класу визначають три методи:

– метод **TurnOn()** – моделює включення гірлянди;

– метод **TurnOff()** – моделює вимикання гірлянди;

– метод **GetLight()** – для доступу до об'єкту класу **ColoredLight**, який моделює окрему кольорову лампочку в складі гірлянди. Наявність такого доступу дає можливість управляти окремими лампочками в складі гірлянди, використовуючи відкритий інтерфейс класу **ColoredLight**.

Метод **RandomLight()** призначений для випадкової генерації об'єктів **ColoredLight** і повинен викликатися в процесі ініціалізації гірлянди. Яким чином здійснюється ініціалізація об'єктів, буде розглянуто в наступних розділах.

Фрагмент вихідного коду, що містить визначення класу **Garland**, може виглядати так:

```
class Garland {                    // Клас, що втілює абстракцію гірлянди
private:
        typedef ColoredLight *PColoredLight;    // Тип-покажчик на ColoredLight
        PColoredLight  *cl;        // Покажчик на початок масиву покажчиків
                                   // на об'єкти типу ColoredLight
        int size;                  // Розмір гірлянди
public:
```

```
        void Create( int customSize );      // Створити гірлянду заданого розміру
        void Destroy ();                    // Знищити раніше створену гірлянду
        void TurnOn();                      // "Включити" гірлянду
        void TurnOff();                     // "Виключити" гірлянду
        PColoredLight   GetLight( int-);    // Доступ до окремої лампочки
        void RandomLight();
};
```

Поширені типи асоціацій. Найпоширенішими асоціаціями є відношення *композиції* й *агрегування*, при цьому композиція, власне кажучи, є варіантом агрегування із чітко вираженим відношенням володіння. Відмінність композиції від простого агрегування може бути проілюстрована наступним прикладом. Студенти можуть навчатися в одному або декількох навчальних закладах (таким чином, студент і навчальний заклад перебувають у відношенні простого агрегування, що не є композицією), а факультет може належати тільки одному закладу (таким чином, факультет і навчальний заклад перебувають у відношенні композиції). Нарешті, студент відвідує курси, але, зрозуміло, не є їхньою складовою частиною (студент і курс перебувають у відношенні асоціації, що є агрегуванням).

У відповідності зі специфікацією UML, графічно асоціація зображується у вигляді лінії, поруч із якою можуть бути присутніми додаткові позначення (наприклад, кратність, імена ролей, ім'я самої асоціації). Лінія може завершуватися стрілкою у формі ромба. Ромб, що завершує стрілку на діаграмі UML, зафарбовується, якщо мова йде про композиції, і не зафарбовується у випадку простого агрегування.

Відношення залежності

Відношення ***залежності*** є таким типом відношень між класами, коли зміна в специфікації або реалізації одного класу впливає на специфікацію або реалізацію іншого класу. Приклад відношення залежності між класами представлений на рисунку 1.9.

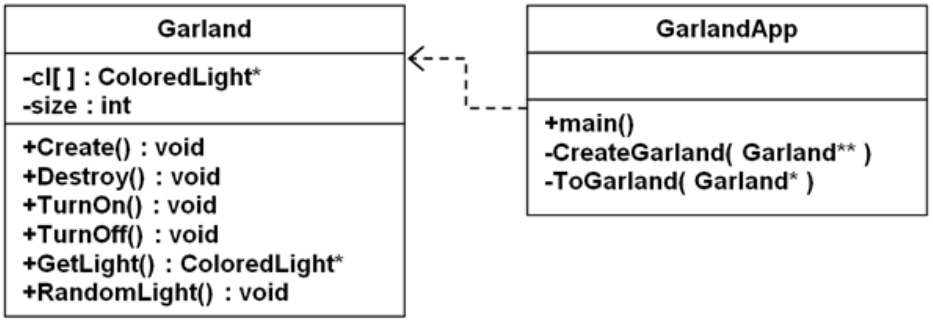


Рисунок 1.9 – Залежність **GarlandApp** від реалізаціїї **Garland**

В цьому прикладі два методи класу **GarlandApp** приймають аргументи типу **Garland** (в першому методі - це адреса покажчика на об'єкт класу **Garland**, в другому - покажчик на об'єкт класу **Garland**). Таким чином, на реалізацію методів **CreateGarland()** і **TestGarland()** можуть впливати зміни у визначенні класу **Garland**.

Графічно відношення залежності зображується пунктирною лінією, спрямованої від залежного елемента діаграми до того елемента, від якого він залежить.

Відношення реалізації

Найчастіше відношення реалізації використовується для визначення відношення між *інтерфейсом* і *класом*, що реалізує інтерфейс. Інтерфейс визначає набір елементів (як правило, дій), характерних для об'єктів, яким притаманні певні властивості.

Оскільки діям відповідають *методи*, то інтерфейс визначає сигнатури методів (тобто визначає, як повинні називатися ***методи***, скільки вони повинні приймати аргументів, який тип повинні мати аргументи й тип значення, що повертається методом). Тим самим, інтерфейс визначає так званий ***контракт***. Клас, що підтримує даний інтерфейс, повинен забезпечити реалізацію контракту, заданого інтерфейсом.

На рисунку 1.10 представлений варіант визначення класу **ColorableObject** на основі інтерфейсу **IColorable**. Проілюструємо на цьому прикладі основну ідею, що лежить в основі концепції інтерфейсу. Кольором можуть характеризуватися не тільки лампочки, але й інші найрізноманітніші об'єкти. У рамках програмної моделі було б непогано мати який-небудь цілком уніфікований механізм роботи з такими об'єктами. Інтерфейс **IColorable** надає

найпростішу модель такого механізму. Інтерфейс містить оголошення двох методів **SetColor()** й **GetColor()**, які повинен підтримувати будь-який об'єкт, який має колір. Інтерфейс не надає реалізацію методів. Тобто інтерфейс **IColorable** – це, власне кажучи, чиста абстракція будь-якого кольорового об'єкта.

Визначивши для будь-якого класу відношення реалізації інтерфейсу **IColorable**, розробник повинен надати реалізацію методів **SetColor()** і **GetColor()**, дотримуючись тим самим контракту, встановленого інтерфейсом (у розглянутому прикладі класом, що реалізує інтерфейс **IColorable**, є клас **ColorableObject**).

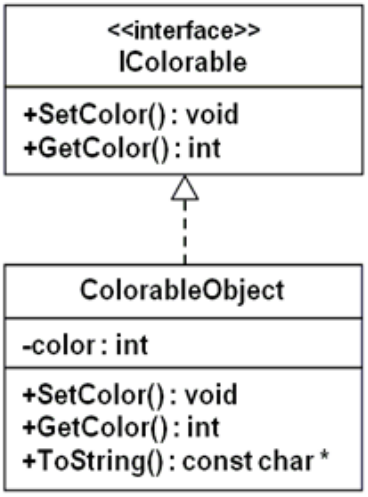


Рисунок 1.10 – Клас **ColorableObject** надає реалізаціїю інтерфейсу **IColorable**

У такій інтерпретації **ColoredLight** може трактуватися, як різновид **ColorableObject**. Можна реалізувати й інші різновиди **ColorableObject**.

Графічно відношення реалізації зображується у вигляді пунктирної лінії з незафарбованою стрілкою, спрямованої від сутності, що забезпечує реалізацію, до сутності, що визначає контракт.

Наявність інтерфейсів надає можливість працювати з об'єктами різних типів у тій частині, що підтримує відповідний інтерфейс. Це означає, що програміст має можливість написати загальний код для об'єктів самих різних типів, якщо цей код стосується тільки тієї функціональності, що пов'язана з реалізованим інтерфейсом, наприклад:

```
void SetWhite ( IColorable *ic_supported )
        { ic_supported->SetColor( 0 /* 0 - білий колір */ ); }
```

Функції **SetWhite** можна передати покажчик на будь-який об'єкт, що підтримує інтерфейс **IColorable**.

Треба відзначити, що більшість сучасних мов програмування підтримує інтерфейси у вигляді вбудованих засобів мови (включаючи Java, C#, Visual C++.NET with Managed Extentions). C++ безпосередньо не підтримує інтерфейси, однак еквівалентна їм семантика може бути виражена за допомогою абстрактних класів.

На основі розглянутих фрагментів, що ілюструють окремі види відношень між класами, можна побудувати повну ієрархію класів проекту «Гірлянда з кольорових лампочок» (**Garland Application**) – результуюча діаграма наведена на рисунку 1.11.

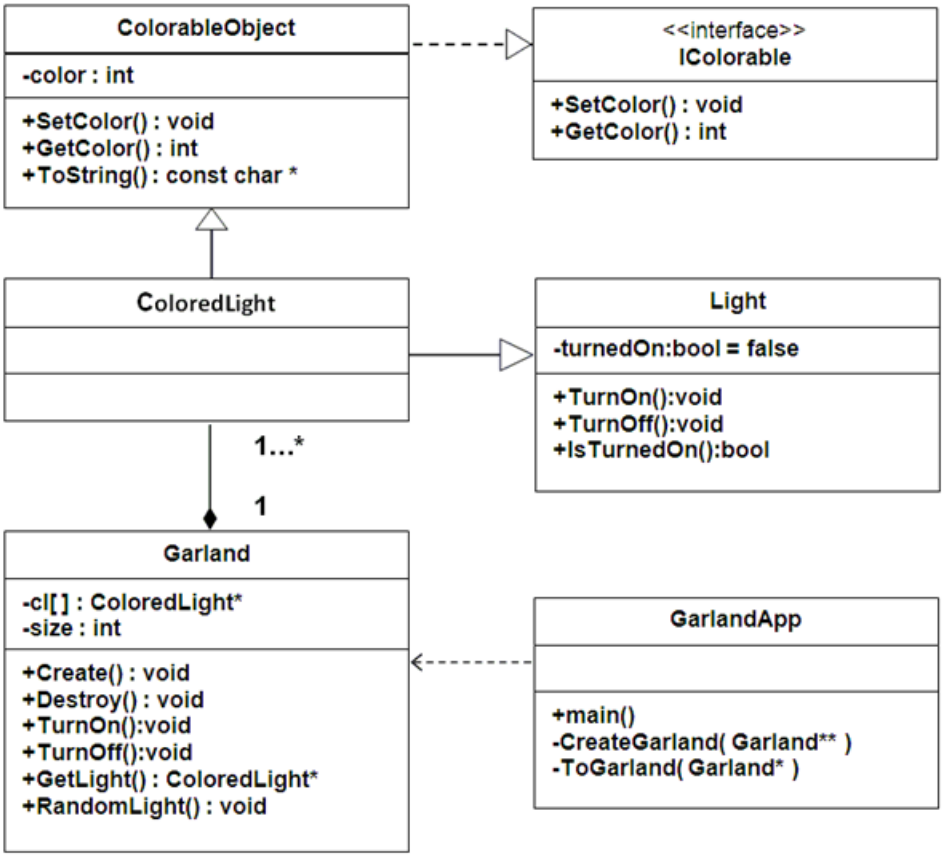


Рисунок 1.11 – Діаграма класів **Garland Application**

З даної діаграми видно, що в рамках проекту **Garland Application** використовуються всі розглянуті відношення між класами. Специфікація класу

ColoredLight спростилася настільки, що він взагалі не містить власних елементів, а поєднує функціональності класів **Light** й **ColorableObject**, використовуючи множинне успадкування.

Фізичний та логічний рівні представлення модульності об’єктної моделі

Нагадаємо, що ***модульність*** була визначена як подання системи у вигляді сукупності відокремлених сегментів, зв'язок між якими забезпечується за допомогою зв'язків між класами, визначеними в цих сегментах. При цьому виділяють два рівні подання модульності, які умовно називаються ***фізичним*** і ***логічним*** рівнями.

Фізичний рівень має справу зі структурою файлів і каталогів, у яких зберігаються окремі модулі проекту. Так, фізичному модулю відповідає файл із вихідним текстом C/C++, до якого підключені необхідні файли-заголовки, що утворять інтерфейс між модулями. Також в C/C++ фізичним модулем є окремий сегмент компіляції, тобто *об'єктний* файл. Мова C та перші реалізації мови C++ не надавали інших механізмів реалізації модульності, тому фізичний модуль був одночасно й логічним модулем. Модульна організація проекту в цьому випадку тісно пов'язана з моделлю обробки програми (рис. 1.12).

В ході трансляції вихідних текстів препроцесор обробляє директиви **#include**, забезпечуючи переміщення в область бачності вихідного тексту необхідних оголошень, наприклад, оголошень функцій, реалізованих в інших об'єктних модулях. Результати компіляції вихідного тексту записуються в об'єктні модулі, в яких деякі посилання залишаються ще в символічній формі, тобто так називані нерозрішені (**unresolved**) посилання. Важливою перевагою проектування, яку надає модульність, є можливість роздільної трансляції модулів. Модулі взаємодіють один з одним за допомогою викликів функцій і читання-запису глобальних даних. Модуль може надавати доступ не тільки до функцій, але також і до визначень типів і констант.

Компонувальник (редактор зв’язків, **linker**) здійснює зв'язування об'єктних модулів, отриманих у результаті трансляції файлів, і об'єктних модулів, що зберігаються в стандартній бібліотеці й в інших бібліотеках.

Проте робота над проектом, організованим на основі концепції фізичної модульності, ускладнюється із зростанням складності задач і кількості

програмістів, що беруть участь у розробці. Об’єктно-орієнтована парадигма з концепцією класу, як фрагменту коду, орієнтованого на його багаторазове використання, також висуває додаткові вимоги до забезпечення модульності засобами мови. Виникає необхідність організації проекту на основі логічного взаємозв'язку його компонентів, а у зв'язку із цим – забезпечення коректного розрішення можливих конфліктів глобальних імен (змінних, констант, типів, функцій), розділення реалізації й інтерфейсної частини визначень, що містяться в модулі, відокремлення бібліотечного й клієнтського коду. Це призводить до необхідності реалізації механізмів модульності, що відображають логічні взаємозв'язки між окремими компонентами.

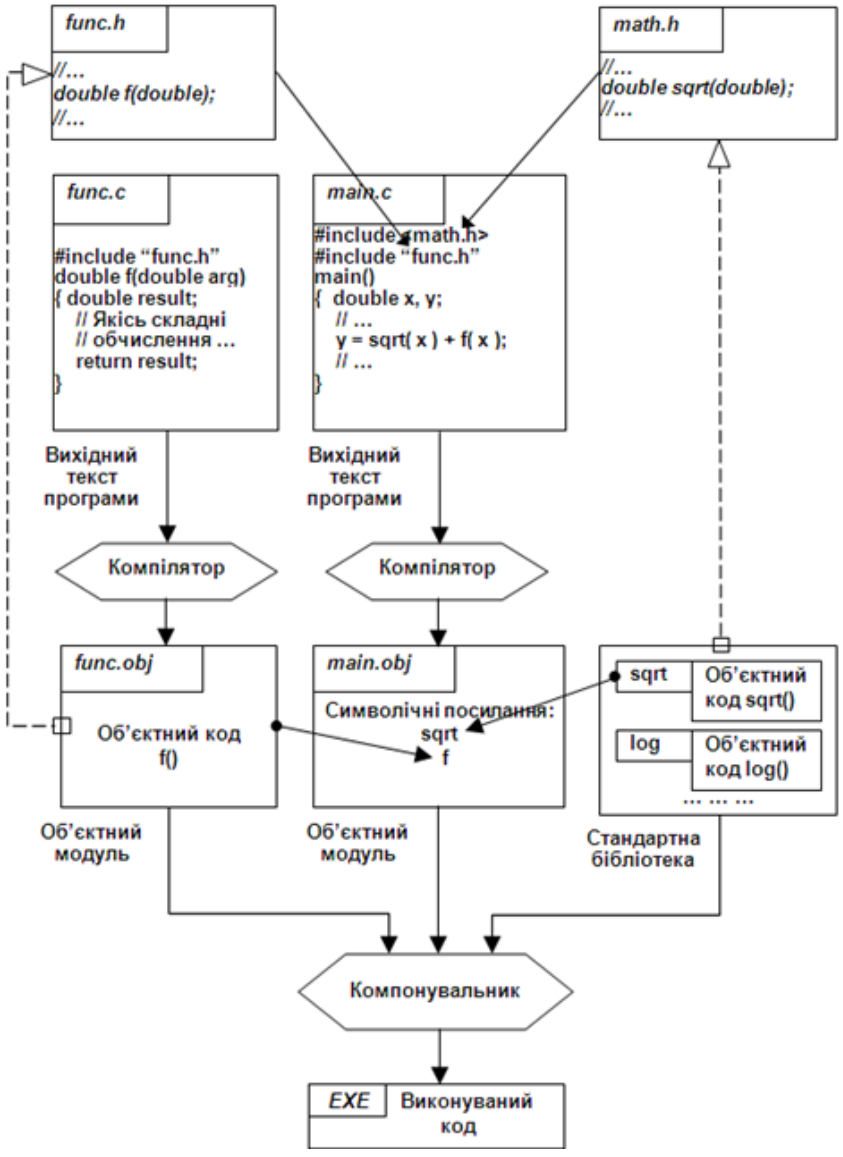


Рисунок 1.12 – Модульність та обробка програми мовою C/C++

Основний механізм логічного групування програмних об'єктів, підтримуваний сучасними мовами програмування – це механізм *просторів імен*. Реалізація просторів імен в C++ розглядається в одному з наступних розділів.

Нижче розглядається можливий варіант модульної організації проекту **Garland Application** (рис. 1.13).

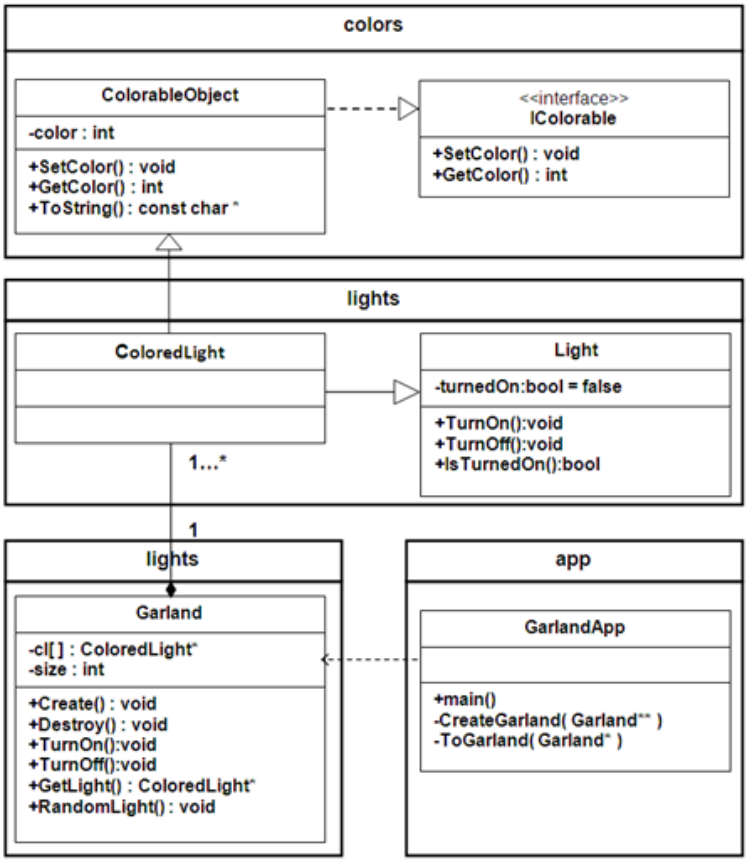


Рисунок 1.13 – Реалізація логічної модульності у проекті **Garland Application**

Дана схема ілюструє модульність на *логічному рівні*. Визначення окремих класів можуть фізично розміщуватися в різних файлах. Досить розповсюдженою в сучасній практиці є модель організації проекту за схемою *один фізичний модуль – один клас*, у якій використовується наступна стандартна інфраструктура файлів з вихідними текстами:

– файл-заголовок (з розширенням **h** або **hpp**) містить визначення класу (і, можливо, використовуваних у цьому визначенні інших оголошень).

– файл із розширенням **cpp** містить реалізацію методів класу.

На рис. 1.13 виділено 4 логічні модулі:

– **colors –** містить визначення типів, інтерфейсів і класів для подання абстракцій, пов'язаних з колірними характеристиками об'єктів;

– **lights –** містить визначення класів для подання моделей лампочок;

– **garlands –** містить визначення класу для подання гірлянди (якщо буде потрібно визначити інший клас для подання якої-небудь іншої моделі гірлянди, його також можна помістити в цей модуль);

– **app –** містить визначення головної функції програми, яка також подається як окремий клас додатку.

Логічні модулі можуть однозначно відповідати фізичним модулям (як, наприклад, у мові Java) або надавати незалежний від розміщення фізичних модулів механізм логічного групування програмних об'єктів (як, наприклад, у мові C#).

1.2. Основи об'єктно-орієнтованого проектування мовою UML

1.2.1. Характеристика мови UML. Канонічні діаграми

Мова UML (Unified Modeling Language – уніфікована мова моделювання) – це загальноцільова мова візуального об’єктного моделювання, яку розроблено для специфікації, візуалізації, проектування і документування компонентів програмного забезпечення, бізнес-процесів та інших систем. Мова UML є достатньо строгим і потужним засобом моделювання, який може бути ефективно застосовано для побудови моделей складних систем різного цільового призначення. Дана мова включає найкращі якості і досвід методів програмної інженерії, які з успіхом застосовувались протягом останніх років при моделюванні великих і складних систем.

UML – це відкритий стандарт, який використовує графічні позначення для створення *абстрактної моделі* системи. Така модель має назву ***UML-модель***. Стандарт UML було створено для визначення, візуалізації, проектування і документування, перш за все, програмних систем. UML не є мовою програмування, але на основі UML-моделей можлива генерація програмного коду.

З погляду методології UML, об’єктна модель складної системи являє собою певну кількість взаємопов’язаних представлень (views), кожне з яких відображає певний аспект поведінки або структури системи. При цьому найбільш загальними представленнями складної системи прийнято вважати *статичне* і *динамічне*, які в свою чергу можуть поділятися на інші більш деталізовані.

При цьому вихідна або первинна модель складної системи є найбільш загальним представленням *концептуального* рівня. Така модель, яка має назву ***концептуальної***, будується на початковому етапі проектування і зазвичай не містить багатьох деталей і аспектів складної системи. Наступні моделі конкретизують концептуальну модель, доповнюючи її представленнями *логічного* і *фізичного* рівнів.

В цілому процес об’єктної декомпозиції можна розглядати як послідовний перехід від розробки найбільш загальних моделей і представлень концептуального рівня до більш деталізованих представлень логічного і фізичного рівнів. При цьому на кожному етапі моделі послідовно доповнюються все більшою кількістю деталей, що дозволяє їм більш адекватно відображати різні аспекти конкретної реалізації складної системи.

Загальна схема взаємозв’язків моделей і представлень складної системи в процесі об’єктно-орієнтованого аналізу та проектування представлена на рисунку 1.14.

Рисунок 1.14 – Загальна схема взаємозв’язків і представлень складної системи

Пакети в мові UML

Пакет (package) – загальноцільовий механізм для організації елементів моделі у множину у відповідності до принципу декомпозиції моделі складної системи, який також припускає вкладеність пакетів один до одного.

Про включені до пакету елементи говорять, що вони належать пакету або входять до нього. Кожен елемент може належати тільки до одного пакету. В свою чергу, деякі пакети можуть бути вкладені в інші пакети.

Підпакет (subpackage) – пакет, який є складовою частиною іншого пакету. Всі елементи підпакета належать до більш загального пакету. Це утворює відношення вкладеності пакетів, яке являє собою ієрархію.

Для графічного зображення пакетів на діаграмах застосовується спеціальний графічний символ – великий прямокутник із невеликим прямокутником, який приєднано до лівої частини верхньої сторони першого (рис. 1.15). Всередині великого прямокутника може бути записана інформація,

яка відноситься до даного пакету. Якщо такої інформації немає, то всередині великого прямокутника записується ім’ям пакета, яке повинно бути унікальним в межах всієї моделі системи (рис. 1.15, а). У випадку, коли така інформація є, ім’я пакету записується у верхньому малому прямокутнику (рис. 1.15, б).

Рисунок 1.15 – Графічне зображення пакетів в UML

Одним з типів відношень між пакетами є відношення *вкладеності* або *включення* пакетів один до одного. В UML таке відношення може бути зображене простим розміщенням одного пакета всередині іншого або у вигляді дерева (рис. 1.16).

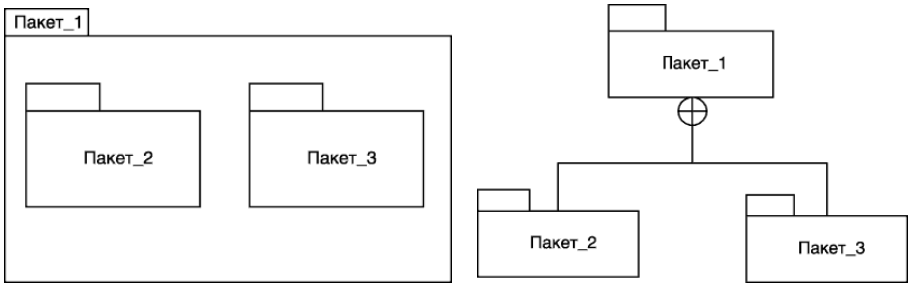


Рисунок 1.16 – Графічне зображення вкладеності пакетів

Підкласом пакета є модель, яка являє собою абстракцію реальної системи. В UML для однієї системи можуть бути визначені різні моделі, кожна з яких специфікує систему з різних точок зору. Пакет може включати до себе декілька різних моделей однієї системи. Загальна модель системи містить у собі *модель аналізу* та *модель* проектування (рис. 1.17).

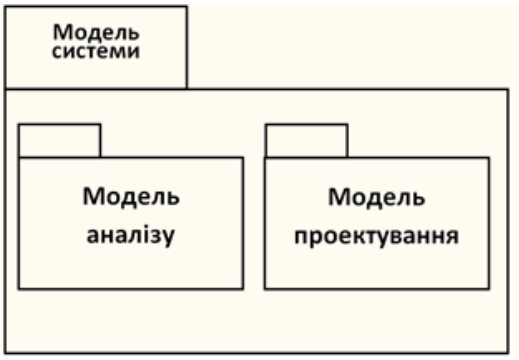


Рисунок 1.17 –Зображення моделі системи у вигляді пакетів моделей

Канонічні діаграми мови UML

Модель складної системи мовою UML представляється у вигляді спеціальних графічних конструкцій, які мають назву «**діаграми»**.

Діаграма **(diagram*)*** – це графічне представлення сукупності елементів моделі в формі зв'язного графу, вершинам і ребрам (дугам) якого надається певна семантика. Нотація *канонічних діаграм* є основним засобом розробки моделей мовою UML.

В нотації мови UML визначені наступні види *канонічних діаграм*:

- варіантів використання (use case diagram);
- класів (class diagram);
- кооперації (collaboration diagram);
- послідовності (sequence diagram);
- станів (statechart diagram);
- діяльності (activity diagram);
- компонентів (component diagram);
- розгортання (deployment diagram).

Перелік цих діаграм і їх назви є ***канонічними*** в тому сенсі, що вони представляють собою невід'ємну частину графічної нотації мови UML. Більш того, процес об'єктно-орієнтованого аналізу і проектування невід'ємно пов'язаний з процесом побудови цих діаграм. Сукупність канонічних діаграм вважається самодостатньою с тому сенсі, що вони містять всю необхідну для реалізації проекту складної системи інформацію.

Кількість видів діаграм для конкретної моделі системи строго не фіксовано. Для простих додатків немає необхідності будувати всі без виключення канонічні діаграми. Відсутність деяких діаграм у проекті системи не вважається помилкою

розробника. Наприклад, в об’єктній моделі системи може бути відсутня діаграма розгортання для програмного додатку, який виконується локально на комп’ютері користувача. Перелік діаграм залежить від специфіки конкретного проекту системи.

Кожна з цих діаграм деталізує і конкретизує окремі уявлення о моделі складної системи в термінах мови UML. При цьому діаграма *варіантів використання* являє собою найбільш загальну концептуальну модель складної системи, яка є вихідною для побудови всіх інших діаграм. Діаграма *класів* є логічною моделлю, яка відображає статичні аспекти структурної побудови складної системи.

Діаграми *кооперації* і *послідовності* представляють різновиди логічної моделі, які відображають динамічні аспекти функціонування складної системи. Діаграми *станів* і *діяльності* призначені для моделювання поведінки системи. А діаграми *компонентів* і *розгортання* призначені для відображення фізичних компонентів складної системи і тому відносяться до її фізичної моделі.

Інтегрована модель складної системи в нотації UML може бути представлена у вигляді сукупності зазначених вище діаграм (рис. 1.18).

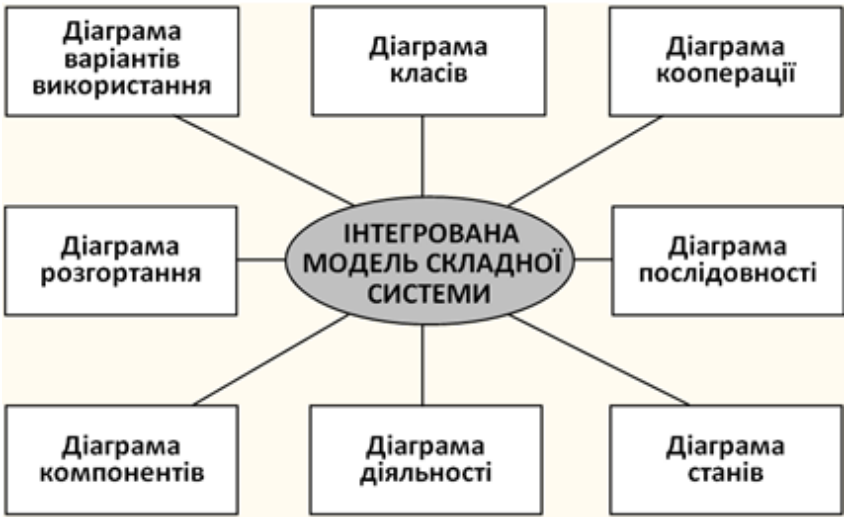


Рисунок 1.18 – Інтегрована модель складної системи

1.2.2. Система позначень уніфікованої мови моделювання

Графічні позначення на діаграмах UML

Діаграми є, переважено, графами спеціального виду. Вони складаються з вершин в формі геометричних фігур, які зв'язані між собою ребрами (дугами). Геометричні розміри і взаємне розташування елементів не мають принципового значення.

Для діаграм UML існують три типи візуальних графічних позначень, які є важливими з точки зору розміщення в них інформації: *геометричні фігури*, *графічні взаємозв'язки* та *спеціальні графічні символи*.

Геометричні фігури використовуються у якості вершин графів відповідних діаграм. Геометричні фігури виступають в ролі графічних примітивів мови UML, а форма цих фігур (прямокутник, еліпс) повинна строго відповідати зображенню окремих елементів мови UML (клас, варіант використання, стан, діяльність). Графічні примітиви мови UML мають фіксовану семантику, перевизначати яку користувачам не дозволяється. Графічні примітиви повинні мати власні імена і, можливо, інший текст, який міститься всередині відповідних геометричних фігур або, як виняток, поблизу цих фігур. У

Графічні взаємозв'язки представляються різними лініями на площині. Взаємозв'язки в мові UML узагальнюють поняття дуг і ребер з теорії графів, але мають менш формальний характер та більш розвинену семантику.

Спеціальні графічні символи зображуються поблизу від тих або інших візуальних елементів діаграм і мають характер додаткової специфікації.

Крім визначених для кожної канонічної діаграми графічних елементів, на діаграмах може бути зображена текстова інформація, яка розширює семантику базових елементів. В UML визначені три спеціальні механізми розширення: *стереотипи*, *помічені значення* і *обмеження*.

Стереотип (stereotype) – елемент моделі, який розширює семантику моделі. Стереотипи повинні базуватися на існуючих в UML типах або класах. Стереотипи призначені для розширення тільки семантики, а не структури вже описаних типів або класів. Деякі стереотипи вже визначені в UML, а інші можуть бути вказані розробником. На діаграмах вони зображуються в формі тексту у кутових дужках (напр., **<< interface >>**).

Помічені значення (tagged value) – це явне визначення властивості у вигляді «ім'я = значення». У поміченому значенні ім'я називають ***тегом*** (tag).

Помічені значення на діаграмах зображуються в формі тексту у фігурних дужках: **{тег = значення}**. Деякі теги визначені в нотації UML, але їх визначення не є строгим, тому теги можуть бути вказані розробником.

Обмеження (constraint) – деяка логічна умова, яка обмежує семантику обраного елемента моделі. Як правило, всі обмеження специфікує розробник. Обмеження на діаграмах зображуються в формі тексту у фігурних дужках.

1.2.3. Діаграми варіантів використання UML

Візуальне моделювання мовою UML можна представити як процес послідовного переходу від найбільш загальної і абстрактної концептуальної моделі системи до логічної, а потім і до фізичної моделі відповідної програмної системи. Для досягнення цієї мети спочатку будується модель в формі діаграми *варіантів використання*, яка описує функціональне призначення системи або, іншими словами, те, що система повинна робити в процесі свого функціонування.

Діаграма варіантів використання – це вихідне концептуальне представлення або концептуальна модель системи в процесі її проектування розробки. Створення діаграми варіантів використання переслідує наступні цілі:

– визначити загальні межі і контекст предметної області на початкових етапах проектування системи;

– сформулювати загальні вимоги до функціональної поведінки проектованої системи;

– розробити вихідну концептуальну модель системи для її подальшого деталізації в формі логічних і фізичних моделей;

– підготувати вихідну документацію для взаємодії розробників системи з її замовниками і користувачами.

Діаграма варіантів використання (use case diagram) – діаграма, яка відображає *відношення* між *акторами* і *варіантами використання* (сервісами).

Актор (actor) – будь-який об’єкт, суб’єкт або система, що взаємодіє з системою *ззовні*. Це може бути людина, технічний пристрій, програма або будь-яка інша система, яка є джерелом впливу на систему.

Актор представляє на діаграмі будь-яку зовнішню по відношенню до системи сутність, яка взаємодіє з системою і використовує її функціональні можливості для досягнення певних цілей або рішення окремих задач.

Стандартним графічним символом актора на діаграмах є фігурка «людини», під якою записується ім’я актора (рис. 1.19). Імена акторів повинні починатися з великої літери.

Рисунок 1.19 – Графічне позначення актора

Варіант використання призначений для опису сервісів, які система надає актору. Кожен варіант використання визначає дії, які виконує система при взаємодії з актором. При цьому не визначається яким чином буде реалізовано взаємодію. Зміст варіанта використання може бути представлено в формі додаткового пояснювального тексту, який розкриває сенс або семантику дій при виконанні даного варіанта використання. Такий пояснювальний текст має назву «**текст-сценарію**» або просто «**сценарій**».

Окремий варіант використання позначається на діаграмі еліпсом, всередині якого міститься його стисле **ім’я** в формі іменника (рис. 1.20, а) або дієслова (рис. 1.20, б) з пояснювальним текстом. Ім’я варіанта використання починається з великої літери.

Рисунок 1.20 – Графічне позначення варіанта використання

Кожен варіант використання відповідає окремому сервісу, який надає система по запиту актора, тобто визначає один з способів застосування системи. Сервіс, який ініційовано по запиту актора, повинен представляти завершену послідовність дій. Це означає, що після обробки запиту система повинна повернутися у вихідний стан, в якому вона готова до виконання наступних запитів.

Для зручності множину варіантів використання можна розглядати у якості окремого *пакету*.

Відношення на діаграмі варіантів використання

Відношення (relationship) – семантичний зв'язок між окремими елементами моделі.

Між елементами діаграми варіантів використання можуть існувати різноманітні *відношення*, які описують взаємодію екземплярів одних акторів і варіантів використання з екземплярами інших акторів та варіантів. Один актор може взаємодіяти з декількома варіантами використання. В свою чергу один варіант використання може взаємодіяти з декількома акторами, надаючи їм свій сервіс.

Два варіанта використання в межах однієї системы також можуть взаємодіяти один з одним, проте характер цієї взаємодії відрізняється від взаємодії з акторами. Проте в обох випадках способи взаємодії елементів моделі передбачають обмін *сигналами* або *повідомленнями*, які ініціюють реалізацію функціональної поведінки системи.

В мові UML визначено декілька стандартних видів *відношень*:

– асоціація (association relationship);
– включення (include relationship);
– розширення (extend relationship);
– узагальнення (generalization relationship).

Відношення асоціації призначене для відображення специфічної ролі актора при його взаємодії з окремим варіантом використання. На діаграмі варіантів використання, як і на інших діаграмах, відношення асоціації позначається суцільною лінією між актором і варіантом використання. Дана лінія може мати додаткові позначення, наприклад, ім’я і кратність (рис. 1.21).

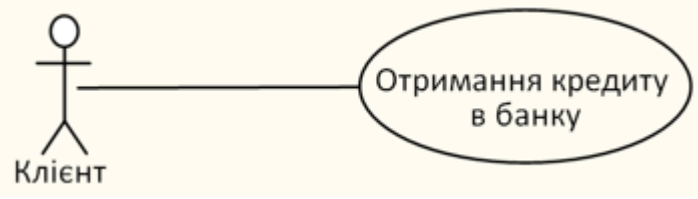


Рисунок 1.21 – Приклад графічного представлення асоціації між актором і варіантом використання

В контексті діаграми варіантів використання відношення асоціації між актором и варіантом використання може вказувати на те, що актор ініціює відповідний варіант використання. Такого актора називають ***головним***. В інших

випадках подібна асоціація може вказувати на актора, якому надається довідкова інформація про результати функціонування системи. Таких акторів часто називають ***другорядними***.

Включення (include) в UML – це різновид відношення залежності між базовим варіантом використання і його спеціальним випадком. Відношення включення встановлюється тільки між двома варіантами використання і зазначає те, що задана поведінка для одного *варіанта використання* включається у якості складового фрагменту в послідовність дій іншого варіанта використання. Дане відношення – це спрямоване бінарне відношення в тому сенсі, що пара екземплярів варіантів використання завжди упорядкована у відношенні включення. Наприклад, відношення включення, яке спрямоване від варіанта використання «Надання кредиту банком» до варіанта використання «Перевірка платоспроможності клієнта», вказує на те, що кожен екземпляр першого варіанта використання завжди включає до себе функціональну поведінку або виконання другого варіанта використання.

Графічно дане відношення позначається як відношення залежності в формі пунктирної лінії зі стрілкою, спрямованою від ***базового*** *варіанта використання* до *варіанту використання*, що ***включається***. Дана лінія помічається стереотипом **<<include>>**, як показано на рисунку 1.22.

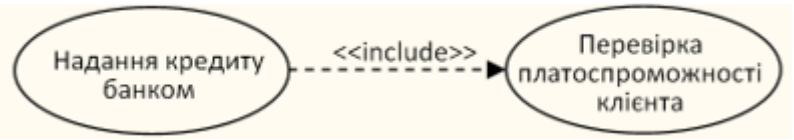


Рисунок 1.22 – Приклад графічного зображення *відношення включення* між варіантами використання

Відношення розширення (extend) визначає взаємозв’язок базового варіанту використання з іншим варіантом використання, функціональна поведінка якого задіюються базовим не завжди, а тільки при виконанні додаткових умов. В мові UML *розширення* представляє відношення, спрямоване до базового варіанту *використання* і з’єднане з ним в, так званій, *точці розширення*. Відношення *розширення* між варіантами використання позначається як відношення залежності в формі пунктирної лінії зі стрілкою, спрямованою від варіанту використання, який є *розширенням* для базового варіанту використання.

Дана лінія зі стрілкою повинна бути помічена стереотипом **<<extend>>**, як показано на рисунку 1.23.

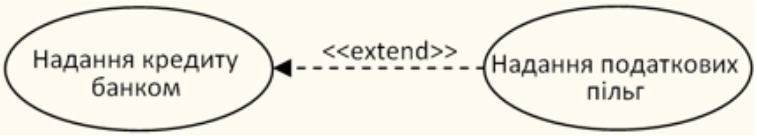


Рисунок 1.23 – Приклад графічного зображення *відношення розширення* між варіантами використання

У фрагменті на рис. 1.23 має місце відношення розширення між базовим варіантом використання «Надання кредиту банком» і варіантом використання «Надання податкових пільг». Це означає, що поведінка першого варіанту використання в деяких випадках може бути доповнена функціональністю другого варіанту використання. Це розширення має місце тільки при виконанні певної логічної умови, визначеної для даного відношення розширення.

Відношення узагальнення між варіантами використання застосовується в тому випадку, коли необхідно відмітити, що похідним (дочірнім) варіантам використання притаманні всі особливості поведінки базових (батьківських) варіантів. При цьому дочірні варіанти *використання* приймають участь в усіх відношеннях батьківських варіантів. Дочірні варіанти використання можуть наділятися новими властивостями поведінки, які відсутні у батьківських варіантів, а також уточнювати або модифікувати успадковані від них властивості поведінки.

Графічно відношення узагальнення позначається суцільною лінією зі стрілкою в формі незафарбованого трикутника, яка спрямована на батьківський варіант використання (рис. 1.24). Дана лінія зі стрілкою має спеціальну назву – **стрілка-узагальнення**.

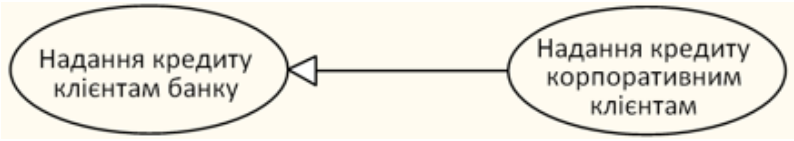


Рисунок 1.24 – Приклад графічного зображення *відношення узагальнення* між варіантами використання

В даному прикладі відношення *узагальнення* вказує на те, що варіант використання «Надання кредиту корпоративним клієнтам» – це спеціальний випадок варіанту використання «Надання кредиту клієнтам банку». Іншими словами, перший варіант використання є спеціалізацією другого *варіанту використання*. При цьому варіант використання «Надання кредиту клієнтам банку» ще називають ***пращуром*** (предком) або ***батьком*** по відношенню до варіанту використання «Надання кредиту корпоративним клієнтам», а останній варіант називають ***нащадком*** по відношенню до першого варіанту використання. Необхідно підкреслити, що нащадок успадковує всі властивості поведінки свого батька, а також може мати додаткові особливості поведінки.

Приклад діаграми варіантів використання для системи продажу товарів наведена на рисунку 1.25. Дана модель може бути застосована при створені і автоматизації відповідних інформаційних систем.

В якості акторів даної системи можуть розглядатися покупець і продавець. Покупець є клієнтом сервісу «Оформлення замовлення на покупку», а продавець бере участь в реалізації даного бізнес-процесу.

Діаграма варіантів використання для даної системи містить 5 варіантів використання, проміж якими встановлено відповідні відношення включення, розширення і узагальнення.

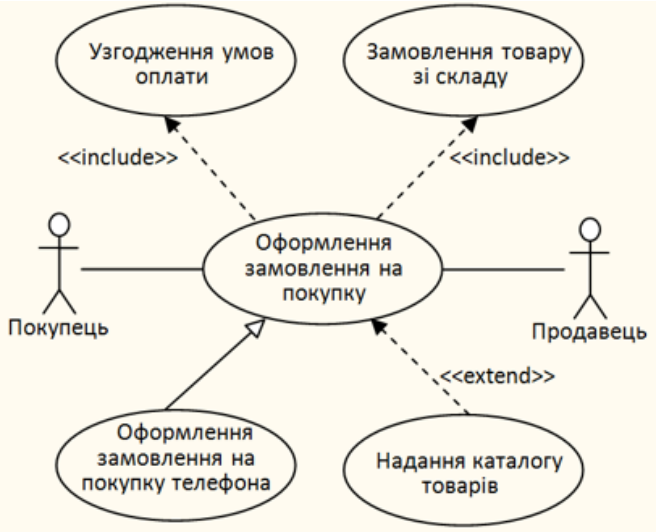


Рисунок 1.25 – Діаграма варіантами використання для системи продажу товарів за каталогом

Представлена система продажу товарів за каталогом є концептуальною моделлю типової бізнес-системи. При цьому ролі покупця і продавця в системі суттєво відрізняються. Покупець є зовнішнім стосовно системи суб'єктом, тоді як продавець є частиною бізнес-системи. Реалізація процесів не відображається на діаграмах варіантів використання. Для моделювання логічних та фізичних аспектів реалізації призначені інші типи канонічних діаграм, які будуть розглянуті у наступних розділах.

1.2.4. Діаграми класів UML

Центральне місце в методології ООП займає розробка логічної моделі системи у вигляді *діаграми класів*. Діаграма класів відображає різні взаємозв'язки між окремими сутностями предметної області, а також описує їх внутрішні структуру і типи відношень. Дана діаграма не містить інформації щодо часових аспектів функціонування системи. З цієї точки зору діаграма класів може слугувати подальшим розвитком концептуальної моделі (діаграми варіантів) системи, що проектується.

Діаграма класів (class diagram) – діаграма мови UML, на якій представлено сукупність статичних декларативних елементів моделі, таких як класи з атрибутами і операціями, а також відношення між класами.

Діаграма класів призначена для представлення статичної структури моделі системи в термінології класів об'єктно-орієнтованого програмування. При цьому вона може містити інтерфейси, пакети, відношення і навіть окремі екземпляри класів, такі як об'єкти. Коли говорять про дану діаграму, мають на увазі статичну структурну модель системи, тобто графічне представлення таких структурних взаємозв'язків логічної моделі системи, які не залежать від часу.

Клас (class) – абстрактний опис множини однорідних об'єктів, які мають однакові атрибути, операціє і відношення з об'єктами інших класів.

Графічно клас в нотації мови UML зображується у вигляді прямокутника, який додатково може бути розділений горизонтальними лініями на розділи (секції) (рис. 1.26). В цих секціях можуть вказуватися ім'я класу, атрибути и операції класу.

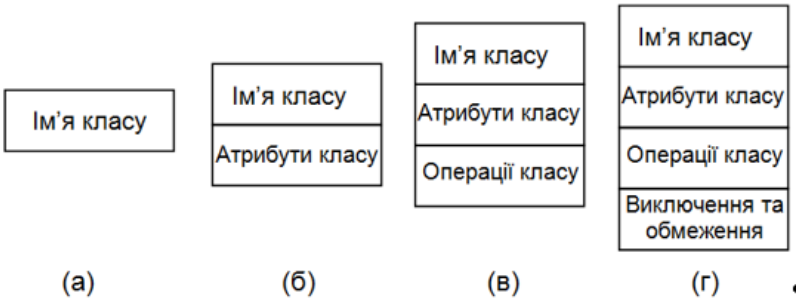


Рисунок 1.26 – Варіанти графічного зображення класу на *діаграмі класів*

На початкових етапах розробки діаграми окремі класи можуть позначатися простим прямокутником, в якому необхідно вказати ім'я відповідного класу (рис. 1.26, а). В подальшому при розробці окремих компонентів діаграми опис класів доповнюється *атрибутами* (рис. 1.26, б) і *операціями* (рис. 1.26, в). Четверта секція (рис. 1.26, г) не є обов'язковою і призначена для розміщення додаткової інформації довідкового характеру, наприклад, про виключення або обмеження класу, відомості про розробника або мову реалізації. Мається на увазі, що остаточний варіант діаграми містить найбільш повний опис класів, які складаються з трьох або чотирьох секцій.

Навіть якщо секції атрибутів і операцій не заповнено, в позначенні *класу* їх необхідно виділити горизонтальною лінією, для того щоб відрізнити клас від інших елементів мови UML. Приклади графічного зображення конкретних класів наведено на рис. 1.27. В першому випадку для класу **Коло** (рис. 1.27, а) вказано тільки його атрибути – крапка на координатній площині, яка визначає розташування його центру, та радіус. Для класу **Вікно** (рис. 1.27, б) вказано тільки його операції, при цьому секцію атрибутів залишено незаповненою. Для класу **Рахунок** (рис. 1.27, в) додатково зображено четверту секцію, в якій вказано вимогу – реалізувати резервне копіювання об'єктів цього класу.

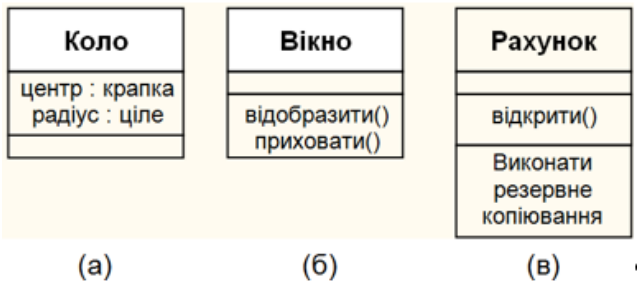


Рисунок 1.27 – Приклади графічного зображення класів

Ім’я класу

Ім’я класу має бути унікальним в межах пакету, який може містити одну або декілька діаграм класів. Ім’я вказується в верхній секції прямокутника, тому вона часто називається секцією імені класу. Крім того, ім’я класу записується в центрі секції імені напівжирним шрифтом і повинно починатися із великої літери. Рекомендується в якості імен класів використовувати іменники, що записуються без пропусків. Необхідно пам’ятати, що імена класів утворюють словник предметної області.

В секції імені класу можуть також знаходитися стереотипи або посилання на стандартні шаблони, від яких утворено даний клас і, відповідно, від яких він успадковує атрибути і операції. В цій секції може також вказуватися інформація про розробника даного класу і статус стану розробки. Тут також можуть записуватися загальні властивості цього класу, які мають відношення до інших класів діаграми або до стандартних елементів мови UML.

Клас може мати або не мати екземплярів (об’єктів). В залежності від цього в мові UML розрізняють *конкретні* і *абстрактні* класи.

Конкретний клас (concrete class) – клас, на основі якого можуть бути безпосередньо створені об’єкти (екземпляри класу).

Розглянуті вище позначення відносяться до конкретних класів. Від них слід відрізняти абстрактні класи.

Абстрактний клас (abstract class) – клас, який не має екземплярів або об’єктів.

Для позначення імені абстрактного класу використовується курсив. В мові UML прийнята загальна угода про те, що будь-який текст, який має відношення до абстрактного елемента, записується курсивом.

В деяких випадках необхідно явно вказувати, до якого пакету відноситься той чи інший клас. Для цієї цілі використовується спеціальний символ роздільник – подвійна двокрапка – '**::**'. Синтаксис рядка імені класу в цьому випадку буде такий: **<Ім’я пакету>::<Ім’я класу>**. Наприклад, якщо визначено пакет з ім’ям **Банк**, то клас **Рахунок** в цьому банку можна записати в вигляді: **Банк::Рахунок**.

Атрибути класу

Атрибут (attribute) – змістовна характеристика класу. Вона описує множину значень, які можуть приймати окремі об'єкти цього класу.

Атрибут класу призначений для представлення окремої властивості або ознаки, які є загальними для всіх об'єктів даного класу. Атрибути класу записуються в другій зверху секції прямокутника класу. Цю секцію часто називають секцією атрибутів.

В мові UML позначення атрибутів класу, підпорядкована певним синтаксичним правилам. Кожному атрибуту класу відповідає окремий рядок тексту, який складається з *квантора видимості атрибуту*, *імені атрибуту*, його *кратності*, *типу значень атрибута* і, можливо, його *вихідного значення*. Загальний формат запису окремого атрибуту класу такий:

<квантор видимості> <ім'я> [кратність] : <тип> = <вихідне значення> {рядок-властивість}.

Видимість (visibility) – якісна характеристика опису елементів класу. Вона характеризує потенційну можливість інших об'єктів моделі впливати на окремі аспекти поведінки даного класу.

Видимість в мові UML специфікується за допомогою ***квантора видимості*** (visibility), який може приймати одне з 4-х можливих значень і позначається спеціальними символами.

1) Символом '+' – позначається атрибут із областю видимості типу загальнодоступний (**public**). Атрибут з цією областю видимості доступний (видимий) з будь-якого іншого класу пакета, в якому визначено діаграму.

2) Символом '#' – позначається атрибут із областю видимості типу захищений (**protected**). Атрибут з цією областю видимості є недоступним або невидимим для всіх класів, за виключенням підкласів даного класу.

3) Символом '-' – позначається атрибут із областю видимості типу закритий (**private**). Атрибут з цією областю видимості недоступний або невидимий для всіх класів без виключення (звичайно, крім класу-власника даного атрибуту).

4) Символом '~' – позначається атрибут із областю видимості типу пакетний (**package**). Атрибут з цією областю видимості недоступний або невидимий для всіх класів за межами пакету, в якому визначено клас-власник даного атрибуту.

Замість зазначених умовних графічних позначень квантора видимості можна записувати відповідні ключові слова: **public**, **protected**, **private**, **package**.

Ім'я атрибуту представляє собою рядок тексту, який використовується в якості ідентифікатора відповідного атрибуту. Він має бути унікальним в межах даного *класу*. Ім'я *атрибуту* – єдиний обов'язковий елемент синтаксичного позначення атрибуту. Ім'я повинно починатися з *малої літери* і не матиме пропусків.

Кратність (multiplicity) – специфікація потужності множини значень, які може мати відповідний атрибут.

Кратність атрибуту характеризує кількість конкретних атрибутів даного типу. У загальному випадку кратність записується в формі рядка цифр в квадратних дужках після імені відповідного атрибуту. Цифри розділяються двома крапками: **[нижня межа .. верхня межа]**, де нижня і верхня межі позитивні цілі числа. Кожна така пара позначає окремий замкнутий інтервал цілих чисел, у якого нижня (верхня) межа дорівнює значенню **нижня межа** (**верхня межа**). В якості верхньої межі може використовуватися символ '*****' (зірочка), який означає довільне позитивне ціле число, тобто необмежене зверху значення кратності відповідного атрибуту.

Інтервалів кратності для окремого атрибуту може бути декілька. В цьому випадку їх спільне використання відповідає теоретико-множинному об'єднанню відповідних інтервалів. Значення кратності з інтервалу слідують в монотонно зростаючому порядку без пропуску окремих чисел, які знаходяться між нижньою і верхньою границею.

Якщо кратність позначено одним числом, то кратність атрибуту дорівнює даному числу. Якщо вказано тільки знак '*****', то кратність атрибуту може бути будь-яким позитивним цілим числом або нулем.

Тип атрибуту позначається виразом, семантика якого відповідає певному типу даних. В нотації UML тип атрибуту іноді залежить від мови програмування, який передбачається використовувати для реалізації даної моделі. Найчастіше тип атрибуту позначається рядком тексту, що має осмислене значення в межах пакету або моделі, до яких належить клас.

Вихідне значення використовується для встановлення початкового значення відповідного атрибуту при створенні окремого екземпляру (об'єкту) класу з дотриманням типу конкретного атрибуту. Якщо вихідне значення не

вказано, то значення відповідного атрибуту не визначено на момент створення нового екземпляру класу.

При створенні атрибутів можуть бути застосовані додаткові синтаксичні конструкції – це *підкреслення рядка* атрибуту, *пояснювальний текст у фігурних дужках* і *коса риска перед ім'ям* атрибуту. ***Підкреслення рядка атрибуту*** означає, що відповідний атрибут загальний для всіх об’єктів даного класу, тобто його значення в усіх створюваних об’єктів однакове (аналог ключового слова **static** в деяких мовах програмування).

Пояснювальний текст у фігурних дужках може означати різні конструкції. Якщо в цьому рядку є знак «дорівнює» ('='), то весь текст *рядок-властивість* використовується для встановлення вихідного значення для всіх екземплярів класу, яке не може бути перевизначене у подальшому. Відсутність рядка-властивості трактується як можливість зміни відповідного атрибуту в програмі.

Знак '/' перед іменем атрибуту вказує на те, що даний атрибут є *похідним* від деякого іншого атрибуту того ж класу. ***Похідний атрибут*** (derived element) – атрибут класу, значення якого для окремих об’єктів можна обчислити за допомогою значень інших атрибутів цього ж об’єкта.

Операції класу

Операція (operation) – це сервіс, який надається кожним екземпляром або об’єктом *класу* на вимогу своїх клієнтів, у якості яких можуть виступати інші об’єкти, в тому числі і екземпляри цього ж класу.

Операції класу записуються в третій зверху секції прямокутника класу, яку часто називають ***секцією операцій***. Сукупність операцій характеризує функціональний аспект поводження всіх об’єктів даного класу. Позначення операцій класу в мові UML також стандартизовано і підпорядковується певним синтаксичним правилам. При цьому кожній операції класу відповідає окремий рядок, який складається з *квантора видимості операції*, *імені операції*, *виразу типу значення*, що повертається операцією та, можливо, *рядка-властивості* даної операції. Загальний формат окремої операції класу такий:

<квантор видимості> <ім’я операції>(список параметрів):
<тип значення, що повертається> {рядок-властивість}

Квантор видимості, як і у атрибутів класу, може приймати одне з чотирьох можливих значень і відображається за допомогою спеціального символу або ключового слова. Символом '+' позначається операція з областю видимості типу загальнодоступний (**public**). Символом '#' позначається операція з областю видимості типу захищений (**protected**). Символ '**-**' використовується для позначення *операції* з областю видимості типу закритий (**private**). Символ '~' використовується для позначення операції з областю видимості типу пакетний (**package**).

Квантор видимості для операції може бути відсутнім. В цьому випадку його відсутність просто означає, що видимість операції не вказується. Замість умовних графічних позначень також можна записувати відповідне ключове слово: **public**, **protected**, **private**, **package**.

Ім'я операції – це рядок тексту, який використовується в якості ідентифікатора відповідної операції і тому повинна бути унікальною в межах даного класу. Ім'я операції – єдиний обов'язковий елемент синтаксичного позначення операції. Воно має починатися з малої літери, і, як правило, записуватися без пробілів.

Список параметрів є переліком розділених комами формальних параметрів, кожен з яких може бути представлений в наступному вигляді:

<напрямок> <ім'я> : <вираз типу> = <значення за замовчуванням>

Параметр (parameter) – специфікація змінної операції, яка може бути змінена, передана або повернена.

Параметр може включати *ім'я*, *тип*, *напрямок* і *значення за замовчуванням*. ***Напрямок параметра*** – це одне з ключових слів **in**, **out** або **inout** зі значенням **in** за замовчуванням, коли напрямок параметра не вказано. ***Ім'я параметра*** – це ідентифікатор відповідного формального параметру, при зазначенні якого слід дотримуватися правил для імен атрибутів. ***Вираз типу*** є специфікацією типу даних для допустимих значень відповідного формального параметру. ***Значення за замовчуванням*** – деяке конкретне значення для цього формального параметра.

Вираз типу значення, що повертається, вказує на тип даних, які повертаються об'єктом після виконання відповідної операції. Дві крапки і вираз типу значення, що повертається, можуть бути не вказані, якщо операція не повертає ніякого значення. Щоб вказати декілька значень, що повертаються,

даний елемент специфікації операції може бути записаний у вигляді списку окремих виразів.

Операція з областю дії «**весь клас**» вказується підкресленням імені і рядка виразу типу. В цьому випадку під областю дії операції розуміються всі об'єкти даного класу.

Рядок-властивість визначає властивості, які можуть бути застосовані до даної операції. Рядок-властивість може бути відсутнім, якщо властивості не специфіковано.

Список формальних параметрів і тип значення, що повертається, не є обов'язковими. Квантори видимості атрибутів і операцій можуть бути вказані у вигляді спеціальної позначки або символу, які використовуються для графічного представлення моделей в інструментальному засобі. Ще раз слід нагадати, що імена операцій, як і атрибутів, і параметрів, записуються з *малої літери*. При цьому обов'язковою частиною рядка запису операції є наявність імені *операції* і круглих дужок.

Розширення мови UML для побудови моделей програмного забезпечення

Однією з беззаперечних переваг мови UML є наявність механізмів розширення, які дозволяють ввести додаткові графічні позначення, орієнтовані на вирішення завдань з певної предметної області. Для розробки програмного забезпечення мова UML містить спеціальне розширення: The UML Profile for Software Development Processes (UML профіль для процесу розробки програмного забезпечення).

В рамках даного профілю запропоновано три спеціальні графічні примітиви, які можуть бути використані для уточнення семантики окремих класів при побудові різних діаграм (рис. 1.28):

– ***управляючий клас*** (control class) – клас, який відповідає за координацію дій інших класів. На кожній діаграмі класів повинен бути хоча б один управляючий клас, який контролює послідовність виконання дій певного варіанту використання. Як правило, даний клас є активним і ініціює розсилку повідомлень іншим класам моделі. Крім спеціального позначення управляючий *клас* може бути зображений у формі прямокутника класу зі стереотипом **<<control>>** (рис. 1.28, а);

– ***клас-сутність*** (entity class) – пасивний клас, інформація про який повинна зберігатися постійно і не знищуватися зі знищенням об'єктів даного класу або виключенням системи. Як правило, цей клас відповідає окремій таблиці бази даних. В цьому випадку його атрибутами є полями таблиці, а операціями - приєднаними або збереженими процедурами. Цей клас пасивний, він лише приймає повідомлення від інших класів моделі. Клас-сутність може бути зображений стандартним чином у формі прямокутника класу зі стереотипом **<<entity>>** (рис. 1.28, б);

– ***граничний клас*** (boundary class) – клас, який розташовується на межі системи з зовнішнім середовищем і безпосередньо взаємодіє з акторами, але є складовою частиною системи. Граничний клас може бути зображений також стандартним чином у формі прямокутника класу зі стереотипом **<<boundary>>** (рис. 1.28, в).

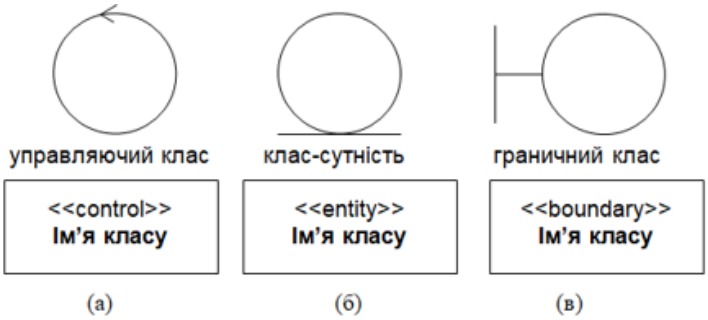


Рисунок 1.28 – Графічне зображення класів для моделювання програмного забезпечення

Інтерфейс

Інтерфейс (interface) – іменована множина *операцій*, які характеризують поведінку окремого елемента моделі.

Інтерфейс в контексті мови UML – це спеціальний випадок класу, у якого є операції, але відсутні атрибути. Для позначення інтерфейсу використовується спеціальний графічний символ коло або стандартний спосіб – прямокутник класу із стереотипом **<<interface>>** (рис. 1.29).

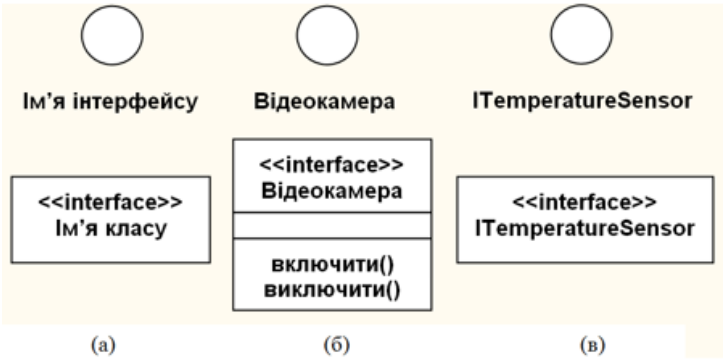


Рисунок 1.29 – Графічне зображення інтерфейсів на діаграмах класів

На діаграмі варіантів використання інтерфейс зображується у вигляді маленького кола, поруч з яким записується його ім'я (рис. 1.29, а). В якості імені може використовуватись іменник, який характеризує відповідну інформацію або сервіс, наприклад, «**Датчик температури**», «**Форма вводу**», «**Сирена**», «**Відеокамера**» (рис. 1.29, б). З урахуванням мови реалізації моделі ім'я інтерфейсу, як і імена інших класів, рекомендується записувати англійською і починати із великої букви, наприклад, **ITemperatureSensor** (рис. 1.29, в).

Інтерфейси на діаграмі використовуються для специфікації таких елементів моделі, які доступні ззовні, але їх внутрішня структура залишається прихованою від клієнтів. Інтерфейси не можуть містити а ні атрибутів, а ні станів, а ні спрямованих асоціацій. Вони містять тільки операції без зазначення особливостей їх реалізації. Формально інтерфейс не тільки відділяє специфікацію операцій системи від їх реалізації, але й визначає загальні границі системи, що проектується. В подальшому інтерфейс може бути уточнений явним визначенням тих операцій, які специфікують окремі аспекти поведінки системи

Відношення

Крім внутрішнього устрою класів важливу роль при розробці системи, що проектується, мають різні відношення між класами, які також можуть бути зображені на діаграмі класів. Сукупність допустимих типів таких відношень строго фіксована в мові UML і визначається самою семантикою таких відношень. Базові відношення, які зображуються на діаграмах класів:

– *відношення асоціації* (association relationship);
– *відношення узагальнення* (generalization relationship);
– *відношення агрегації* (aggregation relationship);

– *відношення композиції* (composition relationship).

Кожне з цих відношень має особисте графічне представлення, яке відображає семантичний характер взаємозв'язку між об'єктами відповідних класів.

Відношення асоціації

Асоціація (association) – семантичне відношення між двома і більше класами, які специфікують характер взаємозв'язків між відповідними екземплярами даних класів.

Відношення *асоціації* відповідає наявності довільного (будь-якого) відношення або взаємозв'язку між класами. Дане відношення позначається суцільною лінією з стрілкою або без неї і з додатковими символами, які характеризують спеціальні властивості асоціації. Асоціації було розглянуто при вивченні елементів діаграми варіантів використання, але стосовно діаграм класів семантика цього типу відношень значно ширша. В якості додаткових спеціальних символів можуть використовуватись *ім'я асоціації*, *символ навігації*, а також *імена* і *кратності класів-ролей асоціації*.

Ім'я асоціації – необов'язковий елемент її позначення. Проте, якщо воно задано, то записується с заголовної букви поруч з лінією асоціації. Окремі асоціації можуть відігравати певну роль у відповідному відношенні, на що явно показує ім'я кінцевих точок асоціації на діаграмі.

Найбільш простий випадок даного відношення – ***бінарна асоціація*** (binary association), яка служить для представлення довільного відношення між двома класами. Вона пов'язує в точності два різних класи і може бути неспрямованим (симетричним) або спрямованим відношенням. Окремий випадок бінарної асоціації – ***рефлексивна асоціація***, яка зв'язує клас з самим собою. Неспрямована бінарна асоціація зображується лінією без стрілки. Для неї на діаграмі може бути вказаний порядок читання з використанням значка в формі трикутника поруч з ім'ям даної асоціації.

Як простий приклад неспрямованої бінарної асоціації можна розглянути відношення між двома класами – класом **Компанія** и класом **Співробітник** (рис. 1.30). Вони пов'язані бінарною асоціацією **Працює**, ім'я якої зазначено на малюнку поруч з лінією асоціації. Для даного відношення визначено такий напрямок читання – співробітник працює в компанії.

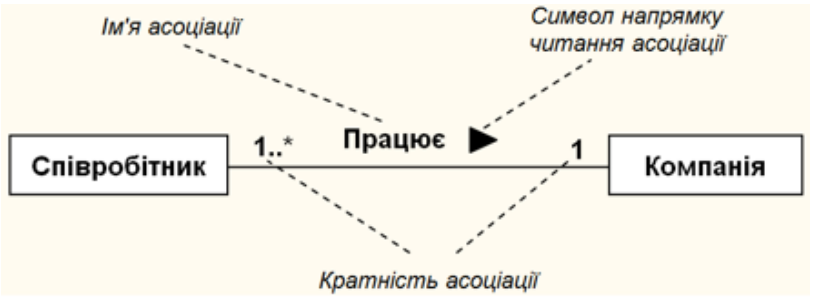


Рисунок 1.30 – Графічне зображення неспрямованої бінарної асоціації

Спрямована бінарна асоціація зображується суцільною лінією з простою стрілкою на одній з її кінцевих точок. Напрямок цієї стрілки вказує на те, який клас є першим, а який – другим.

В якості прикладу спрямованої бінарної асоціації можна розглянути відношення між двома класами: **Клієнт** та **Рахунок** (рис. 1.31). Вони пов'язані між собою бінарною асоціацією з ім'ям **Має**, для якої визначено порядок проходження класів. Це означає, що конкретний об'єкт класу **Клієнт** завжди повинен зазначатися першим при розгляді взаємозв'язку з об'єктом класу **Рахунок**.

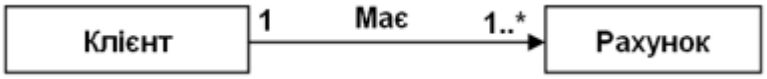


Рисунок 1.31 – Графічне зображення спрямованої бінарної асоціації

Окремий випадок відношення асоціації – так звана *виключна асоціація* (Xor-association). Семантика даної асоціації вказує на те, що з кількох потенційно можливих варіантів даної асоціації в кожен момент часу може використовуватися тільки одна. На діаграмі класів виключна асоціація зображується пунктирною лінією, що з'єднує дві або більше асоціацій. (рис. 1.32), поруч з якою записується обмеження у вигляді рядка тексту в фігурних дужках: **{xor}**.

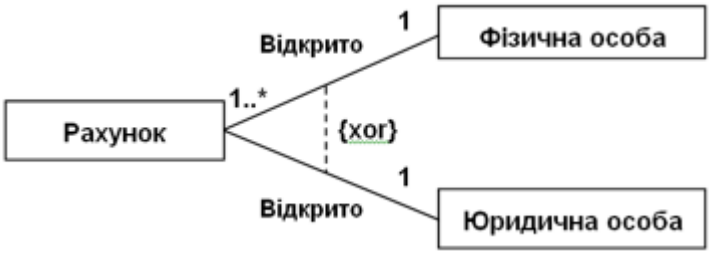


Рисунок 1.32 – Графічне зображення виключної асоціації між трьома класами

Асоціація більш високої арності називається *n-арною асоціацією*.

n-арна асоціація (n-ary association) – асоціація між трьома і більшим числом класів.

Кожен екземпляр такої асоціації являє собою впорядковану множину (кортеж), що містить ***n*** екземплярів з відповідних класів. Така асоціація зв'язує відношенням більш ніж три класи, при цьому клас може брати участь в асоціації більш ніж один раз. Кожен екземпляр n-арной асоціації являє собою n-арний кортеж, що складається з об'єктів відповідних класів. У цьому контексті бінарна асоціація є окремим випадком n-арной асоціації, коли значення n = 2, але має власне позначення. Бінарна асоціація – це спеціальний випадок n-арной асоціації.

Графічно n-арна асоціація позначається ромбом, від якого ведуть лінії до символів класів даної асоціації. Сам же ромб з'єднується з символами класів суцільними лініями. Зазвичай лінії проводяться від вершин ромба або від середини його сторін. Ім'я n-арної асоціації записується поряд з ромбом відповідної асоціації. Однак порядок класів в n-арній асоціації на діаграмі не фіксується.

Як приклад тернарної асоціації можна розглянути відношення між трьома класами: **Співробітник**, **Компанія** та **Проект**. Дана асоціація вказує на наявність відношення між цими трьома класами, яке може представляти інформацію про проекти, що реалізуються в компанії, і про співробітників, що беруть участь у виконанні окремих проектів (рис. 1.33).

Клас може бути приєднаний до лінії асоціації пунктирною лінією. Це означає, що даний клас забезпечує підтримку властивостей відповідної n-арної асоціації, а сам n-арна асоціація має атрибути, операції та/або асоціації. Іншими словами, така асоціація є класом із відповідним позначенням у вигляді прямокутника і самостійним елементом мови UML – *класом-асоціацією*.

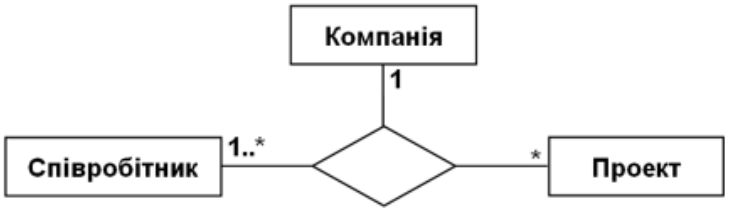


Рисунок 1.33 – Графічне зображення тернарної асоціації між трьома класами

Клас-асоціація (association class) – модельний елемент, який одночасно є асоціацією і класом. Клас-асоціація може розглядатися як асоціація, яка має властивости класу, або як клас, що має також властивості асоціації.

Окремий клас в асоціації може відігравати певну роль в даній асоціації. Ця роль може бути явно специфікована на діаграмі класів. З цією метою в мові UML вводиться у розгляд спеціальний елемент – *кінцева точка асоціації* або *кінець асоціації*, який графічно відповідає точці з'єднання лінії асоціації з окремим класом.

Кінець асоціації (association end) – кінцева точка асоціації, яка зв'язує асоціацію з класифікатором.

Кінець *асоціації* – частина асоціації, а не класу. Кожна асоціація може мати два або більше кінців асоціації. Найбільш важливі властивості асоціації позначаються на діаграмі поруч з цими елементами асоціації і повинні розміщуватися разом з ними. Однією з таких додаткових позначок є *ім'я ролі* окремого класу, який входить в асоціацію.

Роль (role) – специфічне поводження деякої сутності, яке розглядається у певному контексті і має власне ім'я. Роль може бути статичною або динамічною.

Ім'я ролі представляє собою рядок тексту поруч з кінцем асоціації для відповідного класу. Вона вказує на специфічну роль класу, який є кінцем відповідної асоціації. Ім'я ролі не обов'язковий елемент позначень і може бути відсутнім на діаграмі.

Наступний елемент позначень – *кратність асоціації*. Кратність відноситься до кінців асоціації і позначається у вигляді інтервалу цілих чисел, аналогічно кратності атрибутів і операцій класів, але без прямих дужок. Цей інтервал записується поруч з кінцем відповідної асоціації і означає потенціальне число окремих екземплярів класу, які можуть мати місце, коли інші екземпляри або об'єкти класів фіксовані.

Так, для прикладу (рис. 1.33) кратність '**1**' для класу **Компанія** означає, що кожен співробітник може працювати тільки в одній компанії. Кратність '**1..***' для класу **Співробітник** означає, що в кожній компанії можуть працювати декілька співробітників, загальна кількість яких заздалегідь невідома і нічим не обмежена. Замість кратності '**1..***' неприпустимо записувати тільки символ '*', тому що останній означає кратність '**0..***'. Для даного прикладу це мало б значення, що окремі компанії можуть взагалі не мати співробітників у своєму штаті. Така

кратність припустима в ситуаціях, коли в компанії взагалі не виконується ніяких проектів.

Ім’я асоціації розглядається у якості окремого атрибуту у відповідних класах асоціацій і може бути вказано на діаграмі явним чином в визначеній секції прямокутника класу.

Асоціація є найбільш загальною формою відношень в мові UML. Всі інші типи відношень можна вважати окремим випадком даного відношення. Проте важливість відокремлення специфічних семантичних властивостей і додаткових характеристик для інших типів відношень обумовлюють необхідність їх окремого вивчення при побудові діаграм класів. Дані відношення мають спеціальні позначення і відносяться до базових понять мови UML.

Відношення узагальнення

Відношення узагальнення є звичайним таксономічним відношенням або відношенням класифікації між більш загальним елементом (батьком або пращуром (предком)) й більш окремим (частковим) або спеціальним елементом (дочірнім або нащадком).

Таксономія – наука про принципи та способи класифікації й номенклатури складноорганізованих ієрархічних систем дійсності: органічного світу, об'єктів географії, геології, мовознавства, суспільства тощо.

Узагальнення (generalization) – таксономічне відношення між більш загальним поняттям та менш загальним поняттям.

Менш загальний елемент моделі повинен бути узгоджений з більш загальним елементом і може містити додаткову інформацію. Дане відношення використовується для представлення ієрархічних взаємозв'язків між різними елементами мови UML, такими як пакети, класи, варіанти використання.

Стосовно діаграми класів дане відношення описує ієрархічну структуру класів та успадкування їх властивостей і поведінки.

Успадкування (inheritance) – спеціальний концептуальний механізм, за допомогою якого більш спеціальні елементи включають в себе структуру і поведінку більш загальних елементів.

Відповідно до одного з головних принципів методології ООАП – успадкування, клас-нащадок має всі властивості і поведінку класу-батька, а

також має власні властивості і поведінку, які можуть бути відсутніми у класа-батька.

Батько, ***пращур***, ***предок*** (parent) – у відношенні узагальнення – більш загальний елемент. ***Нащадок*** (child) – спеціалізація одного з елементів відношення узагальнення.

Відношення узагальнення позначається на діаграмах суцільною лінією з трикутною стрілкою на одному з кінців (рис. 1.34). Стрілка вказує на більш загальний клас (клас-предок або суперклас), а її початок – на більш спеціальний клас (клас-нащадок або підклас).

Рисунок 1.34 – Графічне зображення відношення узагальнення в UML

Від одного класу-предку одночасно можуть успадковувати декілька класів-нащадків, що відображає таксономічний характер даного відношення. В даному випадку на діаграмі класів для відношення узагальнення вказується декілька ліній зі стрілками.

Наприклад, клас **Транспортний засіб** (курсив позначає абстрактний клас) може виступати в якості суперкласу для підкласів, які відповідають конкретним транспортним засобам, таким як: **Автомобіль**, **Автобус**, **Трактор** і іншим. Це може бути представлено графічно у формі діаграми класів наступного виду (рис. 1.35).

Рисунок 1.35 – Графічне зображення відношення узагальнення для декількох класів-нащадків

З метою спрощення позначень на діаграмі класів і зменшення кількості стрілок-трикутників і ліній, що позначають одне і те ж відношення узагальнення,

може бути просто зображена стрілка. У цьому випадку окремі лінії сходяться до стрілки, яка має з цими лініями єдину точку перетину (рис. 1.36).

Рисунок 1.36 – Альтернативний варіант графічного зображення відношення узагальнення класів у випадку об’єднання окремих ліній

Графічне зображення відношення узагальнення за формою відповідає графу спеціального виду, а саме – *ієрархічному дереву*. В цьому випадку клас-предок є *коренем дерева*, а класи-нащадки – його *листям*. Відмінність полягає в можливості позначення на діаграмі класів додаткової семантичної інформації, яка може відображати різні теоретико-множинні характеристики даного відношення. При цьому клас-предок на діаграмі може займати довільне положення щодо своїх класів-нащадків, яке визначається лише міркуваннями зручності.

До стрілки узагальнення може бути приєднано рядок тексту, що вказує на спеціальні властивості цього відношення у формі обмеження. Цей текст буде відноситись до всіх ліній узагальнення, які йдуть до класів-нащадків. Зазначена властивість стосується всіх підкласів даного відношення і має бути вказана у формі обмеження у *фігурних дужках*.

В якості обмежень використовуються наступні ключові слова мови UML:

– **{complete}** – означає, що в даному відношенні узагальнення специфіковані *всі* класи-нащадки і інших класів-нащадків у даного класу-предку бути не може.

– **{incomplete}** – означає випадок, протилежний першому. А саме, передбачається, що на діаграмі зазначено *не всі* класи-нащадки. У подальшому розробник може заповнити їх перелік, не змінюючи вже побудовану діаграму.

– **{disjoint}** – означає, що класи-нащадки не можуть містити об'єктів, які одночасно є екземплярами двох або більше класів.

– **{overlapping}** – випадок, протилежний попередньому. А саме, передбачається, що окремі екземпляри класів-нащадків можуть належати одночасно кільком класам.

З урахуванням використання обмежень діаграма класів (рис. 1.36) може бути уточнена (рис. 1.37).

Рисунок 1.37 – Варіант уточненого графічного зображення відношення узагальнення класів із використанням рядка-обмеження

Відношення агрегації

Агрегація (aggregation) – спеціальна форма асоціації, яка служить для представлення відношення типу «частина-ціле» між агрегатом (ціле) і його складовою частиною.

Відношення агрегації має місце між декількома класами в тому випадку, коли один з класів є сутність, яка включає в себе в якості складових частин інші сутності. Дане відношення має фундаментальне значення для опису структури складних систем, оскільки застосовується для представлення системних взаємозв'язків типу «частина-ціле». Розкриваючи внутрішню структуру системи, відношення агрегації показує, з яких елементів складається система, і як вони пов'язані між собою.

З погляду моделі окремі частини системи можуть виступати у вигляді, як елементів, так і підсистем, які, в свою чергу, теж можуть складатися з підсистем або елементів. Таким чином, дане відношення по своїй суті описує декомпозицію або розбиття складної системи на більш прості складові частини, до яких також може бути застосовано декомпозицію, коли в цьому виникне необхідність.

Розподіл системи на складові частини, що розглядається в такому аспекті, є ієрархією, але принципово відмінну від тієї, яка породжується відношенням узагальнення. Відмінність полягає в тому, що частини системи не повинні наслідувати її властивості і поведінку, оскільки є самостійними сутностями.

Більш того, частини цілого мають власні атрибути і операції, які суттєво відрізняються від атрибутів і операцій цілого.

Графічно відношення агрегації зображується суцільною лінією, один з кінців якої являє собою незафарбований ромб. Цей ромб вказує на клас, який являє собою «ціле» або клас-контейнер. Решта класів є його «частинами» (рис. 1.38).

Рисунок 1.38 – Графічне зображення відношення агрегації в UML

Як приклад відношення агрегації можна розглянути взаємозв'язок типу «частина-ціле», який має місце між класом **Системний блок ПК** і його складовими частинами: **Процесор**, **Оперативна пам'ять** і **Жорсткий диск**. Використовуючи позначення мови UML, компонентний склад системного блоку можна представити у вигляді відповідної діаграми класів (рис. 1.39), яка в даному випадку ілюструє відношення агрегації.

Рисунок 1.39 – Відношення агрегації на діаграмі класів
(на прикладі системного блоку ПК)

Відношення композиції

Композиція (composition) – різновид відношення агрегації, при якій складові частини цілого мають такий же час життя, як і ціле. Ці частини знищуються разом зі знищенням цілого.

Відношення композиції – частковий випадок відношення агрегації. Це відношення використовується для специфікації більш сильної форми відношення «частина-ціле», при якій складові частини тісно взаємопов'язані з цілим. Особливість цього взаємозв'язку полягає в тому, що частини не можуть існувати

у відриві від цілого, тобто зі знищенням цілого знищуються і всі його складові частини.

Композит (composite) – клас, який пов'язаний відношенням композиції з одним або більшою кількістю класів.

Графічно відношення композиції зображується суцільною лінією, один з кінців якої являє собою зафарбований ромб. Цей ромб вказує на клас, який представляє собою клас-композит. Решта класів є його «частинами» (рис. 1.40).

Рисунок 1.40 – Графічне зображення відношення композиції в UML

Можливо, найбільш наочним прикладом цього відношення є вікно графічного інтерфейсу програми, яке може складатися з рядка заголовка, смуг прокрутки, головного меню, робочої області і рядка стану. Вікно являє собою клас, а його складові елементи також є окремими класами. Складові вікна існують за умови існування самого вікна.

Для відношень композиції та агрегації можуть використовуватися додаткові позначення, які застосовуються для відношення асоціації. А саме, можуть зазначатися кратності окремих класів, які в загальному випадку не є обов'язковими. Згідно описаного вище прикладу клас **Вікно програми** є класом-композитом, а взаємозв'язки складових його частин можуть бути зображені наступною діаграмою класів (рис. 1.41).

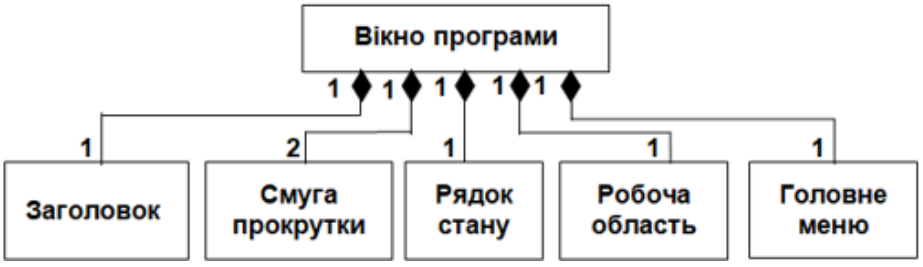


Рисунок 1.41 – Відношення композиції на діаграмі класів
(на прикладі класу-композиту **Вікно програми**)

Рекомендації із розробки діаграм класів

Процес розробки діаграми класів займає центральне місце при розробці проектів складних систем. Від уміння правильно вибрати класи і встановити між

ними взаємозв'язки часто залежить не тільки успіх процесу проектування, а й продуктивність виконання програми. Як показує практика ООАП, кожен програміст в тій чи іншій мірі використовує особистий досвід при розробці нових проектів. Це обумовлено бажанням звести нову задачу до вже вирішених, щоб мати можливість застосовувати не тільки перевірені фрагменти програмного коду, але й окремі компоненти або бібліотеки компонентів.

При визначенні класів, атрибутів і операцій, завданні їх імен і типів перед розробниками завжди постає питання: яку мову взяти за основу, свою рідну або англійську? З одного боку, використання рідної мови для опису моделі здається найбільш природним способом її подання і найбільшою мірою відображає комунікативну функцію моделі системи. З іншого боку, розробка моделі всього лише один з етапів створення відповідної системи, а застосування інструментальних засобів для її реалізації в абсолютній більшості випадків вимагає використання англомовних термінів. Саме тому виникає характерна неоднозначність, з якої, мабуть, абсолютно незнайома англомовна аудиторія.

При побудові діаграми варіантів використання, загальної концептуальної моделі проектованої системи, застосування термінів рідної мови виправдано з точки зору опису структури предметної області. Це сприяє взаєморозумінню, а також ефективному спілкуванню між замовником і розробником проекту. При побудові інших типів діаграм слід дотримуватися розумного компромісу.

Після розробки діаграми класів процес ООАП може бути продовжений в двох напрямках. З одного боку, якщо поведінка системи тривіальна, то можна приступити до розробки діаграм кооперації і послідовності. Однак для складних динамічних систем, зокрема для паралельних систем і систем реального часу, найважливішим аспектом їх функціонування є поведінка. Специфіка поведінки таких систем може бути представлена на діаграмах станів і діяльності, розробка яких в загальному випадку не обов'язкова, а визначається конкретним проектом.

1.2.5. Діаграми кооперації UML

Діаграма кооперації призначена для опису поведінки системи на рівні окремих об'єктів, які обмінюються повідомленнями, щоб досягти потрібної мети або реалізувати певний варіант використання. З погляду аналітика або архітектора системи в проекті важливо уявити структурні зв'язки окремих

об'єктів між собою. Таке уявлення структури моделі як сукупності взаємодіючих об'єктів і забезпечує діаграма кооперації.

Кооперація (collaboration) – специфікація множини об'єктів окремих класів, які спільно взаємодіють з метою реалізації окремих варіантів використання.

Поняття кооперації – одне з фундаментальних в мові UML. Мета самої кооперації полягає в тому, щоб специфікувати особливості реалізації окремих варіантів використання або найбільш значущих операцій в системі. Кооперація визначає поведінку системи в термінах взаємодії учасників цієї кооперації.

На діаграмі кооперації розміщуються об'єкти, що представляють собою екземпляри класів, зв'язки між ними, які є екземплярами асоціацій і повідомлення. Зв'язки доповнюються стрілками повідомлень, при цьому показуються тільки ті об'єкти, які беруть участь в реалізації кооперації, що моделюється. Далі, як і на діаграмі класів, показуються структурні відносини між об'єктами у вигляді різних сполучних ліній. Зв'язки можуть доповнюватися іменами ролей, які грають об'єкти в цьому зв'язку. І, нарешті, зображуються динамічні взаємозв'язки – потоки повідомлень в формі стрілок з вказівкою напрямку поруч з сполучними лініями між об'єктами, при цьому задаються імена повідомлень і їх порядкові номери в загальній послідовності повідомлень.

Одна і та ж сукупність об'єктів може брати участь в реалізації різних кооперацій. Залежно від розглянутої кооперації, можуть змінюватися як зв'язки між окремими об'єктами, так і потік повідомлень між ними. Саме це відрізняє діаграму кооперації від діаграми класів, на якій повинні бути вказані всі без винятку класи, їх атрибути і операції, а також всі асоціації та інші структурні відношення між елементами моделі.

Об’єкти та їх графічне зображення

Об'єкт (object) – сутність з визначеними межами і індивідуальністю, яка інкапсулює стан і поведінку.

В UML будь-який об'єкт є екземпляром класу, представленого на діаграмі класів. Об'єкт створюється на етапі реалізації моделі або виконання програми. Він має ***власне ім'я*** і конкретні значення атрибутів.

Для діаграм кооперації повне ім'я об'єкта являє собою рядок тексту, розділений двокрапкою і записану в форматі:

<Власне ім’я об’єкта >'/'<Ім’я ролі класу>:<Ім’я класу >

Ім'я ролі класу вказується в тому випадку, коли відповідний клас відсутній в моделі або розробнику необхідно акцентувати увагу на особливості його використання в даному контексті моделювання взаємодії. ***Ім'я класу*** – це ім'я одного з класів, представленого на діаграмі класів. Важливо відзначити, що весь запис імені об'єкта підкреслюється, що є візуальним ознакою об'єктів на різних діаграмах мови UML.

Якщо вказано власне ім’я об’єкта, то воно повинно починатися з *малої літери*. В той же час ім’я об’єкта, ім’я ролі з символом '/' або ім’я класу можуть бути відсутні. Однак двокрапка завжди має стояти перед ім'ям класу, а коса риска – перед ім'ям ролі.

Таким чином, на діаграмах кооперації можуть зустрітися такі варіанти можливих записів повного імені об’єкту:

– '**o : C**' – об'єкт з власним ім'ям '**o**', екземпляр класу '**C**';

– ' **: C**' – анонімний об'єкт, екземпляр класу '**C**';

– '**o :** ' (або просто '**o**') – об'єкт-сирота з власним ім'ям «**o**»;

– '**o / R : C**' – об'єкт з власним ім'ям '**o**', екземпляр класу '**C**', який грає роль «**R**»;

– ' **/ R : C**' – анонімний об'єкт, екземпляр класу '**C**', який грає роль '**R**';

– '**o / R** ' – об'єкт-сирота з власним ім'ям '**o**', який грає роль '**R**';

– ' **/ R** ' – анонімний об'єкт і одночасно об'єкт-сирота, який грає роль '**R**'.

Приклади зображення об'єктів на діаграмах кооперації наведено на рисунку 1.42.

Рисунок 1.42 – Приклади графічних зображень об’єктів на діаграмах кооперації

Якщо власне ім'я об'єкта відсутнє, то такий об'єкт прийнято називати ***анонімним***. Однак в цьому випадку обов'язково ставиться двокрапка перед ім'ям відповідного класу (рис. 1.42, в). Відсутнім може також бути ім'я класу – такий об'єкт називається ***сиротою***. Для нього записується тільки власне ім'я об'єкта, двокрапка не ставиться, ім'я класу не вказуються (рис. 1.42, г). Якщо для об'єктів вказуються атрибути, то в більшості випадків вони приймають конкретні значення (рис. 1.42, б). Для окремих об'єктів (рис. 1.42, д, е) можуть бути додатково вказані ролі, які вони відіграють в кооперації.

В контексті мови UML всі об'єкти поділяються на дві категорії: пасивні та активні. ***Пасивний об'єкт*** оперує тільки даними і не може ініціювати діяльність з управління іншими об'єктами. Однак пасивні об'єкти можуть посилати сигнали в процесі виконання запитів, які вони обробляють. На діаграмі кооперації пасивні об'єкти зображуються звичайним чином без додаткових стереотипів.

Активний об'єкт (active object) має власний процес управління і може ініціювати діяльність з управління іншими об'єктами.

Активний об'єкт на діаграмі кооперації позначаються прямокутником з потовщеними межами (рис. 1.43). Кожен активний об'єкт є власником певного процесу управління. У цьому фрагменті діаграми кооперації активний об'єкт '**а: Клієнт**' «**а: Клієнт»** є ініціатором відкриття рахунку, який представлений анонімним об'єктом '**: Рахунок**'.

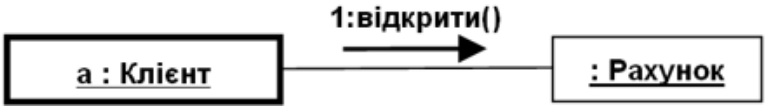


Рисунок 1.43 – Графічне зображення активного об’єктів (ліворуч) на діаграмі кооперації

Мультиоб'єкт (multiobject) – множина анонімних об’єктів, які можна утворити з одного класу.

На діаграмі кооперації мультиоб'єкт використовується для того, щоб показати операції і сигнали, які адресовані всій множині анонімних об'єктів. Мультиоб'єкт зображується двома прямокутниками, один з яких виступає з-за іншого (рис. 1.44, а). При цьому стрілка взаємозв'язку відноситься до всієї множини об'єктів, які позначає даний мультиоб'єкт. На діаграмі кооперації може

бути явно вказано відношення агрегації (композиції) між мультиоб'єктом і окремим об'єктом з його множини (рис. 1.44, б).

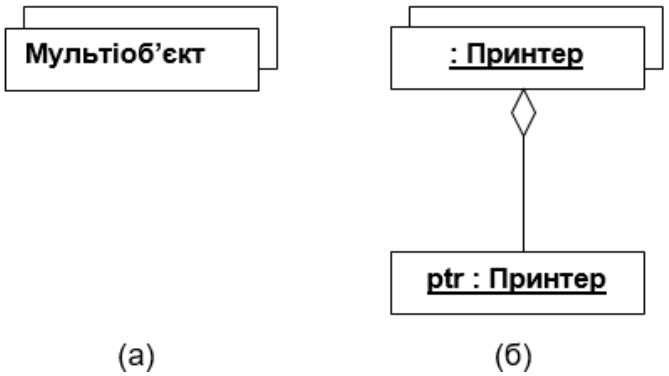


Рисунок 1.44 – Графічне зображення мультіоб’єктів на діаграмі кооперації

У наступному прикладі розглядається ситуація з відправкою поштового повідомлень клієнту з редактора електронної пошти (рис. 1.45). Анонімний активний об'єкт класу **РедакторEmail** спочатку посилає повідомлення анонімному мультиоб'єкту класу **Клієнт**. Це повідомлення ініціює вибір єдиного об'єкта класу **Клієнт**, який задовольняє додатковим умовам. Після цього вибраному об'єкту надсилається повідомлення про необхідність відправити електронного листа.

Рисунок 1.45 – Фрагмент діаграми кооперації для вибору адреси клієнта та відправлення електронного листа

Складений об'єкт (composite object) або об'єкт-композит призначений для представлення об'єкта, що має власну структуру і внутрішні потоки (нитки) управління.

Складений об'єкт є екземпляром класу-композиту, який зв'язаний відношенням композиції зі своїми частинами. Аналогічні відношення зв'язують між собою і відповідні об'єкти. На діаграмах кооперації такий складений об'єкт

зображується як звичайний об'єкт, який складається з двох секцій: верхньої і нижньої. У верхній секції записується ім'я складеного об'єкта, а в нижній – його об'єкти-частини замість списку атрибутів (рис. 1.46). При цьому допускається мати в якості частин інші складені об'єкти.

Рисунок 1.46 – Графічне зображення складеного об’єкта на діаграмі кооперації

Відношення між об'єктами на діаграмі кооперації описуються за допомогою зв'язків, які є екземплярами відповідних асоціацій.

Зв'язки на діаграмі кооперації

Зв'язок (link) – будь-яке семантичне відношення між деякою сукупністю об'єктів.

Зв'язок як елемент мови UML є екземпляром або прикладом довільної асоціації і може мати місце між двома і більше об'єктами. Бінарний зв'язок на діаграмі кооперації зображується відрізком суцільної лінії, що з'єднує два об'єкти (рис. 1.44). На кінцях цієї лінії додатково можуть бути явно вказані імена ролей відповідної асоціації.

Зв'язки не мають власних імен, оскільки є екземплярами деякої асоціації. Іншими словами, всі зв'язки на діаграмі кооперації можуть бути тільки анонімними і при необхідності записуються без двокрапки перед ім'ям асоціації. Однак найчастіше імена зв'язків на діаграмах кооперації не вказуються. Для зв'язків не вказується також і кратність кінцевих точок. Проте інші позначення спеціальних випадків відносин, такі як агрегація і композиція, можуть бути присутніми на окремих кінцях зв'язків. Наприклад, символ зв'язку типу агрегації між мультиоб'єктом класу **Клієнт** і окремим анонімним об'єктом класу **Клієнт** (рис. 1.44).

Приклади зв'язків з різними *стереотипами* показано на рисунку 1.47. Тут представлена узагальнена схема компанії з ім'ям '**c**', яка складається з департаментів (анонімний мультиоб'єкт класу **Департамент**). В останні входять співробітники (анонімний мультиоб'єкт класу **Співробітник**). Рефлексивний зв'язок вказує на те, що керівник департаменту є одночасно і його співробітником.

Рисунок 1.47 – Графічне зображення зв’язків з різними стереотипами

Як було зазначено вище, особливості моделювання взаємодії в контексті мови UML полягають в тому, щоб специфікувати комунікацію між множиною взаємодіючих об'єктів. Кожна взаємодія описується сукупністю повідомлень, якими обмінюються між собою об'єкти, що беруть участь у взаємодії.

Повідомлення і їх графічне зображення

Повідомлення (message) – специфікація передачі інформації від одного елемента моделі до іншого з очікуванням виконання певних дій з боку приймаючого елемента.

При цьому перший об'єкт очікує, що після отримання повідомлення другим об'єктом настане виконання деякої дії. На діаграмі кооперації повідомлення є причиною або стимулом початку виконання операцій, відправки сигналів, створення і знищення окремих об'єктів. Зв'язок забезпечує канал для спрямованої передачі повідомлень між об'єктами: від об'єкта-відправника до об'єкта-одержувача.

У цьому сенсі повідомлення являє собою фрагмент інформації, який відправляється одним об'єктом до іншого. При цьому прийом повідомлення може ініціювати виконання певних дій, спрямованих на вирішення окремого завдання тим об'єктом, якому це повідомлення надіслано. Повідомлення не тільки передають інформацію, а й вимагають (або припускають) від приймаючого об'єкта виконання певних дій. Повідомлення можуть ініціювати виконання операцій об'єктом відповідного класу, а параметри цих операцій передаються разом з повідомленням.

У такому контексті кожне повідомлення має напрямок від об'єкта, який ініціює та відправляє повідомлення, до об'єкта, який його отримує. Іноді відправника повідомлення називають клієнтом, а одержувача – сервером. При цьому повідомлення від клієнта має форму запиту деякого сервісу, а реакція сервера на запит після отримання повідомлення може бути пов'язана з виконанням певних дій або передачі клієнту необхідної інформації теж у формі повідомлення.

Повідомлення в мові UML також специфікують ролі, які грають об'єкти – відправник і одержувач повідомлення. Повідомлення на діаграмі кооперації зображуються додатковими стрілками поруч з відповідним зв'язком або роллю

асоціації. Напрямок стрілки вказує на одержувача повідомлення. Зовнішній вигляд стрілки повідомлення має певний сенс. На діаграмах кооперації може використовуватися один з трьох типів стрілок для позначення повідомлень (рис. 1.48).

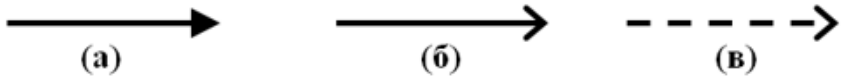


Рисунок 1.48 – Графічне зображення різних типів повідомлень на діаграмі кооперації

1) *Суцільна лінія з трикутною стрілкою* (рис. 1.48, а) позначає виклик процедури (операції) або передачу потоку управління. Повідомлення цього типу можуть бути використані активними об'єктами, коли один з них передає повідомлення цього типу і *очікує*, поки не закінчиться деяка послідовність дій, яка виконується іншим об'єктом. Зазвичай всі такі повідомлення синхронні, тобто ініціюються по завершенні діяльності або при виконанні певної умови.

2) *Суцільна лінія з V-подібною стрілкою* (рис. 1.48, б) означає асинхронне повідомлення в простому потоці управління. У цьому випадку клієнт передає асинхронне повідомлення і продовжує виконувати свою діяльність, *не чекаючи* відповіді від клієнта.

3) *Пунктирна лінія з V-подібною стрілкою* (рис. 1.48, в) означає повернення з виклику процедури. Стрілки цього типу часто відсутні на діаграмах кооперації, оскільки неявно передбачається їх існування після закінчення процесу виконання операції або діяльності.

Кожне повідомлення може бути позначено рядком тексту, який має наступний формат:

<Попередні повідомлення> <Вираз послідовності> <Значення, що повертається := ім'я повідомлення> <(Список аргументів)>

Попередні повідомлення – це розділені комами номери повідомлень, записані перед слеш-символом: **<Номер повідомлення ','> * <'/'>**. Якщо список номерів повідомлень порожній, то весь запис, включаючи слеш, опускається. Якщо номери повідомлень вказуються, то вони повинні відповідати номерам інших повідомлень на цій же діаграмі кооперації. Сенс зазначення попередніх повідомлень полягає в тому, що дане повідомлення не може бути передано, поки

не будуть передані своїм адресатам всі повідомлення, номери яких записані в даному списку. Знак «зірочка» вказує, що номерів може бути декілька, тоді вони розділяються комами.

Вираз послідовності – це розділений крапками список окремих термів послідовностей, після якого записується двокрапка:

<Терм послідовності'.'…> ':'

Кожен з термів послідовності – це окремий рівень процедурної вкладеності в формі завершеної ітерації. Найбільш верхній рівень відповідає крайньому лівому терму послідовності. Якщо всі потоки управління паралельні, то вкладеність відсутня. Кожен терм послідовності має наступний синтаксис:

[Ціле число | Ім’я] [Рекурентність].

Ціле число вказує на порядковий номер повідомлення в процедурній послідовності верхнього рівня. Повідомлення, номери яких відрізняються на одиницю, слідують поспіль один за одним.

Ім'я в формі літери алфавіту використовується для специфікації паралельних потоків (ниток) управління. Повідомлення, які відрізняються тільки ім'ям, є паралельними на цьому рівні вкладеності. На одному рівні вкладеності всі нитки управління еквівалентні в сенсі пріоритету передачі повідомлень.

Рекурентність використовується для зазначення ітеративного або умовного характеру виконання передачі повідомлень. Семантика рекурентності представляє нуль або більше повідомлень, які повинні бути виконані в залежності від записаної умови. Можливі два варіанти запису рекурентності:

– **'*' '[' Пропозиція-ітерація ']'** для запису ітеративного виконання відповідного виразу. Ітерація представляє послідовність повідомлень одного рівня вкладеності. Пропозиція-ітерація може бути опущена, якщо кількість ітерацій ніяк не специфікується. Найчастіше пропозиція-ітерація записується на псевдокоді або мовою програмування. У мові UML формат запису цієї пропозиції строгим чином не визначена;

– **'[' Пропозиція-умова ']'** для запису розгалуження. Ця форма запису специфікує умову для повідомлення, передача якого з даної гілки можлива тільки при його істинності.

У загальному випадку *пропозиція-умова* – звичайний логічний вираз і призначена для синхронізації окремих ниток потоку управління. Записується в

квадратних дужках і може бути опущена, якщо вона відсутня у даного повідомлення. Наявність цієї умови забезпечує передачу повідомлення тільки в тому випадку, якщо ця умова приймає значення «істина». Пропозиція-умова може бути записана звичайним текстом, на псевдокоді або деякій мові програмування.

Пропозиція-умова записується так само, як і ітерація, але без зірочки. Це можна розуміти як деяку однокрокову ітерацію. У загальному випадку передбачається, що специфікована ітерація виконується послідовно. Якщо необхідно відзначити можливість паралельного виконання ітерації, то для цього в мові UML використовується символ '*| |' Ітерація не поширюється на вкладені рівні даного потоку або нитки. Кожен рівень повинен мати власне уявлення для ітеративного повторення процедурної послідовності.

Ім'я повідомлення, записане в сигнатурі після значення, що повертається, означає ім'я події, яка ініціюється об'єктом-одержувачем повідомлення після його прийому. Найчастіше такою подією є виклик операції у об'єкта-одержувача. Це може бути реалізовано різними способами, один з яких – явне зазначення імені необхідної операції. Тоді відповідна операція повинна бути визначена в тому класі, якому належить об'єкт-одержувач.

Список аргументів – це розділені комами і загорнуті в круглі дужки дійсні параметри операції, виклик якої ініціюється даним повідомленням. Список аргументів може бути порожнім, однак дужки все одно записуються. Для запису аргументів також можна використовувати деякий псевдокод або мову програмування.

В UML передбачені стандартні дії, що виконуються у відповідь на отримання відповідного повідомлення. Вони можуть бути явно вказані на діаграмі кооперації у формі стереотипу перед ім'ям повідомлення, до якого вони належать, або вище його. У цьому випадку вони записуються в кутових дужках.

В UML передбачені наступні стереотипи повідомлень:

– **<< call >>** (викликати) – повідомлення, яке потребує виклику операції або процедури об'єкта-одержувача. Якщо повідомлення з цим стереотипом рефлексивне, то воно ініціює локальний виклик операції у того, хто надіслав це повідомлення;

– **<< return >>** (повернути) – повідомлення, яке повертає значення виконаної операції або процедури об'єкту, який її викликав. Значення результату може ініціювати розгалуження потоку управління;

– **<< create >>** (створити) – повідомлення, яке потребує створення іншого об'єкта для виконання певних дій. Створений об'єкт може стати активним (йому передається потік управління), а може залишитися пасивним;

– **<< destroy >>** (знищити) – повідомлення з явною вимогою знищити відповідний об'єкт. Посилається в тому випадку, коли необхідно припинити небажані дії з боку існуючого в системі об'єкта, або коли об'єкт більше не потрібен і повинен звільнити задіяні ним системні ресурси;

– **<< send >>** (послати) – означає надсилання сигналу іншому об'єкту, який асинхронно ініціюється одним об'єктом і приймається (перехоплюється) іншим. Відмінність сигналу від повідомлення полягає в тому, що сигнал повинен бути явно описаний в тому класі, об'єкт якого ініціює його передачу.

Рекомендації з побудови діаграм кооперації

Побудова діаграми кооперації можна починати відразу після побудови діаграми класів. При розробці діаграм кооперації спочатку зображуються об'єкти і зв'язки між ними. Далі на діаграму кооперації необхідно нанести всі повідомлення, вказавши їх порядок і інші семантичні особливості. Діаграма кооперації може містити тільки ті об'єкти і зв'язки, які вже визначені на побудованій раніше діаграмі класів. В іншому випадку, якщо виникає необхідність включення в діаграму кооперації об'єктів, які створюються на основі відсутніх класів, то діаграма класів повинна бути модифікована за допомогою включення в неї явного опису цих класів.

Процес побудови діаграми кооперації повинен бути узгоджений з процесами побудови діаграми класів і діаграми послідовності. У першому випадку, як уже зазначалося, необхідно стежити за використанням тільки тих об'єктів, для яких визначені їх класи. У другому випадку необхідно погоджувати послідовності передавання повідомлень. Йдеться про те, що не допускається різний порядок проходження повідомлень для моделювання однієї і тієї ж взаємодії на діаграмі кооперації і діаграмі послідовності.

Необхідно пам'ятати, що діаграма кооперації, з одного боку, забезпечує концептуально узгоджений перехід від статичної моделі діаграми класів до динамічних моделей поведінки, яку представляють діаграмами послідовності, станів і діяльності. З іншого боку, діаграма цього типу зумовлює особливості реалізації моделі на діаграмах компонентів і розгортання.

1.2.6. Діаграми послідовності UML

Діаграма послідовності (sequence diagram) – діаграма, на якій показана взаємодія об'єктів, впорядкована за часом їхньої прояву.

Особливості взаємодії елементів системи, що моделюється, можуть бути представлені на діаграмах кооперації і послідовності. Діаграми кооперації використовуються для специфікації динаміки поведінки систем, хоча час в явному вигляді в них відсутній. Однак часовий аспект поведінки може мати суттєве значення при моделюванні синхронних процесів, що описують взаємодію об'єктів. Саме для цієї мети в мові UML використовуються діаграми послідовності.

На діаграмі послідовності явно присутня вісь (шкала) часу, що дозволяє візуалізувати часові відносини між переданими повідомленнями. За допомогою діаграми послідовності можна представити взаємодію елементів моделі, як своєрідний часовий графік «життя» всієї сукупності об'єктів, пов'язаних між собою для реалізації варіанту використання програмної системи, досягнення бізнес-мети або виконання будь-якої задачі.

Об'єкти і їх зображення на діаграмі послідовності

На діаграмі послідовності також зображуються об'єкти, які безпосередньо беруть участь у взаємодії, при цьому ніякі статичні зв'язку з іншими об'єктами не візуалізуються. Для діаграми послідовності ключовим моментом є саме динаміка взаємодії об'єктів в часі. При цьому діаграма послідовності має як би два виміри. Один – зліва направо у вигляді вертикальних ліній, кожна з яких зображує лінію життя окремого об'єкта, який бере участь у взаємодії. Другий вимір діаграми послідовності – вертикальна часова вісь, спрямована зверху вниз.

Кожен об'єкт графічно зображується у формі прямокутника і розташовується у верхній частині своєї лі*нії життя* (рис. 1.49). Всередині прямокутника записуються власне ім'я об'єкта (*з малої літери*) і ім'я класу, розділені двокрапкою. При цьому весь запис підкреслюється, що є ознакою об'єкта, який, як зазначалося раніше, представляє собою екземпляр класу.

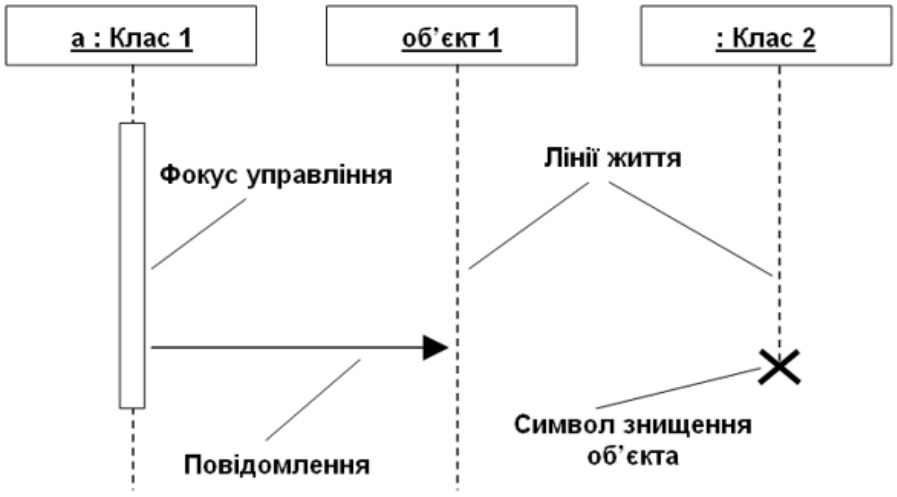


Рисунок 1.49 – Графічні елементи діаграми послідовності

Для об'єктів діаграми послідовності залишаються справедливими правила іменування, розглянуті раніше стосовно до діаграм кооперації. Якщо на діаграмі послідовності відсутнє власне ім'я об'єкта, то має бути зазначено ім'я класу. Такий об'єкт вважається ***анонімним***. Може бути відсутнім і ім'я класу, але при цьому має бути зазначено власне ім'я об'єкта. Такий об'єкт вважається ***сиротою***. Роль класів в іменах об'єктів на діаграмах послідовності, як правило, не вказується.

Крайнім зліва на діаграмі зображується об'єкт – ініціатор модельованого процесу взаємодії (об'єкт '**а**' на рис. 1.49). Праворуч – інший об'єкт, який безпосередньо взаємодіє з першим. Таким чином, порядок розташування об'єктів на діаграмі послідовності визначається виключно міркуваннями зручності візуалізації їх взаємодії.

Початковому моменту часу відповідає верхня частина діаграми. При цьому процес взаємодії об'єктів реалізується за допомогою повідомлень, які надсилаються одними об'єктами іншим. Повідомлення зображуються у вигляді горизонтальних стрілок з ім'ям повідомлення і утворюють певний порядок щодо часу своєї ініціалізації. Іншими словами, повідомлення, розташовані на діаграмі послідовності вище, передаються раніше тих, які розташовані нижче. Масштаб на осі часу не вказується, оскільки діаграма послідовності моделює лише часову упорядкованість взаємодій типу «раніше-пізніше».

Лінія життя об'єкту (object lifeline) – це вертикальна лінія на діаграмі послідовності, яка представляє певний період часу, протягом якого існує об'єкт.

Лінія життя об'єкта зображується пунктирною вертикальною лінією, асоційованою з єдиним об'єктом на діаграмі послідовності. Лінія життя призначена для позначення періоду часу, протягом якого об'єкт існує в системі і, отже, може потенційно брати участь у всіх її взаємодіях. Якщо об'єкт існує в системі постійно, то і його лінія життя повинна тривати по всій робочій області діаграми послідовності від самої верхньої її частини до найнижчої ('**об'єкт 1**' і анонімний об'єкт '**:Клас 2**' на рис. 1.49).

Окремі об'єкти, закінчивши виконання своїх операцій, можуть бути знищені, щоб звільнити займані ними ресурси. Для таких об'єктів лінія життя обривається в момент знищення. Для позначення часу знищення об'єкта в мові UML застосовується спеціальний символ у формі латинської букви «X». На рис. 1.50 цей символ використовується для знищення анонімного об'єкта, утвореного від **Класу 3**. Нижче цього символу пунктирна лінія не зображується, оскільки відповідного об'єкта в системі вже немає, і цей об'єкт повинен бути виключений з усіх наступних взаємодій.

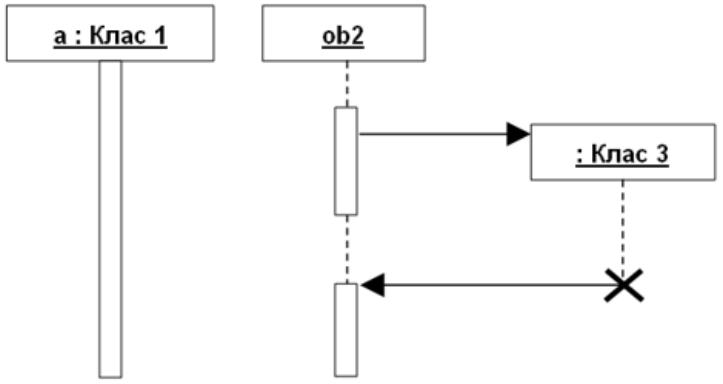


Рисунок 1.50 – Графічне зображення лінії життя і фокусів управління на діаграмі послідовності

Зовсім не обов'язково створювати всі об'єкти в початковий момент часу. Окремі об'єкти в системі можуть створюватися при необхідності, істотно економлячи ресурси системи і підвищуючи її продуктивність. В цьому випадку прямокутник такого об'єкта зображується не в верхній частині діаграми послідовності, а в тій, яка відповідає моменту створення об'єкта (анонімний об'єкт, утворений від **Класу 3** на рис. 1.50). При цьому прямокутник об'єкта вертикально розташовується в тому місці діаграми, яке по осі часу збігається з

моментом його виникнення в системі. Об'єкт створюється зі своєю лінією життя а, можливо, і з фокусом управління.

В процесі функціонування об'єктно-орієнтованих систем одні об'єкти можуть перебувати в активному стані, безпосередньо виконуючи певні дії, або в стані пасивного очікування повідомлень від інших об'єктів. ***Фокус управління*** – символ, застосовуваний для того, щоб явно виділити подібну активність об'єктів на діаграмах послідовності.

Фокус управління (focus of control) – спеціальний символ на діаграмі послідовності, який вказує період часу, протягом якого об'єкт виконує деяку дію, перебуваючи в активному стані.

Фокус управління зображується у формі витягнутого вузького прямокутника (об'єкт '**a**' на рис. 1.49), верхня сторона якого позначає початок отримання фокусу управління об'єктом (початок активності), а його нижня сторона – закінчення фокусу управління (закінчення активності). Цей прямокутник розташовується нижче позначення відповідного об'єкта і може заміняти його лінію життя (об'єкт '**a**' на рис. 1.50), якщо на всій її довжині він активний.

Періоди активності об'єкта можуть чергуватися з періодами його пасивності або очікування. В цьому випадку у такого об'єкта фокуси управління змінюють своє зображення на лінію життя і навпаки (об'єкт сирота '**ob2**' на рис. 1.50). Важливо розуміти, що отримати фокус управління може тільки об'єкт, у якого в цей момент є лінія життя. Якщо ж об'єкт був знищений, то знову виникнути в системі він вже не може. Замість нього може бути створений лише екземпляр цього ж класу, який, строго кажучи, буде іншим об'єктом.

В окремих випадках ініціатором взаємодії в системі може бути актор або зовнішній користувач. При цьому актор зображується на діаграмі послідовності найпершим об'єктом зліва зі своїм фокусом управління (рис. 1.51). Найчастіше актор і його фокус управління існуватимуть в системі постійно, відзначаючи характерну для користувача активність в ініціюванні взаємодій з системою. Актор може мати власне ім'я або залишатися анонімним.

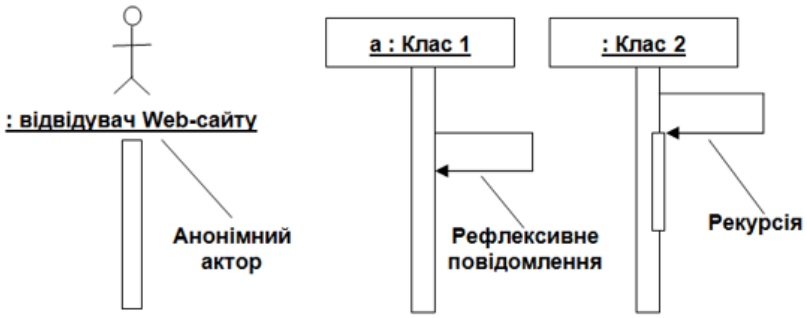


Рисунок 1.51 – Графічне зображення актора, рефлексивного повідомлення і рекурсії на діаграмі послідовності

В окремих випадках об'єкт може послати повідомлення самому собі, ініціюючи так звані ***рефлексивні*** повідомлення. Для цього використовується спеціальне зображення (повідомлення у об'єкта '**a**' на рис. 1.51). Такі повідомлення зображуються у формі повідомлення, початок і кінець якого стикаються з лінією життя або фокусом управління одного і того ж об'єкта. Подібні ситуації виникають, наприклад, при обробці натискань на клавіші клавіатури при введенні тексту в редагований документ, при наборі цифр номера телефону абонента.

Якщо в результаті рефлексивного повідомлення створюється новий підпроцес або нитка управління, то говорять про ***рекурсивний*** або *вкладений* фокус управління. На діаграмі послідовності рекурсія позначається маленьким прямокутником, приєднаним до правого боку фокусу управління того об'єкта, для якого зображується дана рекурсивна взаємодія (анонімний об'єкт **Класу 2** на рис. 1.51).

Повідомлення на діаграмі послідовності

Повідомлення, як елементи мови UML, вже розглядалися раніше при вивченні діаграми кооперації. Стрілки повідомлень зображуються аналогічно розглянутим раніше, але стосовно діаграм послідовності повідомлення мають додаткові семантичні особливості. При цьому на діаграмі послідовності всі повідомлення впорядковані за часом передавання, хоча номери у них можуть не вказуватися.

Повідомлення на діаграмах послідовності, мають певне графічне зображення (рис. 1.52).

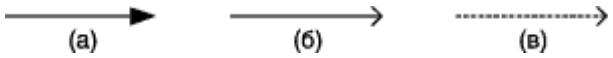


Рисунок 1.52 – Графічне зображення різних видів повідомлень між об’єктами на діаграмі послідовності

Перший різновид повідомлення (рис. 1.52, а) найбільш поширений і використовується для виклику процедур, виконання операцій або позначення окремих вкладених потоків управління. Початок цієї стрілки, як правило, стикається з фокусом управління того об'єкта-клієнта, який ініціює це повідомлення. Кінець стрілки стикається з лінією життя того об'єкту, який приймає це повідомлення і виконує у відповідь певні дії. При цьому приймаючий об'єкт може отримати фокус управління, стаючи в цьому випадку активним. Передаючий об'єкт може втратити фокус управління або залишитися активним.

Другий різновид повідомлення (рис. 1.52, б) використовується для позначення простого *асинхронного повідомлення*, яке передається в довільний момент часу. Передача такого повідомлення зазвичай не супроводжується отриманням фокуса управління об'єктом-одержувачем.

Третій різновид повідомлення (рис. 1.52, в) використовується для повернення з виклику процедури. Прикладом може служити просте повідомлення про завершення обчислень без надання результату розрахунків об'єкту-клієнта. У процедурних потоках управління ця стрілка може бути опущена, оскільки її наявність неявно передбачається в кінці активізації об'єкта. У той же час вважається, що кожен виклик процедури має свою пару – повернення виклику. Для непроцедурних потоків управління, включаючи паралельні і асинхронні повідомлення, стрілка повернення повинна вказуватися явно.

Зазвичай повідомлення зображуються горизонтальними стрілками, що з'єднують лінії життя або фокуси управління двох об'єктів на діаграмі послідовності. При цьому неявно передбачається, що час передавання повідомлення достатньо малий в порівнянні з процесами виконання дій об'єктами. Вважається також, що за час передачі повідомлення з відповідними об'єктами не може статися жодних подій. Іншими словами, стани об'єктів не змінюються. Якщо ж це припущення не може бути визнано справедливим, то стрілка повідомлення зображується під нахилом, так щоб кінець стрілки розташовувався нижче її початку.

Кожне повідомлення на діаграмі послідовності асоціюється з певною операцією, яка повинна бути виконана об'єктом, що прийняв його. При цьому операція може мати аргументи або параметри, значення яких впливають на отримання результатів. Відповідні параметри матиме і повідомлення, що викликає цю дію. Більш того, значення параметрів окремих повідомлень можуть містити умовні вирази, утворюючи розгалуження або альтернативні шляхи основного потоку управління.

Розгалуження потоку управління

Одна з особливостей діаграми послідовності – можливість візуалізувати просте *розгалуження* процесу. Для зображення розгалуження використовуються дві або більше стрілки, що виходять з однієї точки фокусу управління об'єкта (об'єкт '**ob1**' на рис. 1.53). При цьому поряд з кожною з них має бути явно вказана відповідна умова гілки в формі логічного виразу.

Кількість гілок може бути довільною, однак наявність *розгалужень* може істотно ускладнити інтерпретацію діаграми послідовності. Пропозиція-умова має бути явно вказана для кожної гілки і записується в формі звичайного тексту, псевдокоду або виразу мови програмування. Цей вираз завжди має повертати деяке логічне значення. Запис цих умов повинна виключати одночасну передачу альтернативних повідомлень за двома і більше гілок. В іншому випадку на діаграмі послідовності може виникнути конфлікт розгалуження.

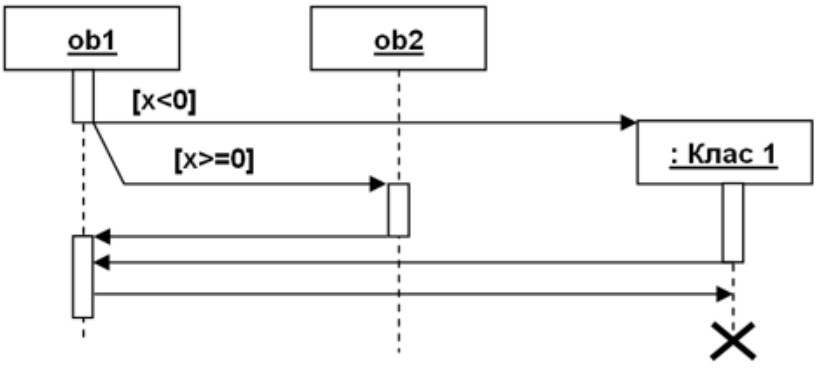


Рисунок 1.53 – Графічне зображення бінарного розгалуження потоку управління на діаграмі послідовності

За допомогою розгалуження можна зобразити і більш складну логіку взаємодії об'єктів (об'єкт '**ob1**' на рис. 1.53). Якщо умов більше двох, то для

кожного з них необхідно передбачити ситуацію єдиного виконання. Описаний нижче приклад відноситься до моделювання взаємодії програмної системи обслуговування клієнтів в банку. У цьому прикладі діаграми послідовності об'єкт '**ob1**' викликає виконання дій у одного з трьох інших об'єктів.

Умовою розгалуження може слугувати сума коштів, які знімаються клієнтом зі свого поточного рахунку. Якщо ця сума перевищує 1500$, то можуть знадобитися додаткові дії, пов'язані зі створенням і подальшим руйнуванням об'єкта **«:Клас 1»**. Якщо ж сума перевищує 100 $, але не перевищує 1500 $, то викликається операція або процедура об'єкта '**ob3**'. І, нарешті, якщо сума не перевищує 100 $, то викликається операція або процедура об'єкта '**ob2**'. При цьому об'єкти '**ob1**', '**ob2**' і '**ob3**' постійно існують в системі. Останній об'єкт **«:Клас 1»** створюється тільки в тому випадку, якщо справедлива перша з альтернативних умов. В іншому випадку він може бути ніколи не створений.

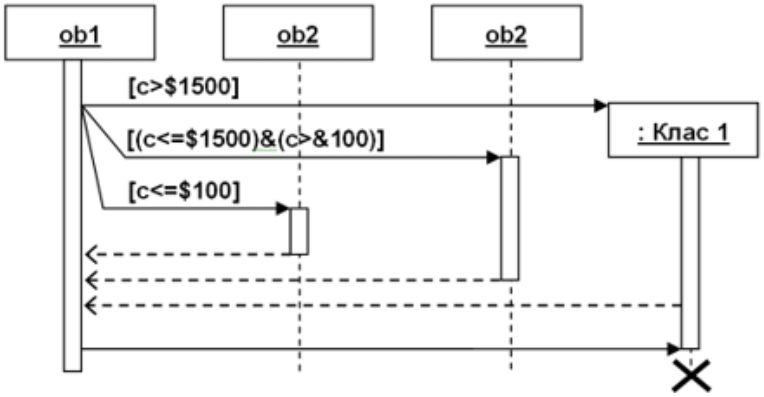


Рисунок 1.54 – Графічне зображення тернарного розгалуження потоку управління на діаграмі послідовності

Об'єкт '**ob1**' має постійний фокус управління, а всі інші об'єкти – отримують фокус управління тільки для виконання ними відповідних операцій.

На діаграмах послідовності для повідомлень також можуть використовуватися стереотипи, які розглянуті раніше для діаграми кооперації. Їх семантика і синтаксис залишаються без зміни так, як вони визначені в нотації мови UML. Нижче подано діаграму послідовності для описаного вище випадку розгалуження, яку доповнено стереотипними значеннями окремих повідомлень (рис. 1.55). Очевидно, ця діаграма послідовності є більш виразною і простою для змістовної інтерпретації.

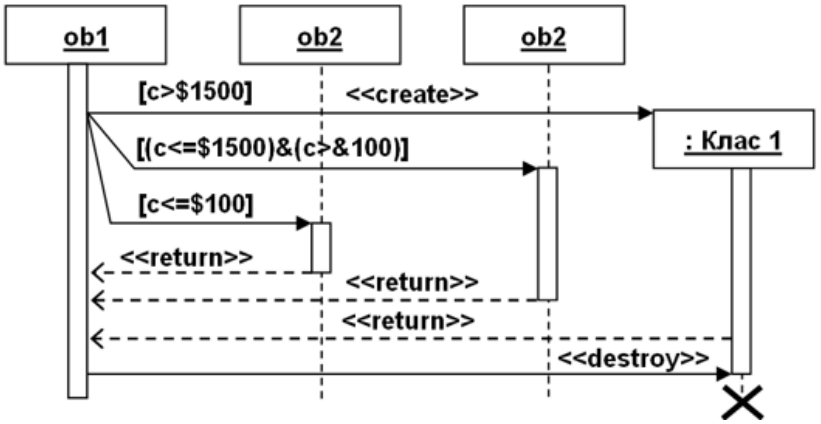


Рисунок 1.55 – Діаграма послідовності із стереотипними значеннями повідомлень

Як вже зазначалося раніше, повідомлення можуть мати власне ім'я, в якості якого виступає ім'я операції, виклик якої ініціюють ці повідомлення у приймаючого об'єкта. У цьому випадку поряд зі стрілкою записується ім'я операції з круглими дужками, в яких можуть зазначатися параметри або аргументи відповідної операції. Якщо вони відсутні, то дужки після імені операції все одно повинні бути зображені.

Рекомендації з побудови діаграм послідовності

Побудову діаграми послідовності доцільно починати з виділення з усієї сукупності класів лише тих, об'єкти яких беруть участь в модельованій взаємодії. Після цього всі об'єкти наносяться на діаграму, з дотриманням порядку ініціалізації повідомлень. Тут необхідно встановити, які об'єкти будуть існувати постійно, а які тимчасово - тільки на період виконання ними необхідних дій.

Коли об'єкти візуалізовані, можна приступати до специфікації повідомлень. При цьому необхідно враховувати ті операції, які мають класи відповідних об'єктів в моделі системи. При необхідності уточнення цих операцій слід використовувати їх стереотипи. Для знищення об'єктів, які створюються на час виконання своїх дій, потрібно передбачити явне повідомлення. Найбільш прості випадки розгалуження процесу взаємодії можна зобразити на одній діаграмі з використанням відповідних графічних примітивів. У більш складних випадках для моделювання кожної гілки управління може знадобитися окрема діаграма послідовності. Слід пам'ятати, що кожен альтернативний потік управління ускладнює розуміння побудованої моделі.

Загальним правилом є візуалізація особливостей реалізації кожного варіанту використання на окремій діаграмі послідовності. У цій ситуації окремі діаграми повинні розглядатися спільно як одна модель взаємодії. Необхідність синхронізації складних потоків управління, як правило, вимагають введення в модель додаткових обмежень.

1.2.7. Діаграми діяльності UML

При моделюванні поведінки проектованої або аналізуємої програмної системи виникає необхідність не тільки представити процес зміни її стану, але і деталізувати особливості ***алгоритмічної*** та ***процедурної*** реалізації виконуваних системою операцій. Для цієї цілі, як правило, використовуються блок-схеми або структурні схеми алгоритмів. Кожна така схема акцентує увагу на послідовності виконання заданих процедур або елементарних операцій, які в сукупності призводять до появі бажаного результату.

Зі збільшенням складності системи суворе дотримання певної послідовності виконуваних дій набуває великого значення. Порушення серії операцій при монтажі двигуна призводить до його пошкодження або виходу з ладу. Ще більш катастрофічні наслідки можуть мати місце у разі відхилення від встановленої послідовності дій при підйомі або посадці авіалайнера, запуску ракет, регламентних робіт на атомних електростанціях.

Для моделювання процесу виконання операцій у мові UML використовуються *діаграми діяльності*. Застосовувана в них графічна нотація багато в чому схожа на нотацію діаграми станів, оскільки на діаграмах діяльності також присутні позначення станів і переходів. Відмінність полягає в семантиці станів, які використовуються для подання *діяльності* і *дій*, а також у відсутності на переходах сигнатури подій. Кожне стан на діаграмі діяльності відповідає виконанню деякої операції, а перехід в наступний стан відбувається тільки після завершення виконання цієї операції. Діаграма діяльності подається у формі графу діяльності, вершинами якого є *стани дії* або *діяльності*, а дугами – *переходи* від одного стану дії до іншого.

Діаграми діяльності – частковий випадок діаграм станів. Вони дозволяють реалізувати в мові UML особливості процедурного і синхронного управління, обумовленого завершенням внутрішніх дій і діяльності. Основним напрямком

використання діаграм діяльності є візуалізація особливостей реалізації операцій класів, коли необхідно представити *алгоритми* їх виконання. При цьому кожен стан може представляти виконання операції певного класу або її частини, дозволяючи використовувати діаграми діяльності для опису реакцій на внутрішні події системи.

В контексті мови UML *діяльність* являє собою сукупність окремих обчислень, виконуваних автоматом. При цьому окремі елементарні обчислення можуть приводити до результату або *дії*. На діаграмі діяльності відображається логіка або послідовність переходу від однієї діяльності до іншої, при цьому увагу фіксується на результаті діяльності. Сам же результат може призвести до зміни стану системи або поверненню деякого значення. Діаграма діяльності призначена для моделювання поведінки систем, хоча час в явному вигляді відсутній на цій діаграмі.

Стани діяльності і дії

Стан діяльності (activity state) – стан у графі діяльності, який слугує для подання процедурної послідовності дій, що вимагають певного часу.

Перехід зі стану діяльності відбувається після виконання специфікованої в ньому послідовності дій. Стан діяльності не може мати внутрішніх переходів, оскільки він є елементарним. Діяльність, описана в стані діяльності, не може бути перервана ніякими зовнішніми подіями. Звичайне використання стану діяльності полягає в моделюванні підпроцесу виконання окремих алгоритмів або процедур.

Стан дії (action state) – спеціальний випадок стану з деякою вхідною дією і, щонайменше, з одним переходом, що виходить зі стану.

Перехід зі стану дії відбувається після завершення вхідної дії. Стан дії не може мати внутрішніх переходів, оскільки він є елементарним. Звичайне використання стану дії полягає в моделюванні кроку виконання *алгоритму* або *процедури* в рамках одного потоку управління.

Графічно стани діяльності і дії зображуються однаковою фігурою, що нагадує прямокутник, бічні сторони якого замінені випуклими дугами (рис. 1.56). Всередині цієї фігури записується ім'я стану діяльності (рис. 1.56, а) або дії (рис. 1.56, б) у формі *виразу* (expression), якій повинен бути унікальним в межах однієї діаграми діяльності.

Ім’я діяльності або дії може бути записано природною мовою, на псевдокоді або мовою програмування. Ніяких додаткових або неявних обмежень при запису імені не накладається. Рекомендується в якості імені простої діяльності використовувати дієслово з пояснювальними словами (рис. 1.56, а). Якщо *дія* може бути представлена у формальному вигляді, то доцільно записати її тією мовою програмування, якою передбачається реалізовувати проект, що розробляється (рис. 1.56, б).

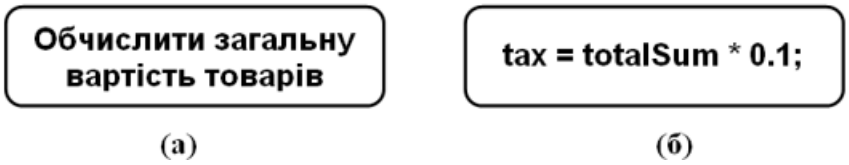


Рисунок 1.56 – Графічне зображення станів діяльності і дії на діаграмі діяльності

Іноді виникає необхідність представити на діаграмі діяльності складну діяльність, яка складається з декількох більш простих. В цьому випадку можна використовувати спеціальне позначення так званого стану ***піддіяльності***.

Стан піддіяльності (subactivity state) – стан на діаграмі діяльності, який слугує для представлення неатомарної послідовності кроків процесу.

Цей стан позначається спеціальною піктограмою в правому нижньому куту символу стану діяльності (рис. 1.57). Дана конструкція може застосовуватися до будь-якого елементу мови UML, який підтримує «вкладеність» своєї структури. При цьому піктограма може бути додатково позначена типом вкладеної структури.

Кожна діаграма діяльності повинна мати один *початковий* і *кінцевий* стани. Вони мають такі ж позначення, як і на діаграмі станів. При цьому кожна діяльність починається в початковому стані і закінчується в кінцевому стані. Саму діаграму діяльності прийнято розташовувати таким чином, щоб дії слідували зверху вниз або зліва направо. В цьому випадку початковий стан буде зображуватися у верхній або лівій частині діаграми, а кінцевий – в її нижній або правій частині. В інтересах зручності візуального представлення на діаграмі діяльності допускається зображати кілька кінцевих станів. В цьому випадку всі їх прийнято вважати еквівалентними один одному.

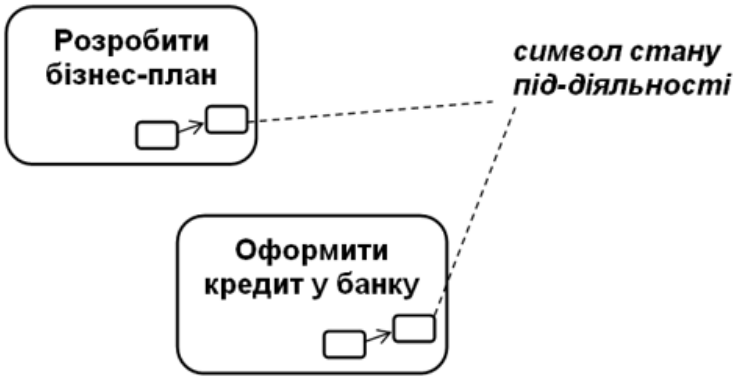


Рисунок 1.57 – Графічне зображення стану піддіяльності

Початковий стан (start state) – різновид стану, що позначає початок виконання процесу зміни станів модельованої системи.

У цьому стані система знаходиться в початковий момент часу. Він слугує для позначення на діаграмі діяльності графічної області, від якої починається процес зміни станів. Графічно початковий стан в мові UML позначається у вигляді зафарбованого кола (рис. 1.58, а), з якого стрілка-перехід може тільки виходити.

Рисунок 1.58 – Графічне зображення початкового і кінцевого станів на діаграмі діяльності

Кінцевий стан (final state) – різновид стану, що позначає припинення процесу зміни станів системи, що моделюється.

У цьому стані повинен перебувати модельований об'єкт або система після завершення роботи. Воно слугує для позначення на діаграмі діяльності графічної області, в якій завершується процес зміни станів або життєвий цикл даної системи або об’єкта. Графічно кінцевий стан в мові UML позначається у вигляді зафарбованого кола, яке міститься у незафарбованому колі (рис. 1.58, б). У кінцевий стан стрілка-перехід може тільки входити.

Переходи **на діаграмі діяльності**

Перехід (transition) – відношення між двома станами, яке вказує на те, що об'єкт в першому стані повинен виконати певні дії і перейти в другий стан.

Перехід здійснюється при настанні деякої події: закінчення виконання діяльності, отримання об'єктом повідомлення або приймання сигналу.

Перехід може бути спрямований в той же стан, з якого він виходить. В цьому випадку його називають ***переходом в себе***. Початковий і цільовий стани переходу в себе збігаються. Цей перехід зображується петлею зі стрілкою. При переході в себе об'єкт залишає початковий стан, а потім знову входить в нього. При цьому щоразу виконуються внутрішні дії.

При побудові діаграми діяльності використовуються *переходи*, які відбуваються відразу після завершення діяльності або виконання відповідної дії. Такий перехід передає управління в подальший стан відразу, як тільки закінчиться дія або діяльність в попередньому стані. На діаграмі такий перехід зображується суцільною лінією зі стрілкою.

Якщо зі стану дії виходить єдиний перехід, то його можна ніяк не позначати. Якщо ж таких переходів декілька, то при моделюванні послідовності діяльностей запускається тільки одна з них. В цьому випадку для кожного з таких переходів має бути явно записана власна *сторожова умова* в прямих дужках.

Сторожова умова (guard condition) – записана в прямих дужках логічна умова, яка представляє собою булевський вираз.

При цьому для всіх переходів, що виходять з деякого стану діяльності, має виконуватися вимога істинності тільки одного з них. Подібний випадок зустрічається тоді, коли послідовність діяльностей повинна розділитися на альтернативні гілки в залежності від значення проміжного результату. Така ситуація отримала назву ***розгалуження***, а для її позначення застосовується спеціальний символ *рішення*.

Графічно розгалуження на діаграмі діяльності позначається символом *рішення* (decision), зображуваного в формі невеликого ромба, всередині якого немає ніякого тексту (рис. 1.59). В цей ромб може входити тільки одна стрілка від того стану дії, після виконання якого потік управління повинен бути продовжений за однією з взаємно виключних гілок. Прийнято вхідну стрілку приєднувати до верхньої або лівої вершині символу рішення. Вихідних стрілок може бути дві або більше, але для кожної з них явно вказується відповідна сторожова умова в формі булевського виразу.

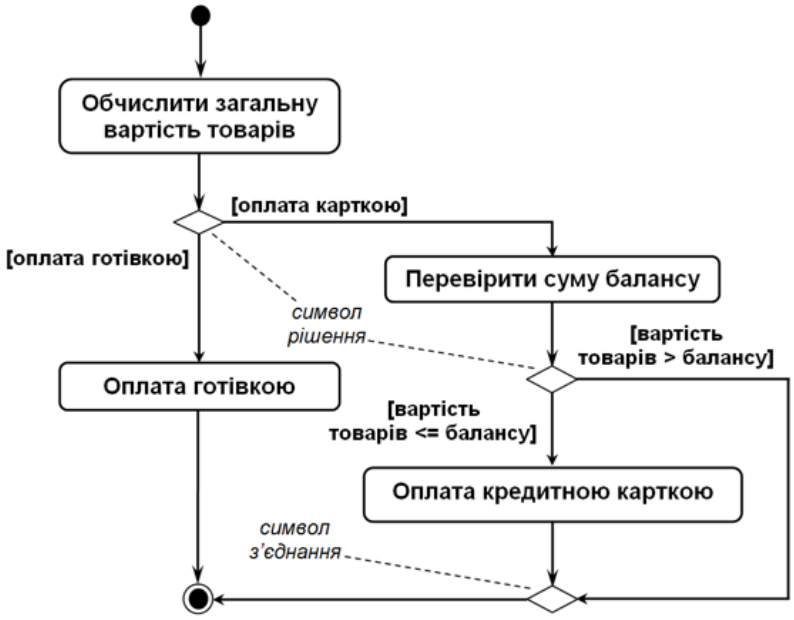


Рисунок 1.59 – Графічне зображення розгалужень на діаграмі діяльності

Для графічного об'єднання альтернативних гілок на діаграмі діяльності рекомендується також використовувати аналогічний символ у формі ромба, який в цьому випадку називається ***з'єднанням*** (merge). Цей символ, всередині якого теж не записується ніякого тексту, спрощує візуальний контроль логіки виконання процедурних дій на діаграмі діяльності (рис. 1.59 внизу). Стрілок, що входять у символ з'єднання, може бути декілька, вони виходять від станів дії, що належать до однієї з взаємно виключних гілок. Виходити з ромба з'єднання може тільки одна стрілка, при цьому, ні вхідні, ні вихідні стрілки не повинні містити сторожових умов. Винятком є ситуація, коли з метою скорочення діаграми об'єднуються символ рішення з символом з'єднання. Порушення цих правил робить діаграму діяльності *неправильною* (ill formed).

Діаграма діяльності (рис. 1.59) моделює ситуацію, що виникає при оплаті товарів. Як правило, заплатити за покупки можна або готівкою, або по кредитній картці. Якщо покупцем обрано варіант оплати за кредитною карткою, то перевіряється сума балансу пред'явленої до оплати кредитної картки. При цьому оплата відбувається тільки в тому випадку, якщо загальна вартість придбаних товарів не перевищує суми балансу цієї картки. В іншому випадку оплати не відбувається.

Один з найбільш значущих недоліків звичайних блок-схем або структурних схем алгоритмів пов'язаний з проблемою зображення паралельних гілок окремих

обчислень. Оскільки розпаралелювання обчислень істотно підвищує загальну швидкодію програмних систем, необхідно мати графічні примітиви для подання паралельних процесів. У діаграмах діяльності з цією метою використовується спеціальний символ – пряма риска, якою позначаються *розділення* і *злиття* паралельних обчислень або потоків управління.

На діаграмах діяльності така риска зображується відрізком горизонтальної, рідше – вертикальної, лінії, товщина якої трохи ширше ліній простих переходів діаграми діяльності. При цьому *розділення* (fork) має один вхідний перехід і декілька вихідних (рис. 1.60, а), які зображуються відрізками вертикальних, рідше – горизонтальних, ліній. *Злиття* (join), навпаки, має декілька вхідних переходів і один вихідний (рис. 1.60, б).

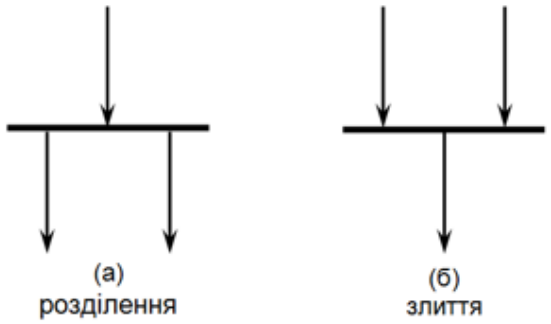


Рисунок 1.60 – Графічне зображення розділення та злиття паралельних потоків на діаграмі діяльності

Розглянутих переходів виявляється достатньо для моделювання різних за складністю ситуацій. Для ілюстрації особливості зображення розгалуження і паралельних діяльностей можна розглянути приклад реєстрації пасажирів в аеропорту (рис. 1.61). Спочатку виконується діяльність з перевірки квитка. У разі якщо квиток недійсний, він повертається пасажиру, при цьому ніяких додаткових дій не виконується.

Якщо ж білет дійсний, то пасажиру видається посадковий талон. На додаток до цього перевіряється громадянство і наявність багажу у пасажира. Якщо є багаж, то його перевірка може бути виконана паралельно, за результатами якої пасажиру видається талон на багаж. Якщо пасажир є іноземним громадянином, то додатково перевіряється наявність у нього візи. Якщо віза дійсна, то перевірка завершується успішно, і пасажир з повернутим йому квитком може пройти на посадку.

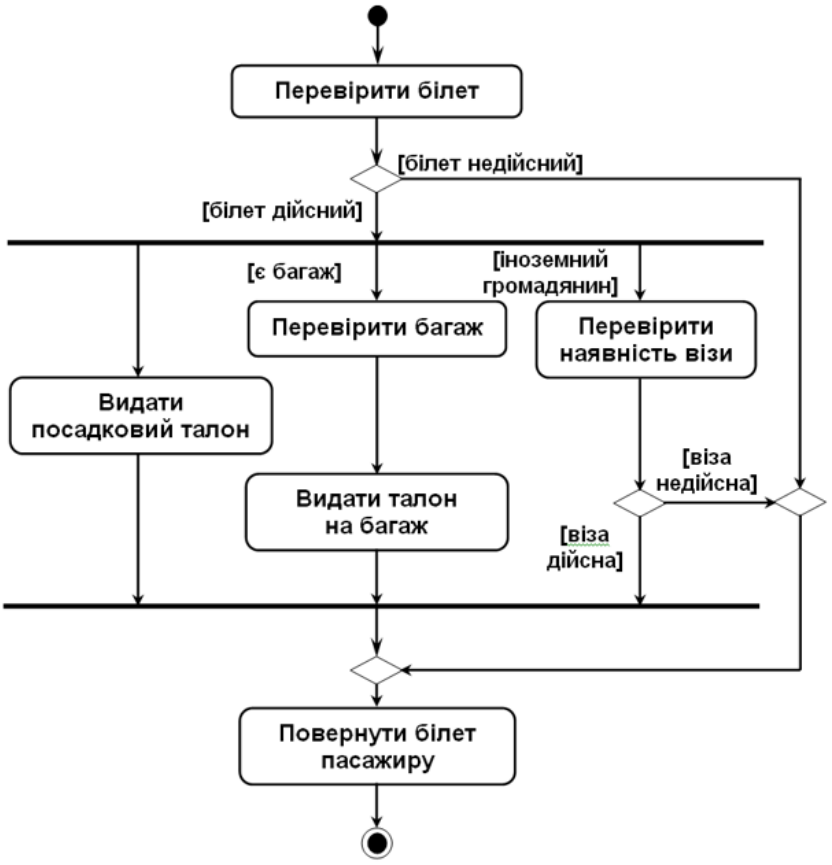


Рисунок 1.61 – Діаграма діяльності для прикладу реєстрації пасажирів в аеропорту

Якщо ж віза виявиться недійсною, то для цього пасажира посадка на даний рейс виявляється неможливою. У цьому випадку йому не видається посадковий талон і талон на багаж, в разі його наявності, оскільки відбувається припинення всіх виконуваних співробітниками аеропорту дій.

Доріжки

Діаграми діяльності можуть бути використані не тільки для специфікації алгоритмів обчислень або потоків управління в програмних системах. Не менш важлива область їх застосування пов'язана з моделюванням бізнес-процесів. У цьому контексті діяльність будь-якої компанії або фірми є не що інше, як сукупність окремих дій, робіт, операцій, спрямованих на досягнення необхідного результату.

Однак стосовно до бізнес-процесів бажано виконання кожної дії асоціювати з конкретним підрозділом компанії. В цьому випадку підрозділ буде нести відповідальність за реалізацію певних дій, а сам бізнес-процес представляється у вигляді переходів дій з одного підрозділу до іншого. Для моделювання цих особливостей в мові UML запропонована спеціальна конструкція, яка отримала назву *доріжки*.

Доріжка (swimlane) – графічна область діаграми діяльності, що містить елементи моделі, відповідальність за виконання яких належить окремим підсистемам.

В даному випадку мається на увазі візуальна аналогія з плавальними доріжками в басейні, якщо дивитися на відповідну діаграму діяльності зверху. При цьому всі стани на діаграмі діяльності поділяються на групи, розмежовані вертикальними лініями. Дві сусідні лінії утворюють доріжку, а група станів між цими лініями виконується організаційним підрозділом (відділом, групою, відділенням, філією) або співробітником компанії (рис. 1.62). В останньому випадку прийнято вказувати посаду співробітника, відповідального за виконання певних дій.

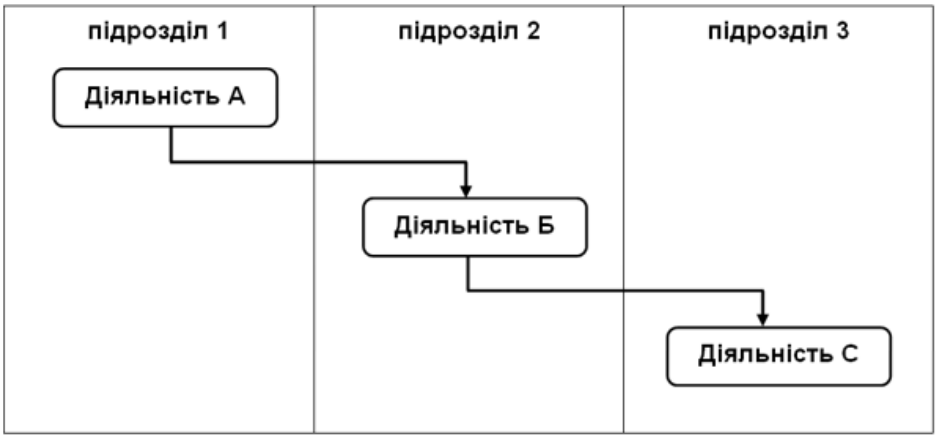


Рисунок 1.62 – Варіант діаграми діяльності з доріжками

Назви підрозділів або посад явно вказуються у верхній частині доріжки. Перетинати лінію доріжки можуть тільки переходи, які в цьому випадку позначають вихід або вхід потоку управління до відповідного підрозділу компанії. Порядок проходження доріжок не несе будь-якої семантичної інформації і визначається міркуваннями зручності.

Як приклад можна розглянути фрагмент діаграми діяльності торгової компанії, яка обслуговує замовлення клієнтів. Підрозділами компанії зазвичай є відділ прийому і оформлення замовлень, відділ продажів і склад. Цим підрозділам відповідатимуть три доріжки на діаграмі діяльності, кожна з яких специфікує зону відповідальності підрозділу. У цьому випадку діаграма діяльності містить в собі не тільки інформацію про послідовність виконання робочих дій, але і про те, який підрозділ торгової компанії має виконувати ту чи іншу дію (рис. 1.63).

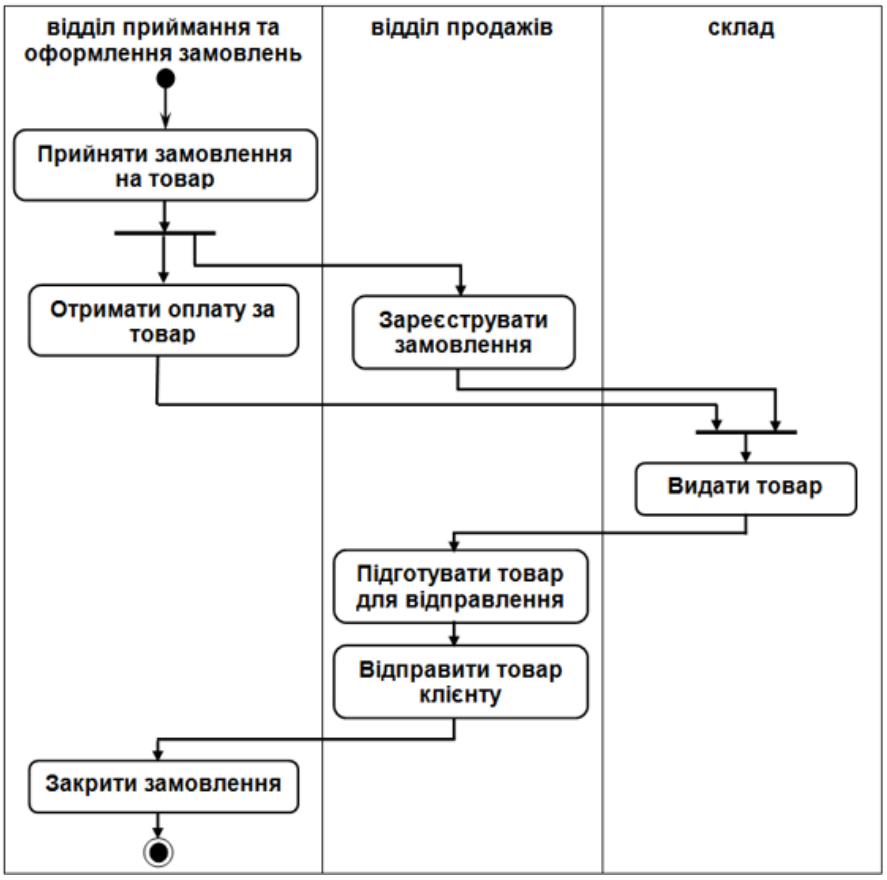


Рисунок 1.63 – Фрагмент діаграми діяльності для торгової компанії

Із зазначеної діаграми діяльності видно, що після прийняття замовлення від клієнта відділом приймання і оформлення замовлень здійснюється розпаралелювання діяльності на два потоки (*перехід-розділення*). Перший з них залишається в цьому ж відділі і пов'язаний з отриманням оплати від клієнта за замовлений товар. Другий ініціює виконання дії по реєстрації замовлення у відділі продажів (модель товару, розміри, колір, рік випуску та ін.). Однак видача товару зі складу починається тільки після того, як буде отримана від клієнта

оплата за товар (*перехід-злиття*). Потім виконується підготовка товару до відправлення і його відправка клієнту у відділі продажів. Після завершення цих діяльностей замовлення закривається в відділі приймання і оформлення замовлень.

Об'єкти на діаграмі діятельності

Дії на діаграмі діяльності можуть здійснюватися над тими чи іншими об'єктами. Ці об'єкти або ініціюють виконання дій, або визначають результати цих дій. При цьому дії специфікують виклики, які передаються від одного об'єкта графа діяльності іншому. Оскільки у такому ракурсі об'єкти відіграють певну роль у розумінні процесу діяльності, іноді виникає необхідність явно вказати їх на діаграмі діяльності.

Слід нагадати, що базовим графічним зображенням об'єкта в нотації мови UML є прямокутник, в якому ім'я об'єкта підкреслюється. На діаграмах діяльності після імені може вказуватись характеристика стану об'єкта в прямих дужках. Такі прямокутники об'єктів приєднуються до стану дії відношенням залежності - пунктирною лінією зі стрілкою (рис. 1.64). Відповідна залежність визначає стан конкретного об'єкта після виконання попередньої дії.

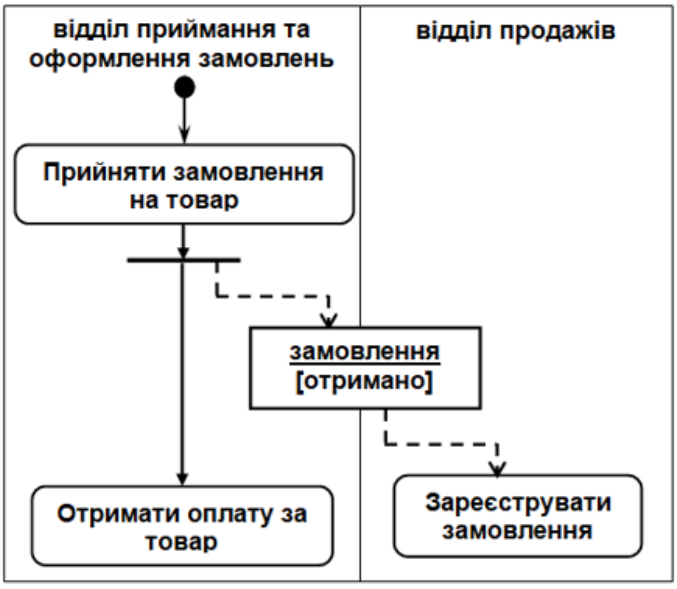


Рисунок 1.64 – Фрагмент діаграми діяльності з об’єктом-замовленням

Стосовно діаграм діяльності об'єкти, зазвичай, є екземплярами класів. Варто також зауважити, що на діаграмі діяльності той самий об'єкт може бути

зображений кілька разів, при цьому для виключення неузгодженості діаграми необхідно вказувати для них різні характеристики стану.

Перевагою діаграми діяльності є можливість візуалізувати окремі аспекти поведінки системи, що розглядається, або реалізації окремих операцій класів у вигляді *процедурної послідовності* дій. Таким чином, повна модель системи може містити одну або кілька діаграм діяльності, кожна з яких описує послідовність реалізації або найважливіших варіантів використання (типовий перебіг подій і всі винятки) або нетривіальних операцій класів.

1.3. Приклад розробки об'єктної моделі мовою UML

1.3.1. Постановка задачі в термінах предметного середовища

Задача

Розробити об’єктну модель мовою UML та програму мовою C++ (програму див. п. 2.1.4), яка б відображала поведінку системи, що складається з наступних елементів:

– АТС;
– абонент №111 (телефон);
– абонент №222 (телефон);
…
– абонент №NNN (телефон).

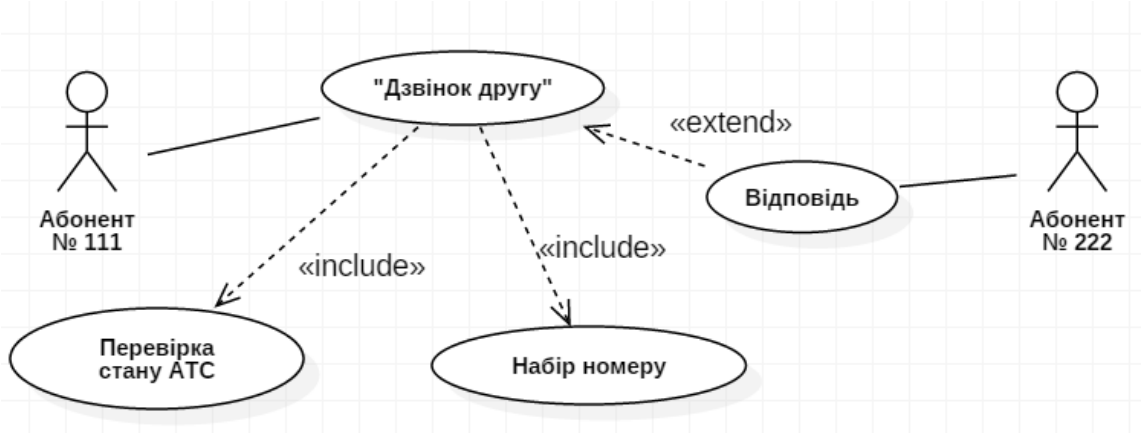
В моделі та програмі передбачити виконання наступних операцій (дій):

– опитування стану АТС («прослуховування» довгого гудка при піднятті телефонної трубки);
– набір номеру на будь-якому телефоні;
– встановлення з’єднання між будь-якими двома телефонами;
– відповідь на виклик абонента, що викликається.

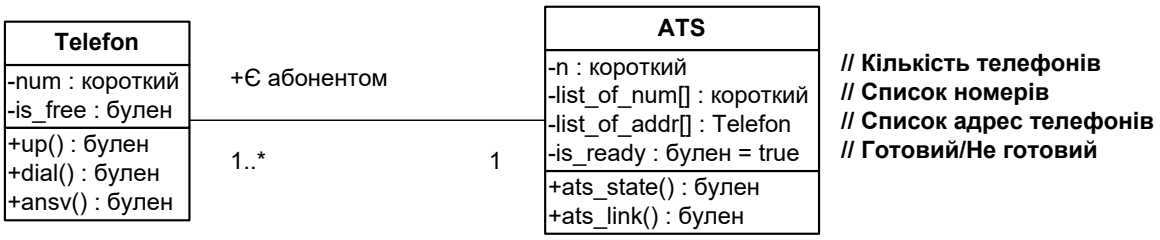
Далі представлено варіант об’єктної моделі для системи, до складу якої входить два телефони і АТС.

1.3.2. Розробка діаграм

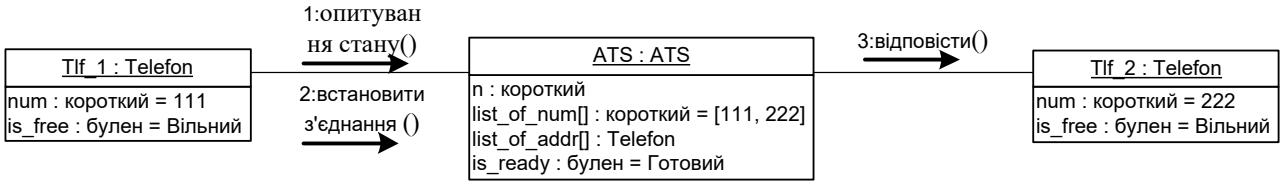
Діаграма варіантів використання



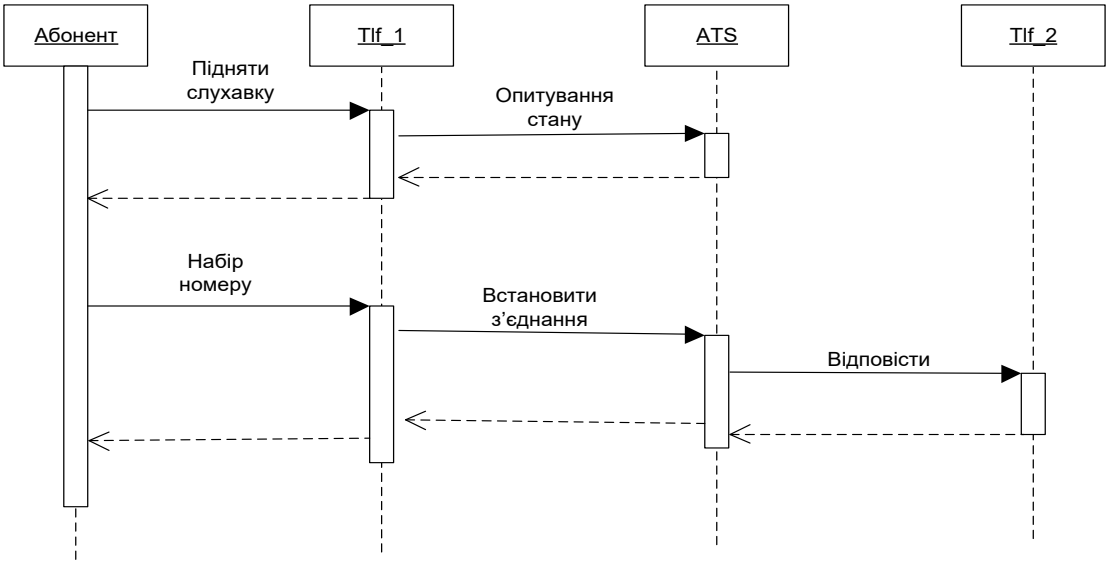
Діаграма класів



Діаграма кооперації



Діаграма послідовності



Програмна реалізація мовою C++ наведеної об'єктної моделі розглядається в розділі 2.1.4 даного навчального посібника.

Контрольні запитання та завдання

1. Якими чинниками зумовлене зростання складності програмного забезпечення? Навести приклади складних систем.

2. Навести приклади абстрактних моделей, що лежать в основі сучасних мов програмування.

3. Пояснити різницю між алгоритмічною (процедурною) і об’єктною (об’єктно-орієнтованою) декомпозицією при розробці програм.

4. Пояснити сутність об'єктно-орієнтованого підходу при створенні програм. Дати визначення термінам OOA, OOD, OOP.

5. Дати визначення поняттю «об'єктна модель». Дати характеристику її основним і додатковим елементам.

6. Пояснити призначення уніфікованої мови моделювання (UML) і застосування системи позначень на діаграмах мови.

7. Перелічити канонічні діаграми UML і їх призначення при розробці об’єктної моделі предметної області.

8. Розробити мовою UML проект об’єктної моделі системи обліку успішності студентів у класному журналі.

2. ТЕХНОЛОГІЯ ОБ’ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ МОВОЮ С++

2.1. Класи та об’єкти в мові С++. Абстрагування даних та інкапсуляція

2.1.1. Визначення класу, атрибутів, методів. Розмежування доступу до елементів класу

Програми на С++ будуються з використанням класів. Класи дозволяють групувати для певного об'єкта його дані й методи (функції), які оперують цими даними, в одній змінній. Класи дуже схожі на структури. Класи С++ є основою об’єктно-орієнтованого програмування. В даному розділі буде розглянуто наступні основні концепції побудови й використання класів.

1. У певному розумінні об'єкт являє собою сутність, з якою у програмах виконуються різні операції.

2. Програми на С++ створюють об'єкти за допомогою класів.

3. Клас, подібно структурі, містить елементи. Елементи класу можуть зберігати інформацію (дані) або бути функціями (методами), які оперують цими даними.

4. Кожен клас має унікальне ім'я.

5. Після визначення класу в програмі можна оголошувати об'єкти цього класу, використовуючи ім'я класу як тип.

6. Для звертання до елементів класу (як до даних, так і до функцій) в програмах використовують операцію **'крапка'** (для звернення через покажчик на об’єкт – операцію '**стрілка**').

7. Програми можуть визначати функцію класу в межах або поза визначенням класу. При визначенні функції поза визначенням класу, необхідно вказати ім'я класу з використанням оператора глобального розрішення '**::**' (кваліфікатора області бачення), наприклад **class :: function**.

Визначення класу в мові С++

Клас являє собою головний інструментальний засіб С++ для об’єктно-орієнтованого програмування. Клас дуже схожий на структуру, у якій згруповано елементи, що відповідають *даним* про об'єкт (*змінні класу*), і *функції* (*методи класу*), що оперують цими даними. Наприклад, у випадку, коли об'єктом є телефон, клас може містити такі елементи даних, як номер і тип телефону (кнопкової або дисковий) і функції, які може виконувати телефон, наприклад **dial**

(набрати номер), **answer** (відповісти), і **hang_up** (підняти слухавку). Групування даних про об'єкт та його функцій в одній змінній спрощує процес програмування й збільшує можливість повторного використання коду. Основні поняття, що будуть розглянуті нижче:

– для визначення класу у програмі необхідно вказати ім'я класу, елементи даних класу (змінні) й функції класу (методи);

– визначення класу – це визначення нового типу (абстрактного, користувальницького), який можна використовувати у програмах для створення об'єктів типу цього класу, подібно тому, як створюються змінні типу **int**, **char** і т.д.;

– значення елементам даних класу привласнюються із використанням операції **'крапка'** (або операції '**стрілка**');

– виклик функцій-елементів класу також виконується за допомогою операції **'крапка'** (або операції '**стрілка**').

В С++ програміст використовує клас для визначення об'єктів. Можна сказати, що об'єкт являє собою сутність. В об’єктно-орієнтованих програмах для зберігання інформації про різні реально існуючі сутності використовуються змінні, наприклад, про службовців, книги, файли тощо. Тобто, при об’єктно-орієнтованому програмуванні змінними можуть бути об’єкти, в яких зосереджено інформацію про предмети, що утворюють систему (предметну область), і операціях, які необхідно виконувати над цими предметами. Наприклад, для об'єкта-файла можна було б мати операції, які друкують, відображають або змінюють файл. Ціль полягає в тому, щоб включити в клас стільки інформації про об'єкт, скільки потрібно. Виходячи із цього, можна підібрати клас, створений для однієї програми, і використовувати його в декількох різних програмах.

Клас дозволяє програмам групувати дані й функції, які виконують операції над цими даними. Більшість авторів книг і статей про об’єктно-орієнтоване програмування називають функції класу ***методами***.

Формат визначення класу подібний до формату структури. Клас С++ повинен мати унікальне *ім'я*, за яким записується *відкриваюча фігурна дужка*, один або кілька *елементів*, *закриваюча фігурна дужка* і *крапка з комою*:

```
class Class_name                        // Ім’я класу - Class_name
{
        int     data_member;            // Елемент даних
        void    show_member(int);       // Функція (метод)
};
```

Після визначення класу можна оголошувати змінні типу цього класу (об'єкти), як показано нижче:

```
Class_name  object_one,  object_two,  object_three;
```

Наступна програма створює два об'єкти **employee**. Використовуючи операцію **'крапка'**, програма привласнює значення елементам даних Потім програма використовує елемент **show_employee** для виводу інформації про працівника:

Приклад 2.1:

```
#include <iostream>
#include <string.h>
using namespace std;
class employee
{ public:
        char name [64];
        long employee_id;
        float salary;
        void show_employee()
        {  cout << "Ім’я: " << name << endl;
           cout << "Номер службовця: " << employee_id << endl;
           cout << "Оклад: " << salary << endl; }          };

void main(void)
{       employee worker, boss;
        strcpy(worker.name, "Іван Приходько");
        worker.employee_id = 12345;
        worker.salary = 25000;
        strcpy(boss.name, "Євген Драч");
        boss.employee_id = 101;
        boss.salary = 101101.00;
        worker.show_employee();
        boss.show_employee();             }
```

Як можна бачити, програма оголошує два об'єкти типу **employee** - **worker** й **boss**, а потім використовує операцію **'крапка'** для привласнення значень елементам і виклику функції **show_employee**.

Визначення методів класу поза класом

У розглянутому класі **employee** функцію **show_employee()** було визначено всередині самого класу (вбудована (**inline**) функція). При збільшенні кількості функцій визначення вбудованих функцій в класі може внести безлад в опис класу. У якості альтернативного підходу, можна в межах оголошення класу записувати тільки прототип функції, а потім визначити функцію поза класом. Визначення класу із прототипом функції стає наступним:

```
class employee
{ public:
        char name[64];
        long employee_id;
        float salary;
        void show_employee();     //  Прототип функції
};
```

Внаслідок того, що різні класи можуть використовувати функції з однаковими іменами, необхідно випереджати імена визначених поза класом функцій ім'ям класу й *оператором глобального розрішення* (**::**). Тоді визначення функції стає наступним:

```
void employee :: show_employee ()       //  <Клас>::<Функція>
{       cout << " Ім'я: " << name << endl;
        cout << " Номер працівника: " << employee_id << endl;
        cout << " Оклад: " << salary << endl;     }
```

Як можна бачити, у наведеному коді перед визначенням функції записано ім'я класу (**employee**) і оператор глобального розрішення (**::**).

У наступній програмі визначення функції **show_employee** відбувається поза класом. При цьому, для зазначення імені класу використовується оператор глобального розрішення (кваліфікатор області бачення) (**::**).

Приклад 2.2:

```
#include <iostream>
#include <string.h>
using namespace std;
class employee
{
 public:
        char name [64];
```

```
        long employee_id;
        float salary;
        void show_employee();
};

void employee :: show_employee (void)          //  <Клас>::<Функція>
{       cout << " Ім'я: " << name << endl;
        cout << " Номер працівника: " << employee_id << endl;
        cout << "Оклад: " << salary << endl;     }

void main(void)
{       employee worker, boss;
        strcpy(worker.name, "Іван Приходько");
        worker.employee_id = 12345;
        worker.salary = 25000;
        strcpy(boss.name, "Євген Драч");
        boss.employee_id = 101;
        boss.salary = 101101.00;
        worker.show_employee();
        boss.show_employee();
}
```

Треба звернути увагу на використання мітки **public** усередині визначення класу. Елементи класу можуть бути *приватними* (чи *закритими*) **private** або *загальними* (чи *відкритими*) **public**, від чого залежить, як програми звертаються до елементів класу. У цьому прикладі всі елементи є загальними (**public**). Це означає, що програма може звертатися до будь-якого елемента, використовуючи операцію '**крапка**'.

Загальні і приватні елементи

Класи C++ містять елементи даних й методи (функції). Щоб указати спосіб звертання до елементів класів, C++ дозволяє визначати їх як *загальні* (**public**) і *приватні* (**private**). Програми можуть безпосередньо звертатися до будь-яких загальних елементів класу, використовуючи операції **'крапка'**. З іншого боку, до приватних елементів можна звернутися тільки через методи даного класу. Як правило, захищається більшість елементів даних класу, шляхом оголошення їх приватними. Отже, єдиним правильним способом, за допомогою якого програми можуть привласнити значення елементам даних, є використання функцій класу, які здатні перевірити й скорегувати ці значення.

Наступна програма ілюструє використання загальних і приватних елементів класу. Програма визначає тип **employee**, як показано нижче:

```
class employee
{
  public:
        int assign_values(char *, long, float);
        void show_employee();
        int change_salary(float);
        long get_id();
  private:
        char name [64];
        long employee_id;
        float salary;
};
```

Як можна бачити, клас захищає всі свої елементи даних, оголошуючи їх приватними. Для доступу до елементів даних програма повинна використовувати загальні функції, тобто оголошені, як **public**. Такі функції називаються *інтерфейсними функціями*. Нижче наведений повний текст цієї програми:

Приклад 2.3:

```
#include <iostream>
#include <string.h>
using namespace std;
class employee
{ public:
        int assign_values(char *, long, float);
        void show_employee();
        int change_salary(float);
        long get_id();
  private:
        char name [64];
        long employee_id;
        float salary;  };

int employee::assign_values(char *emp_name, long emp_id, float emp_salary)
{       strcpy(name, emp_name);
        employee_id = emp_id;
        if (emp_salary < 50000.0)
          { salary = emp_salary; return(0); }       // Успішно
        else    return(-1);     }         // Неприпустимий оклад

void employee::show_employee()
{       cout << " Службовець: " << name << endl;
        cout << " Номер службовця: " << employee_id << endl;
        cout << " Оклад: " << salary << endl;    }

int employee::change_salary(float new_salary)
```

```
{       if (new_salary < 50000.0)
        {   salary = new_salary;  return(0); }       // Успішно
   else return(-1);    }                  // Неприпустимий оклад

long employee::get_id()
{       return(employee_id) ;       }

void main(void)
{       employee worker;
        if (worker.assign_values((char *)"Приходько", 101, 10101.0) == 0)
          { cout << " Новий робітник: " << endl;
            worker.show_employee(); }
        if (worker.change_salary(35000.0) == 0)
          { cout << " Призначено оклад:" << endl;
            worker.show_employee(); }
        else    cout << " Вказано неприпустимий оклад " << endl;  }
```

Наведену програму варто пояснити більш докладно. Незважаючи на те що програма досить довга, її функції насправді дуже прості. Метод **assign_values** виконує ініціалізацію приватних даних класу. Метод застосовує оператор **if**, для перевірки припустимості розміру окладу. Метод **show_employee** виводить приватні елементи даних у вихідний потік. Методи **change_salary** й **get_id** являють собою інтерфейсні функції, що забезпечують програмі доступ до приватних даних. Після успішної компіляції й запуску цієї програми варто для перевірки доступності відредагувати її й спробувати звернутися прямо до приватних елементів даних з функції **main**, використовуючи операцію **'крапка'**. Внаслідок неможливості безпосередньо звертатися до приватних елементів, компілятор повідомить про синтаксичні помилки.

Таким чином:

– щоб управляти тим, як програми звертаються до елементів класу, C++ дозволяє визначати елементи як приватні (закриті) або загальні (відкриті);

– приватні елементи дають можливість класу сховати інформацію, яку програмі не потрібно знати.

– клас, що використовує приватні елементи, повинен мати *інтерфейсні функції*, які звертаються до приватних елементів класу.

Приховування інформації

Як вже відзначалось, клас містить *дані* (*змінні*) й *методи* (*функції*). Для використання класу програми повинні знати інформацію, яку зберігає клас (його елементи даних) і методи, які маніпулюють даними (функції). Програмам не потрібно знати, як працюють методи. Більше того, програми повинні знати тільки, яку задачу вирішують методи. У якості прикладу можна навести деякий клас **file**, який забезпечує виконання операцій з файлами. В ідеалі програми повинні знати тільки те, що цей клас забезпечує метод **file.print()**, який друкує відформатовану копію поточного файлу, або **file.delete()**, що видаляє файл. Програмі не потрібно знати, як ці два методи працюють. Інакше кажучи, програма повинна розглядати клас як «чорний ящик». Програма знає, які методи необхідно викликати і які параметри їм передати, але програма нічого не знає про реальну роботу, виконувану всередині класу (в «чорному ящику»).

Приховування інформації являє собою процес, у результаті якого програмі надається тільки мінімальна інформація, необхідна для використання класу.

Інтерфейсні функції

З метою зменшення кількості можливих помилок необхідно обмежувати доступ програм до даних класу, визначаючи елементи даних класу як приватні. Таким чином, програма не може звернутися до елементів даних класу, використовуючи операцію **'крапка'**. Замість цього клас повинен визначати *інтерфейсні функції*, за допомогою яких програма може привласнювати значення приватним елементам. Інтерфейсні функції у свою чергу, можуть перевіряти й коригувати значення, які програма намагається привласнити.

Забороняючи у такий спосіб прямий доступ до елементів даних, можна гарантувати, що їм завжди будуть привласнюватися коректні значення. Наприклад, нехай об'єкт **nuclear_reactor** використовує змінну з ім'ям **melt_down**, яка завжди повинна мати значення в діапазоні від 1 до 5. Якщо елемент **melt_down** є загальним, програма може безпосередньо звернутися до елемента, змінюючи його значення довільним, навіть неприпустимим, чином:

```
nuclear_reactor.melt_down = 101
```

Якщо замість цього зробити змінну **melt_down** приватною, то необхідно використовувати відкритий метод класу, наприклад **assign_meltdown()**, щоб

змінити значення цієї змінної. Як показано нижче, функція **assign_meltdown()** може перевіряти нове значення, щоб переконатися, що воно є припустимим:

```
int  assign_meltdown(int value)
{  if ((value > 0) && (value <= 5))
     melt_down = value;  return(0);          // Успішне привласнення
   else return(-1); }                        // Неприпустиме значення
```

Метод **assign_meltdown()** управляє доступом до елементів даних і являє собою ***інтерфейсну функцію***, яка використовується для захисту даних свого класу.

Використання оператора глобального розрішення для уникнення конфліктів імен

При уважному вивченні функцій у наведених прикладах програм можна виявити, що іменам параметрів функції часто передують символи **emp_**, як показано нижче:

```
int employee::assign_values(char *emp_name, long emp_id, float emp_salary)
```

Символи **emp_** використовувалися, щоб уникнути конфлікту між іменами параметрів й іменами елементів класу. Якщо подібний конфлікт імен все же відбувається, можна розрішити його, шляхом зазначення перед іменами елементів класу імені класу й *оператора глобального розрішення* (**::**). Наступна функція використовує оператор глобального розрішення й ім'я класу перед ідентифікаторами елементів класу. Це дозволяє легко зрозуміти, які імена належать класу **employee**:

```
int employee::assign_values(char *name, long employee_id, float salary)
{       strcpy(employee::name, name) ;
        employee::employee_id = employee_id;
        if (salary < 50000.0)
          {  employee::salary = salary;  return(0); }       // Успішно
        else    return(-1);                                 // Неприпустимий оклад
}
```

При створенні функцій, що працюють із елементами класу, необхідно використовувати ім'я класу й оператор глобального розрішення, щоб у такий спосіб уникнути конфлікту імен.

Використання оператора глобального розрішення перед іменами елементів класу дозволяє уникати не тільки конфліктів із іменами параметрів, як це показано у наведеному прикладі. При створенні функцій-елементів класу можливі ситуації, коли ім'я локальної змінної, яку оголошено в тілі функції, конфліктує з ім'ям елемента класу. За замовчуванням ім'я локальної змінної буде *перевизначати* ім'я елементу класу. Коли відбувається подібний конфлікт імен, у функції також треба використовувати ім'я класу й оператор глобального розрішення (**::**) для доступу до елементів класу:

```
Class_name :: member_name = some_value;
                // <Ім’я класу> :: <Змінна класу> = <Значення>
```

Приватні елементи класу не завжди є даними. У представленому прикладі приватними були тільки елементами даних. В процесі роботи над проектом визначення класу може стати значно складнішим і, цілком можливо, що виникне потреба створити функції для використання тільки методами того ж класу, але для іншої частини програми доступ до таких функцій повинен бути закритий. У подібних випадках такі методи необхідно також відносити до приватних елементів Якщо функція класу не оголошена як загальна, програма не може викликати таку функцію, використовуючи операцію **'крапка'**.

Іноді буває корисним заборонити деяким методам класу змінювати елементи даних свого ж класу. Такі методи називають ***константними методами***. Вони оголошуються із зазначенням ключового слова **const** після списку параметрів. Наприклад, метод **method_name()** не може виконувати запис в свій клас **Class_name**:

```
Class_name :: method_name() const {/* оператори */}
```

Висновки

Свідоме управління доступом програми до елементів класу, значно знижує можливість помилок, які відбуваються в результаті вільного використання цих елементами. Щоб управляти доступом до елементів класу, можна в, першу чергу, використовувати приватні елементи. Більшість визначень класів C++, які зустрічаються на практиці, використовують ак приватні, так і загальні елементи. Однією з найбільш часто використовуваних операцій, яка виконується при створенні об’єкта у програмі, є ініціалізація елементів даних об'єкта. Далі буде

показано, що C++ дозволяє визначати спеціальну функцію, звану ***конструктором***, яка автоматично викликається щоразу при створенні об'єкта. Використовуючи конструктор, програма легко може забезпечити автоматичну ініціалізацію елементів класу при створенні об’єкту. Приступаючи до розгляду наступного питання слід переконатися що засвоєні наступні основні концепції:

1. Елементи класу можуть бути *загальними* або *приватними*. Програми можуть прямо звертатися до загальних елементів, використовуючи операцію **'крапка'**. До приватних елементів можна звертатися, використовуючи тільки методи класу.

2. Якщо це не задано явно, C++ вважає всі елементи *приватними*.

3. Програми привласнюють значення й звертаються до приватних елементів, використовуючи *інтерфейсні функції*.

4. Для уникнення можливих конфліктів імен при створенні програми, що маніпулює елементами класу, можна записувати ім’я кожного елемента у форматі **<Ім’я класу>::<Ім’я елементу>**, де подвійна двокрапка (**::**) – це *оператор глобального розрішення* (*кваліфікатор області бачення*), наприклад: **employee::name**.

2.1.2. Конструктори класів

При створенні об'єктів однією з найбільш часто використовуваних операцій, яка виконується у програмах на C++, є ініціалізація елементів даних об'єкта. Як вже було зазначено, єдиним способом, за допомогою якого можна звернутися до приватних елементів даних класу, є використання функцій-методів цього ж класу. Щоб спростити процес ініціалізації елементів даних класу, C++ використовує спеціальну функцію – ***конструктор***, який запускається для кожного створюваного в програмі об'єкта. Також, C++ забезпечує функцію, звану ***деструктором***, що запускається при знищенні об'єкта.

Призначення конструктора і деструктора зрозуміти досить просто, незважаючи на «лякаючи» терміни. Можна представити конструктор як функцію, що допомагає будувати (конструювати) об'єкт. Подібно цьому, деструктор являє собою функцію, що допомагає знищувати об'єкт. Деструктор звичайно використовується, якщо при знищенні об'єкта потрібно звільнити пам'ять, що займав об'єкт.

Конструктор – це спеціальна функція-метод класу, яку C++ автоматично викликає щоразу при створенні об'єкта типу даного класу. Звичайне призначення конструктора полягає в ініціалізації елементів даних об'єкта. Конструктор має таке ж ім'я, як і клас. Наприклад, клас із ім'ям **file** використовує конструктор з ім'ям **file**. В програмі конструктор визначається так само, як і будь-який метод класу. Головне розходження полягає в тому, що конструктор не повертає значення і тому при його оголошенні перед ім'ям не вказується тип значення. Коли оголошується об'єкт, конструктору можна передавати параметри, як показано нижче:

```
Class_name object(valuel, value2, value3)
    // Ім'я_класу  об'єкт(значення1, значення2, значення3)
```

Головне, що характеризує конструктор:

– конструктор – це метод класу, що спрощує ініціалізацію елементів даних класу;

– конструктор має таке ж ім'я, як і клас;

– конструктор не повертає значення;

– щоразу, коли у програмі створюється змінна класу, C++ викликає конструктор класу.

– якщо клас має більше ніж один конструктор, то вони повинні розрізнятися числом і типом своїх параметрів;

– конструктор є у кожного класу; якщо в програмі для класу явно не визначено жодного конструктора, C++ автоматично створить для цього класу *конструктор за замовчуванням*.

Як приклад, можна розглянути конструктор класу **Window**:

```
Window::Window (Length x0, Length y0, Length width, Length height)
{  xmin = x0;  ymin = y0;  xmax = x0 + width; ymax = y0 + height;  }
```

При звертанні до цього конструктора породжується нове вікно з координатами нижнього лівого кута (**x0**, **y0**), шириною **width** і висотою **height**. Поряд з розглянутим у цьому класі можуть бути визначені й інші конструктори:

```
Window (Length x0, Length y0);  // стандартні розміри вікна
Window ();                      // стандартні розміри і положення вікна
```

У мові C++ підтримується три способи створення об’єктів, в залежності від типу пам'яті, що виділяється для розміщення об'єктів: у фіксованій глобальній пам'яті (*статична пам'ять*, **static**), у стеку (*автоматична пам'ять*, **auto**), «в купі» (*динамічна пам'ять*, **heap**).

Щоб розмістити об'єкт у статичній пам'яті досить оголосити його поза будь-якою функцією, або при його оголошенні вказати ключове слово **static**. Статична пам'ять виділяється компілятором під час компіляції програми й не звільняється під час її виконання. Конструктор можна використовувати для ініціалізації об'єктів, розташованих у статичній пам'яті. Оголошення (поза межами будь-якої функції)

```
Window  main_window = Window(0.0, 0.0, 8.5, 11.0);
```

або

```
Window  main_window(0.0, 0.0, 8.5, 11.0);
```

визначає статичний об'єкт **main_window**, проініціалізований значеннями параметрів конструктора.

Те ж саме оголошення, якщо воно виконується в межах деякої функції, породжує локальний об'єкт, який розміщується в автоматичній пам'яті (у стеку). Їх також можна ініціалізувати за допомогою конструктора.

Нарешті, звертання до конструктора в операторі **new** під час виконання програми породжує об'єкт, розташований у динамічній пам'яті. Наприклад:

```
Window  * window = new Window(0.0, 0.0, 8.5, 11.0);
```

Значенням виразу **new Window (0.0, 0.0, 8.5, 11.0)** є покажчик на породжений динамічний об'єкт. В цьому випадку для доступу до загальних елементів класу замість операції '**крапка**' треба використовувати операцію '**стрілка**' **(->)**:

```
window -> xmin = 0.0;
window -> ymin = 0.0;
```

Приклад створення простого конструктора

Як вже було відмічено, конструктор являє собою метод класу, що має таке ж ім'я, як і клас. Наприклад, якщо в програмі оголошено клас із ім'ям **employee**,

конструктор також буде мати ім'я **employee**. Подібно цьому, для класу з ім'ям **student** конструктор буде мати ім'я **student**. Якщо в програмі визначений конструктор, C++ буде автоматично викликати його щоразу, коли створюється новий об'єкт відповідного типу. Наступна програма створює клас із ім'ям **employee**. Програма також визначає конструктор з ім'ям **employee** який привласнює початкові значення об'єкту. Однак конструктор не повертає ніякого значення, незважаючи на те, що він не оголошується як **void**. Замість цього для нього просто не треба вказувати тип значення, що повертається:

```
class employee
{ public:
        employee(char *, long, float);  //  Конструктор (оголошення)
        void show_employee();
        int change_salary(float);
        long get_id();
  private:
        char name [64];
        long employee_id;
        float salary;
};
```

Визначення конструктора в програмі виконується так само, як будь-якого іншого методу класу:

```
employee::employee(char *name, long employee_id, float salary)
{       strcpy(employee::name, name) ;
        employee::employee_id = employee_id;
        if (salary < 50000.0)
          employee::salary = salary;
        else                            //  Неприпустимий оклад
          employee::salary = 0.0;
}
```

Як можна бачити, конструктор не повертає значення в точку його виклику. Для нього також не використовується тип **void**. У даному прикладі конструктор використовує оператор глобального розрішення й ім'я класу перед ім'ям кожного елемента для уникнення конфліктів імен. Повний текст програми з конструктором наведений нижче.

Приклад 2.4:

```
#include <iostream>
#include <string.h>
```

```
using namespace std;
class employee
{ public:
        employee(char *, long, float);      //  Конструктор
        void show_employee();
        int change_salary(float);
        long get_id();
  private:
        char name [64];
        long employee_id;
        float salary;   };

employee::employee(char *name, long employee_id, float salary)
{       strcpy(employee::name, name) ;
        employee::employee_id = employee_id;
        if (salary < 50000.0)
          employee::salary = salary;
        else                                //  Неприпустимий оклад
          employee::salary = 0.0;           }

void employee:: show_employee ()
{       cout << " Ім'я: " << name << endl;
        cout << " Номер працівника: " << employee_id << endl;
        cout << "Оклад: " << salary << endl;     }

void main(void)
{       employee worker((char *)"Іван Приходько ", 101, 10101.0);
        worker.show_employee(); }
```

Треба звернути увагу, що за оголошенням об'єкта **worker** в круглих дужках записані початкові значення для ініціалізації, як і при виклику функції. Таким чином, коли використовується конструктор, йому треба передавати параметри при оголошенні об'єкта:

```
employee worker((char *)"Іван Приходько ", 101, 10101.0);
```

Якщо в програмі буде потрібно створити декілька об'єктів типу **employee**, за допомогою конструктора можна виконати ініціалізацію елементів кожного з них, як показано нижче:

```
employee worker((char *)"Іван Приходько", 101, 10101.0);
employee secretary((char *)"Євгенія Драч", 57, 20000.0);
employee manager((char *)"Євген Драч", 102, 30000.0);
```

Конструктори й параметри за замовчуванням

В програмах на C++ дозволяється вказувати значення *за замовчуванням* для параметрів функції. Якщо користувач не вказує яких-небудь параметрів, функція буде використовувати значення за замовчуванням. Конструктор не є виключенням; у програмі можна вказати для нього значення за замовчуванням так само, як і для будь-якої іншої функції. Наприклад, наступний конструктор **employee** встановлює за замовчуванням значення окладу (**salary**) рівним **10000.0**, якщо програма не вказує оклад при створенні об'єкта. Проте програма повинна вказати ім'я службовця і його номер:

```
employee::employee(char *name, long employee_id,
                   float salary = 10000.00)
{       strcpy(employee::name, name);
        employee::employee_id = employee_id;
        if (salary < 50000.0)
          employee::salary = salary;
        else  employee::salary = 0.0; }     // Неприпустимий оклад
```

Перевантаження конструкторів

Мова C++ дозволяє програмам перевантажувати визначення функцій, вказуючи альтернативні функції для інших типів параметрів. C++ дозволяє також перевантажувати конструктори. Наступна програма перевантажує конструктор **employee**. Перший конструктор вимагає, щоб програма вказувала ім'я службовця, номер службовця й оклад. Другий конструктор пропонує користувачу ввести необхідний оклад, якщо програма не вказує його:

```
employee::employee(char *name, long employee_id)
{       strcpy(employee::name, name);
        employee::employee_id = employee_id;
        do {    cout << " Введіть оклад для " << name
                        << " меньший за $50000: ";
                cin >> employee::salary;   }
        while (salary >= 50000.0);
}
```

У визначенні класу необхідно вказувати прототипи для обох конструкторів, як показано нижче:

```
class employee
{ public:
        employee (char *, long, float);      // Прототип конструктора
        employee(char *, long);              // Прототип конструктора
        void show_employee();
```

```
        int change_salary(float);
        long get_id();
  private:
        char name [64];
        long employee_id;
        float salary;   };
```

Нижче наведено повний текст програми з перевантаженим конструктором.

Приклад 2.5:

```
#include <iostream>
#include <string.h>
using namespace std;
class employee
{ public:
        employee (char *, long, float);      // Прототип конструктора
        employee(char *, long);              // Прототип конструктора
        void show_employee();
        int change_salary(float);
        long get_id();
  private:
        char name [64];
        long employee_id;
        float salary;   };

employee::employee(char *name, long employee_id, float salary)
{       strcpy(employee::name, name);
   employee::employee_id = employee_id;
   if (salary < 50000.0) employee::salary = salary;
   else          employee::salary = 0.0;     }         // Неприпустимий оклад

employee::employee(char *name, long employee_id)
{       strcpy(employee::name, name);
   employee::employee_id = employee_id;
   do{cout<<"Введіть оклад для "<<name<<" меньший за $50000";
        cin >> employee::salary;    }
   while (salary >= 50000.0);       }

void employee:: show_employee()                  //  <Ім'я класу>::<Ім'я функції>
{       cout << " Ім'я: " << name << endl;
        cout << " Номер працівника: " << employee_id << endl;
   cout << "Оклад: " << salary << endl; }

void main(void)
{   employee worker("Іван Приходько", 101, 10101.0);
   employee manager("Євген Драч", 102);
   worker.show_employee();
   manager.show_employee();             }
```

Після компіляції й запуску на виконання цієї програми, на екрані монітору з'явиться запит ввести оклад для менеджера з ім’ям **Євген Драч**. Після вводу окладу, програма відобразить інформацію про обох службовців.

2.1.3. Деструктори класів

Кожна із розглянутих дотепер програм створювала об'єкти на самому початку свого виконання, просто оголошуючи їх. При завершенні програм об'єкти знищувались автоматично, без втручання з боку програми. Якщо в програмі визначений *деструктор*, C++ буде автоматично викликати його для кожного об'єкта, коли програма завершується (тобто коли об'єкти знищуються). Подібно конструктору, деструктор має таке ж ім'я, як і клас об'єкта. Однак у випадку деструктора його ім'я повинно починатися з символу **тильда** (~), як показано нижче:

```
~class_name() {     /*       Оператори деструктора  */       }
```

Наприклад:

```
Window::~Window ()
{  /* перефарбувати звільнену частину екрану  */  }
```

Деструктор автоматично запускається щоразу, коли програма знищує об'єкт. Найбільшу користь від застосування деструкторів можна отримати при роботі з об’єктами, для яких пам’ять розподіляється динамічно, тобто під час виконання програми. Необхідність у цьому виникає, коли на початку виконання програми можуть бути ще невідомі розміри об’єкту. Прикладом можуть бути списки об'єктів, що збільшуються або зменшуються при виконанні програми. Щоб створити такі динамічні списки, програма для зберігання об'єктів розподіляє пам'ять динамічно. У таких випадках для припинення існування об'єктів можна використовувати спеціальний метод, що є одним з членів класу й має назву ***деструктор***.

Головне, що характеризує деструктор:

– коли виконується знищення об'єкта, C++ викликає спеціальну функцію-*деструктор*, який може звільняти пам'ять (або інші ресурси), очищаючи її після об'єкту;

– *деструктор* має таке ж ім'я, як і клас, за винятком того, що перед його ім’ям необхідно зазначити символ **тильда** (~);

– *деструктор* не повертає значення;

– на відміну від конструктора, *деструктору* не можна передавати параметри (крім **void**);

– *деструктор* є у кожного класу;

– якщо в програмі *деструктор* для класу явно не визначений, C++ автоматично створить для цього класу *деструктор за замовчанням*.

Наступна програма визначає деструктор для класу **employee**:

```
employee::~employee()
{       cout << "Знищення об’єкта для " << name << endl;  }
```

У цьому випадку деструктор просто виводить на екран повідомлення про те, що C++ знищує об'єкт. Коли програма завершується, C++ автоматично викликає деструктор для кожного об'єкта. Нижче наведена реалізація програми з деструктором:

Приклад 2.6:

```
#include <iostream>
#include <string.h>
using namespace std;
class employee
{ public:
        employee(char *, long, float);
        ~employee(void);
        void show_employee();
        int change_salary(float);
        long get_id();
  private:
        char name [64] ;
        long employee_id;
        float salary;  };

employee::employee(char *name, long employee_id, float salary)
{       strcpy(employee::name, name) ;
        employee::employee_id = employee_id;
          if (salary < 50000.0)  employee::salary = salary;
        else    employee::salary = 0.0;     }

        employee::~employee()
{       cout << "Знищення об’єкта для " << name << endl;  }
```

```
void employee:: show_employee()
{       cout << " Ім'я: " << name << endl;
        cout << " Номер працівника: " << employee_id << endl;
        cout << "Оклад: " << salary << endl;     }

void main(void)
{       employee worker((char *)"Іван Приходько", 101, 10101.0);
        worker.show_employee(); }
```

Після компіляції й запуску на виконання цієї програми, на екрані монітору з'явиться наступний текст:

```
Ім'я: Іван Приходько
Номер працівника: 101
Оклад: 10101
Знищення об’єкту для Іван Приходько
```

Як можна бачити, програма автоматично викликає деструктор, без будь-якого явного виклику функції деструктора. У більшості простих програмам, ймовірно, не має потреби у використанні деструктора. Однак, у програмах, що розподіляють пам'ять динамічно в самих об’єктах, може виявитися, що деструктор забезпечує зручний спосіб звільнення пам'яті при знищенні об'єкта.

Висновки

Підводячи підсумки розгляду *конструкторів* і *деструкторів*, можна зауважити, що їм властиві риси звичайних функцій-методів класу, але існують і важливі особливості:

– *конструктори* й *деструктори* не можуть визначатися як функції, що повертають значення (навіть типу **void**);

– *конструктори* можуть мати аргументи; у якості параметрів можуть бути і такі, що одержують значення за замовчуванням; *деструктор* не має параметрів;

– ім'я *конструктора* збігається з ім'ям класу; ім'я *деструктора* – це ім'я класу, якому передує символ '~' (**тильда**);

– неможливо одержати адресу *конструктора* або *деструктора*;

– якщо вони явно не описані, то *конструктори* й *деструктори* автоматично створюються компілятором C++;

– *конструктор* не може бути викликаний як звичайна функція; *деструктор* може бути викликаний як звичайна функція із вказівкою повного імені:

```
Class_name  s(100), *ps=new Class_name(50);
s.~Class_name();            // явне знищення s
ps -> ~Class_name();        // виклик деструктора через покажчик
```

– компілятор автоматично вставляє виклики *конструктора* й *деструктора* при оголошенні й знищенні об'єкта;

– *конструктори* можуть бути і у закритій, і у захищеній, і у відкритій частині класу; якщо *конструктор* перебуває в закритій частині класу, то об'єкти цього класу з використанням такого конструктора можуть створювати тільки ті, хто має доступ до закритої частини класу;

– *деструктор* автоматично викликається компілятором:

- при виході з області бачення;
- після створення тимчасових об'єктів при перетвореннях у виразах або при передачі функціям аргументів;
- коли повернуті функцією значення більше не потрібні;
- при виконанні операції **delete** для об'єктів, розміщених у динамічній пам'яті.

– *конструктор* не може бути ні **const**, ні **volatile**, ні **static**, ні **virtual**;

– *конструктор* може бути викликаний трьома еквівалентними способами (користувачеві надається право вибору):

```
Class_name  good(100);
Class_name  bad = Class_name(100);
Class_name  ff = 100;
```

– якщо в класі є *конструктор*, то він викликається завжди, коли створюється об'єкт класу; якщо в класі є *деструктор*, то він викликається завжди, коли об'єкт класу знищується;

– об'єкти можуть створюватися як:

- *автоматичний* об'єкт: створюється щоразу, коли його оголошення зустрічається при виконанні програми, і знищується щоразу при виході з блоку, у якому воно (оголошення) з'явилося;
- *статичний* об'єкт: створюється один раз, при запуску програми, і знищується один раз, при її завершенні;
- об'єкт у *динамічній* пам'яті: створюється за допомогою операції **new** і знищується за допомогою операції **delete**; при створенні об’єкта у

динамічній пам’яті звертання до його елементів виконується за допомогою операції **стрілка** '->' (знаки мінус і більше);

- г) *об'єкт-елемент*: як елемент даних іншого класу;

– визначення *конструктора* може мати *список ініціалізації*, який записується через двокрапку після списку параметрів у форматі **<елемент даних>(<значення>)**; наприклад, конструктор класу **A** використовує список ініціалізації для привласнення значення 0 елементу **x**, і значення параметра **p** елементу **y**:

```
class A
{       int x,y;
  public:
        A(int p): x(0), y(p) { }};
```

– *конструктори* використовуються для ініціалізації константних полів і полів, які є змінними класу; константне поле повинно бути ініціалізоване в кожному визначенні конструктора; класи з константними полями повинні мати щонайменше один конструктор:

```
class Class_name
{       const int size;         // Константний елемент даних
  public:
        Class_name(int i): size(i) { }          };
```

Заключний приклад демонструє роботу з динамічною пам’яттю в *конструкторі* і *деструкторі*.

Приклад 2.7: Клас «**Рядок**»

```
#include <iostream>
#include <string.h>
using namespace std;
class StringHolder
{       char *contens;
  public:
        StringHolder(char *aString=0);              // Конструктор, оголошення (прототип)
        ~StringHolder();                        // Деструктор, оголошення (прототип)
        char *getContens();
        void setContens(char *aString);   };

StringHolder::StringHolder(char *aString)       // Конструктор, визначення
{       if(aString) { contens = new char[strlen(aString)+1];
                      strcpy(contens,aString); }
        else {contens = new char[1]; *contens='\0'; }    }
```

```
StringHolder::~StringHolder() { delete contens; }  // Деструктор

char *StringHolder::getContens() { return contens; }

void StringHolder::setContens(char *aString)
{       delete contens;
        contens=new char[strlen(aString)+1];
        strcpy(contens,aString);    }

void main()
{       StringHolder str1, str2((char *)"Object2"), *str3, *str4;  // Об'єкти і покажчики
        str3=new StringHolder((char *)"Object3");                  // Об'єкт у динамічній пам'яті
        str4=new StringHolder;  // Об'єкт у динамічній пам'яті
        cout << str1.getContens(); cout << str2.getContens();
        cout << str3->getContens(); cout << str4->getContens();      }  /*звертання за
        допомогою операції стрілка "->"  */
```

2.1.4. Програмна реалізація простої об'єктної моделі

Раніше в даному навчальному посібнику (див. п.1.3) наведено приклад об'єктної моделі системи, до складу якої входить АТС та два телефонних апарати. Програмна реалізація такої системи мовою С++ розглянута в наступному прикладі.

```
// Програма моделювання телефонної мережі: АТС и два телефони
#include <iostream>
#include <string.h>
using namespace std;
class telefon
{
private:
        int num;              // Номер
        bool is_free;         // Вільний/Зайнятий
public:
        telefon(int n, bool s) : num(n), is_free(s)  // Конструктор
                {cout << "Constructed telefon N " << num << endl; };
        bool up(class ats *);                        // Підняти слухавку
        bool dial(class ats *, int);                 // Набрати номер
        bool ansv(int);                              // Відповісти
};
class ats
{
private:
        int n;                                       // Кількість телефонів
        int * list_of_num;                           // Список номерів
        telefon ** list_of_addr;                     // Список адрес телефонів
```

```
        bool is_ready;                          // Готова/Не готова
public:
        ats(int n1, int * ln, telefon * la[], bool s) : // Конструктор
                n(n1), list_of_num(ln), list_of_addr(la), is_ready(s)
                { cout << " Constructed ATS\n"; };
        bool ats_state(void)                            // Опитування стану
                {if (is_ready) {cout << "Long be-e-e-e-ep " << endl; return true;}
                else { cout << "Beep,beep,beep,...\n"; return false; }};
        bool ats_link(int);                             // Встановити з'єднання
};

bool telefon::up(class ats * ats)
{       return (ats->ats_state());    }

bool telefon::dial(class ats * ats, int num)
{       return (ats->ats_link(num));        }

bool telefon::ansv(int n)
{       if (n==num && is_free)
{cout << endl << "Telefon N " << num << " Answ: \"Allo!\"" << endl; return true;}
        else return false;    }

bool ats::ats_link(int num)
{       int n;
        bool rez = false;
        for (n=0; n<ats::n; n++)
        { if(list_of_num[n] == num) { rez = true; break;} }
        return rez ? (list_of_addr[n])->ansv(num) : false;  }

void main ()
{       int list[] = {111, 222};
        telefon tlf_1(111, true), tlf_2(222, true);
        telefon * list_of_addr[] = {&tlf_1, &tlf_2};
        ats ats(2, &list[0], &list_of_addr[0], true);
        tlf_2.up(&ats);                 // підняти слухавку
        tlf_2.dial(&ats, 111); }        // встановити з'єднання
```

2.1.5. Статичні функції та елементи даних

В усіх прикладах програм, які було розглянуто, кожен створений об'єкт мав власний набір елементів даних (змінних). В залежності від призначення додатку можливі ситуації, коли різні об'єкти одного і того ж класу повинні разом використовувати один або декілька елементів даних. Наприклад, треба розробити програму, в якій 1000 об'єктів класу **A** мають однакове значення змінної **int i**. В такій програмі можна було б спільно використовувати таку змінну для всіх об'єктів типа **A**. У такий спосіб в програмі зменшується необхідна кількість пам'яті, заощаджуючи 999 копій однакової інформації. Для спільного використання елемента класу цей елемент повинен бути оголошений, як **static** (*статичний*). Основні концепції цього питання:

– C++ дозволяє створювати об'єкти (одного класу), які спільно використовують один або декілька елементів класу;

– якщо в програмі привласнюється значення елементу, що використовується спільно, то всі об'єкти цього класу одразу ж отримують доступ до цього нового значення;

– для створення елемента даних класу, який буде використовуватися спільно, треба оголошувати цей елемент класу з типом пам'яті **static**;

– після оголошенні в програмі елемента класу, як **static**, в даній програмі необхідно оголосити глобальну змінну (поза визначенням класу), яка має ім'я і тип такі ж, що і призначений для спільного використання елемент класу;

– в програмі можна використовувати ключове слово **static**, щоб зробити метод класу таким, який може бути викликаний навіть тоді, коли в програмі, можливо, ще не оголошено жодного об'єкту даного класу.

Спільне використання елементів даних

Як правило, при створенні об'єктів певного класу кожен об'єкт отримує свій власний набір елементів даних. Проте, можливі такі обставини, при яких об'єктам одного і того ж класу необхідно спільно використовувати один або декілька елементів даних (*статичні елементи даних*). У таких випадках треба оголошувати елементи даних, як загальні або приватні, і вказувати для них клас пам'яті **static**, як зазначено нижче:

```
private:
   static  int  shared_value;
```

Після оголошення класу необхідно оголосити такий елемент, як глобальну змінну поза класом, наприклад:

```
int  class_name::shared_value;
```

Наступна програма визначає клас **book_series**, в який входить призначений для спільного використання елемент **page_count**, тобто він є однаковим для всіх об'єктів (книг) класу **book_series**. Якщо в програмі виконується зміна значення цього елемента, то змінене значення одразу ж з'являється в усіх об'єктах класу.

Приклад 2.8:

```
#include <iostream>
#include <string.h>
using namespace std;
class book_series
{ public:
     book_series(char *, char *, float);
     void show_book();
     void set_pages(int) ;
  private:
     static int page_count;   // Статичний елемент класу
     char title[64];
     char author[ 64 ];
     float price;                  };

int book_series::page_count;       // Глобальна змінна

void book_series::set_pages(int pages) { page_count = pages; }

book_series::book_series(char *title, char *author, float price)
{       strcpy(book_series::title, title);
     strcpy(book_series::author, author);
     book_series::price = price;     }

void book_series:: show_book()
{       cout << "Заголовок: " << title << endl;
     cout << "Автор: " << author << endl;
     cout << "Ціна: " << price << endl;
     cout << "Сторінки: " << page_count << endl;                }

void main()
{       book_series programming( "Учимся программировать на C++",
                                    "Jamsa", 22.95);
     book_series word( "Учимся работать с Word для Windows",
                              "Wyatt", 19.95);
     word.set_pages(256); // Зміна вплине на всі об'єкти
```

```
    programming.show_book();
    word.show_book();
    cout << endl << "Зміна page_count " << endl;
    programming.set_pages(512); // Зміна вплине на всі об’єкти
    programming.show_book();
    word.show_book();              }
```

Як можна бачити, елементи даних **page_count** оголошено в класі, як **static int**. Безпосередньо за визначенням класу елемент **page_count** оголошено, як глобальну змінну. Коли програма змінює елемент **page_count**, ці зміни одночасно розповсюджується по всіх об’єктах класу **book_series**.

Використання елементів класу, якщо об’єкти ще не існують

Як вже було сказано, при оголошенні елемента класу, як **static**, цей елемент спільно використовується всіма об’єктами даного класу. Проте, можливі ситуації, коли у програмі ще не створено жодного об’єкта, але їй необхідно використовувати цей елемент. Для використання елементу у програмі необхідно оголосити його як **public** і **static**. Наприклад, наступна програма використовує елемент **page_count** з класу **book_series**, навіть якщо об’єкти цього не існують.

Приклад 2.9:

```
#include <iostream>
#include <string.h>
using namespace std;
class book_series
{ public:
     static int page_count;            // Статичний елемент класу
  private:
     char title [64];
     char author[64];
     float price;                 };

int book_series::page_count;       // Глобальна змінна

void main()
{    book_series::page_count = 256;  //   об’єкти ще не існують
   cout << "Значення page_count: " << book_series::page_count ;  }
```

В даному випадку, оскільки клас визначає елемент класу **page_count** як **public** і **static**, програма може звернутися до цього елемента класу, навіть при відсутності об’єктів класу **book_series**.

Статичні функції-методи

Попередня програма проілюструвала використання статичних елементів даних. Подібним чином C++ дозволяє визначати статичні елементи-функції (методи). При створенні в програмі статичного методу, можна викликати такий метод, навіть якщо об'єктів не було створено. Наприклад, якщо клас містить метод, який може бути застосований для даних поза класом, то цей метод можливо зробити статичним. Нижче наведено клас **menu**, який використовує **esc**-послідовність драйверу **ANSI.SYS** для очищення екрану дисплея. Якщо у комп'ютері встановлено драйвер **ANSI.SYS**, то можна використовувати метод **clear_screen** для очищення екрану. Оскільки цей метод оголошено як *статичний* і *загальний*, у програмі можна використовувати його, навіть якщо об'єкти типу **menu** не існують. Наступна програма **clr_screen.cpp** використовує метод **clear_screen** для очищення екрану дисплея.

Приклад 2.10:

```
#include <iostream>
using namespace std;
class menu
{ public:
     static void clear_screen(void);         // Статичний метод, оголошення
     // Тут можна розмістити інші методи
  private:
     int number_of_menu_options;                  };

void menu::clear_screen(void)              // Визначення статичного методу
{        cout << '\033' << "[2J";      }

void main()
{        menu::clear_screen();  /*  об'єкти ще не існують */ }
```

В програмі функцію **clear_screen** оголошено як *статичний метод*, тому можна використовувати цю функцію для очищення екрану, навіть при відсутності об'єктів типу **menu**. Функція **clear_screen** використовує **esc**-послідовність **ANSI Esc[2J** для очищення екрану.

Для виклику такої функції в програмі необхідно використовувати оператор глобального розрішення:

```
menu :: clear_screen();
```

Висновки

Якщо перед оголошенням елементів даних класу зазначено ключове слово **static**, то всі об’єкти даного класу можуть спільно використовувати цей елемент. Коли елемент даних є *загальним*, в програмі можна звертатися до його значень, навіть якщо об’єкти цього класу не існують. Аналогічно цьому, якщо перед оголошенням *загального* методу класу поставити ключове слово **static**, в програмі буде можливо використовувати цю функцію для операцій, які не включено в об’єкти класу. Основні розглянуті концепції підсумовано нижче.

1. Коли елемент класу оголошено **static**, такий елемент може спільно використовуватися всіма об’єктами даного класу.

2. Після оголошення елементу класу, як **static**, необхідно поза визначенням класу оголосити глобальну змінну, яка відповідає елементу класу, який використовується спільно.

3. При оголошенні елементу, як **public** і **static**, програма може використовувати такий елемент, навіть якщо об’єкти даного класу не існують. Для звернення до цього елемента необхідно використовувати оператор глобального розрішення, наприклад, **class_name::member_name**.

4. Якщо в програмі оголошено загальну статичну функцію-елемент класу, програма може викликати цю функцію, навіть якщо об’єкти даного класу не існують. Для виклику даної функції в програмі необхідно використовувати оператор глобального розрішення, наприклад **menu::clear_screen()**.

2.2. Перевантаження функцій та операцій

2.2.1. Перетворення типів, що визначаються класом

Поліморфізм – це засіб для надання одному повідомленню різних значень залежно від типу оброблюваних даних. ***Перетворення*** – це явна або неявна зміна значення залежно від типу. Перетворення забезпечує форму поліморфізму. *Перевантаження функцій* надає одному імені функції різні значення. Одне й те саме ім'я має різні інтерпретації, які залежать від вибору функції. Обрана функція задовольняє *алгоритму відповідності сигнатур* для C++. Така форма поліморфізму називається ***спеціальний поліморфізм***.

Операції перевантажуються й вибираються також на підставі *алгоритму відповідності сигнатур*. Перевантаження операцій надає їм нові значення. Наприклад, вираз **a+b** має різні значення, залежно від типів змінних **a** й **b**. Перевантаження операції '+' для типів, визначених користувачем, дозволяє використовувати їх у виразах додавання у більшості випадків так само, як і вбудовані типи. При визначенні функції перетворення припустимі також вирази змішаного типу.

Один із принципів ООП полягає в тому, що визначені користувачем типи повинні мати ті ж привілеї, що і вбудовані типи. Типи, вбудовані в базову мову, можуть змішуватися у виразах, оскільки це зручно, але, з іншого боку, при цьому не завжди просто визначати послідовність необхідних перетворень.

Явне перетворення

Механізм *явного перетворення* об'єктів будь-якого типу (класу) в інший тип забезпечується застосуванням операції *приведення* (*перетворення*) *типів*. Явне перетворення типів у виразах мови C/C++ застосовується тоді, коли неявне перетворення небажане або коли без нього вираз є неприпустимим. Мова C++ надає засоби інтеграції абстрактного (користувальницького) типу даних (АТД) і вбудованих типів. Це досягається завдяки механізму функції-члена класу, що забезпечує явне перетворення. *Функціональний запис* явного перетворення:

```
ім'я_типу(вираз)
```

еквівалентний застосуванню *операції приведення типів*. Тип у функціональному записі явного перетворення повинен бути представлений, як ***ідентифікатор*** типу. Таким чином, наступні два вирази еквівалентні:

```
x=(float)i;
x=float(i);
```

Вираз

```
p=(int *)x;          // припустиме приведення
```

не може безпосередньо функціонально виражатися як

```
p=int *(x);          // помилка! int * - не ідентифікатор
```

Однак, для цього може бути використане **typedef**

```
typedef  int  *  int_ptr;
p = int_ptr(x);      // припустиме приведення
```

Неявне перетворення

Перетворення відбувається *неявно*:

– у виразах при виконанні операцій с даними різних типів;
– при передаванні параметрів функціям;
– в значеннях, що повертають функції.

Конструктор з одним аргументом

Конструктор з одним аргументом забезпечує перетворення з типу аргументу до типу класу конструктора. Наприклад:

```
String(const char *p) { len=strlen(p); s=new char[len+1]; strcpy(s, p); }
```

Представлений конструктор класу **String** забезпечує перетворення типів від **char*** до **String**. Воно доступно як *явно*, так і *неявно*. Явно воно використовується або як *операція перетворення* (*приведення*), або у функціональній формі. Таким чином, можливі два правильних варіанти коду:

```
String  s;
char * logo="Hello!";
s=String(logo);      // привласнення з явним перетворенням до типу String
```

та

```
s=logo;              // неявний виклик перетворення в операції привласнення
```

Даний код – приклад перетворення з деякого існуючого типу до типу, визначеного користувачем.

Функція перетворення

Користувач не може додавати конструктори вбудованих типів (таких як **int**, **char** та ін.). Але необхідність у перетворенні з користувальницького типу у вбудований тип може виникнути, наприклад, з **String** в **char ***. Це може бути здійснено за допомогою визначення в класі спеціальної *функції перетворення*. Загальна форма запису такої функції-методу класу:

```
operator <тип>() {...}
```

Така функція перетворення повинна відповідати наступним вимогам:

– бути нестатичною функцією-методом класу;
– не визначати тип значення, що повертається, у заголовку;
– мати порожній список аргументів;
– повертати значення, приведене до типу **<тип>**.

Функція перетворення **operator <тип>()** забезпечує перевантаження операції приведення типу визначеного класом до типу **<тип>**.

Для перетворення типу **String** до типу **char *** в класі **String** необхідно визначити функцію перетворення наступним чином:

```
operator char * () { char * s; … return s; }
```

Таким чином, перетворююча функція-метод у формі **A::operator B()** і конструктор у формі **B::B(const A &)** забезпечують перетворення з типу об'єкта **A** у тип об'єкта **B**.

Далі наведено приклади перетворення об'єктів одного класу в об'єкти іншого класу за допомогою *конструктора з одним параметром* і *функції перетворення*.

Якщо необхідно перетворити об'єкт класу **OldType** в об'єкт класу **NewType**, то клас **NewType** повинен мати конструктор з параметром типу **OldType** або **OldType &**:

```
class OldType{       …       };

class NewType
{ public:
     NewType(void) {}
     NewType(OldType &old) { }   };

void main()
{        OldType oldtype;
   NewType newtype=oldtype;}  // перетворення конструктором
```

Або з використанням функції перетворення типів **operator NewType(void)**:

```
class NewType
{ public:
        NewType(void) {}    };

class OldType
{ public:
        operator NewType() { return NewType(); }                };

void main()
{       OldType oldtype;
        NewType newtype=oldtype;  }     // перетворення функцією
                                        // operator NewType()
```

Треба зазначити, що одночасне використання для перевантаження однієї і тієї ж операції конструктора й функції приведення неможливо. У зв'язку з тим, що функції перетворення не мають ні явних аргументів, ні значення, що повертається, вони не можуть бути перевантажені. Може бути кілька операторів перетворення для різних типів. Операції перетворення успадковуються й можуть бути віртуальними.

2.2.2. Перевантаження та вибір функцій

Можливість перевантаження функцій – важлива особливість мови C++. Перевантажена функція обирається у відповідності зі списками параметрів, які задано при звертанні до функції й при оголошенні функції. При виклику перевантажених функцій компілятор діє у відповідності до певного алгоритму. Цей алгоритм залежить від того, які перетворення типів доступні.

Найкраща відповідність повинна бути унікальною. Вона повинна бути найкращою, щонайменше для одного параметра, а для всіх інших параметрів бути такою ж, як будь-яка інша відповідність.

Алгоритм відповідності *для кожного параметра* наступний.

1. Використати точну відповідність, якщо її знайдено.
2. Перевірити підтримку стандартних типів.
3. Перевірити стандартні перетворення типів.
4. Перевірити перетворення, обумовлені користувачем.

5) Використати відповідність для аргументів, якщо її знайдено.

Нижче наведено приклад програми, в якій продемонстровано можливості C++ щодо перевантаження та вибору функцій в залежності від типів параметрів і наявності засобів їх перетворення.

Приклад 2.11:

```
#include <iostream>
using namespace std;
class complex
{       double real, imag;
public:
        // Конструктор перетворення double в complex:
   complex(double r) { real=r; imag=0; }
   void assign(double r,double i) { real=r; imag=i; }
   void print() { cout << real << "+"<<imag<<"i"; }
      // Функція перетворення complex в double:
   operator double() { return sqrt(real*real+imag*imag); }     };

int greater(int i, int j) { return (i > j) ? i : j; }
double greater(double i, double j) { return (i > j) ? i : j; }
complex greater(complex i, complex j) { return (i > j) ? i : j; }

void main()
{   int i=5, j=10;   float x=7;  double y=14;
   complex w(0), z(0);
   w.assign(x,y);              // перетворення float x в double x
   z.assign(i,j);              // перетворення (i,j) в double
// 1. Вибір першого визначення greater
//    згідно алгоритму відповідності
        cout << greater(i,j) << endl;
// 2. Вибір другого визначення greater внаслідок
//    застосування стандартного перетворення з float в double
        cout << greater(x,y) << endl;
// 3. Обрано друге визначення greater
//    внаслідок точної відповідності
        cout << greater(y,double(z)) << endl;
```

```
// 4. Точна відповідність третьому визначенню greater
    cout << greater(w,z) << endl;      }
```

Функція перетворення **operator double()** – метод класу **complex**, потрібна для перетворення типу **complex** в **double**. Вона застосовується у цій програмі тричі:

1) неявно перетворення при виконанні порівняння **w>z**;

2) у виклику функції **greater(y, double(z))** – явне перетворення **double(z)**;

3) неявне перетворення при виконанні виводу **cout << greater(w,z)**.

Для запобігання неоднозначності в третьому виклику функції **greater(y, double(z))** необхідне явне перетворення **double(z)**. Якщо замість цього буде використовуватися виклик функції:

```
greater(y,z); // помилка !!!
```

то він буде мати два доступних перетворення, що забезпечують відповідність. Перетворення параметра '**y**' з **double** в **complex** визначено користувачем, як перетворення конструктором. Воно відповідає третьому визначенню функції **greater**. Перетворення '**z**' з **complex** в **double**, також обумовлене користувачем, відповідає другому визначенню. Але друге визначення функції має кращу відповідність для першого параметра, в той час як третя функція має кращу відповідність для другого параметра. Це порушує вимогу про те, що «*Максимальна відповідність повинна бути унікальною. Вона повинна бути найкращою щонайменше для одного параметру й однаковою для всіх інших*».

2.2.3. Дружні функції

Клас може надавати особливі привілеї певним зовнішнім функціям або функціям-методам іншого класу. Ці функції одержали назву ***дружніх***. Якщо функцію або клас оголошено *дружніми* даному класу, то такі дружні функції або функції-методи такого (дружнього) класу можуть здійснювати безпосередній доступ до всіх елементів класу, для якого вони дружні. Дружні функції й класи можуть здійснювати прямий доступ до закритих елементів класу без використання інтерфейсних функцій-методів цього класу.

Ключове слово **friend** – специфікатор, що надає функції – не члену класу доступ до прихованих елементів класу. Він використовується для того, щоб вийти за строгі рамки типізації й приховування даних у C++.

Однією з причин використання дружніх функцій є те, що деяким функціям необхідно надавати привілейований доступ більш, ніж до одного класу. Друга причина – **friend**-функція передає всі параметри через список фактичних параметрів, і значення кожного з них підлягає перетворенню, сумісному із призначенням. Такі перетворення застосовуються неявно до переданих аргументів-класів і тому особливо корисні у випадках перевантаження операцій.

Функція оголошується дружньою у визначенні того класу, якому вона дружня. Перед іменем функції записується ключове слово (специфікатор) **friend**, і її оголошення може знаходитися як в **public** так й в **private** частині класу, що не вплине на значення. Функція-метод одного класу може бути **friend**-функцією іншого класу. Це відбувається тоді, коли таку функцію оголошено **friend** в класі з використанням оператора розрішення контексту для визначення імені класу, членом якої є дружня функція. Якщо всі функції-методи одного класу є **friend**-функціями іншого класу, то це можна визначити записом: **freind class <ім'я класу>.**

Наприклад:

```
class t1
{   friend void a();    // friend-функція
    int b();      };      // функція-метод класу t1

class t2
{   friend int t1::b();  }; // функція-метод класу t1 дружня класу t2

class t3
{   friend class t1;    }; // всі методи класу t1 є дружніми класу t3
```

Наступний приклад містить оголошення класів **matrix** і **vector**. Функція **mpy()** виконує множення вектора на матрицю. Вона повинна мати доступ до **private**-членів обох класів. Ця функція буде **friend** для обох класів.

```
class matrix;
class vect
{       int *p;
        int size;
        friend  vect mpy(const vect &, const matrix &);
public:
```

```
        // < Відкрита частина класу vect >
};

class matrix
{       int **base;
        int row, column;                // розмірність матриці
        friend  vect mpy(const vect &, const matirx &);
public:
        // < Відкрита частина класу matrix >
};

vect mpy(const vect &v, const matrix &m)
        {       if( v.size != m.row ) { exit(1); }
                vect ans(column);
                // < Реалізація алгоритму множення >
                return ans;    }
```

У прикладі застосовується попереднє оголошення класу **matrix**. Воно необхідне тому, що **mpy()** з'являється в обох класах, і використовує кожен клас для визначення типів аргументів.

Тобто **friend**-функції можна розглядати як частину загального інтерфейсу класу. Існують ситуації, у яких вони можуть бути альтернативою інтерфейсним функціям-методам. Використання **freind**-функцій є спірним, тому що вони порушують інкапсуляцію **private**-членів класів. Парадигма ООП стверджує, що об'єкти (у C++ вони є змінними класу) доступні через їх **public** елементи. Тільки функції-методи класу повинні мати доступ до прихованої реалізації класу. Це ясний і строгий принцип проектування. Проте, **freind**-функція дозволяє виходити за його межі, оскільки має доступ до **private**-членів, в той час, як сама не є функцією-методом даного класу. За її допомогою можна організувати швидкий код для доступу до подробиць реалізації класу.

2.2.4. Перевантаження операцій

Для перевантаження вбудованих операцій C++ використовують ключове слово **operator**. Їм може бути привласнено ряд нових значень, які залежать від параметрів. У такий спосіб операції '*' можна привласнити додаткові значення:

```
vect oprator*(const vect &, const matrix &)
```

де '*' є двомісною (з двома операндами) операцією множення.

```
matrix t; vect s, r; s = t * r; // Замість s=mpy(r, t);
```

Перевантажену операцію можна викликати також із використанням функціональної форми запису:

```
s=operator*(t, r);
```

Хоча операціям і можуть додаватися нові значення, але їхній пріоритет залишається без змін. Наприклад, операція множення має більше високий пріоритет, ніж додавання.

Перевантажувати можна не всі операції. Неможна використовувати аргументи за замовчуванням. Неможна перевантажувати оператори **'.'**, **'::'**, **'? :'**, **'sizeof'**.

Доступними для перевантаження є всі арифметичні, логічні операції, операції порівняння, привласнення, оператори порозрядних операцій, префіксні й постфіксні форми операцій інкременту й декременту. Можуть бути перевантажені операції індексації **'[ ]'** і звертання до функції **'( )'**. Також можуть бути перевантажені операція покажчика класу **'->'** й операція покажчика на елемент класу **'->*'**. Можливе перевантаження **new**, **delete**.

Перевантаження унарної операції

Одномісні (унарні) операції типу **'!'**, '++', '~', **'[ ]'** застосовуються для обробки одного операнда.

Приклад перевантаження операції інкременту **'++'** для класу **clock**:

```
class clock
{       unsigned long sec;
  public:
    // конструктор з одним параметром:
    clock(unsigned long s):sec(s) { }
    void tick() { sec++; }
    clock operator++() { tick(); return *this; } };   // перевантаження

void main()
{       clock t(100);
        ++t;    }   // застосування перевантаженого оператору
```

Цей клас перевантажує **префіксну** операцію збільшення **'++'**. Перевантажена операція являє собою функцію-метод класу **clock**. Функція

перевантаження **operator++()** також оновлює неявну змінну типу **clock** і повертає оновлене значення.

Префіксну операцію **'++'** також можна перевантажити, використовуючи **friend**-функцію.

```
clock operator++(clock &s1) { s1.tick(); return s1; }
```

Змінна типу **clock** повинна збільшуватися тому вона передається по посиланню. Рішення про вибір між **friend** і функцією-методом звичайно залежить від того, наскільки необхідні й доступні оператори неявного перетворення. Явна передача аргументу, як в **friend**-функції, дозволяє автоматичне його приведення.

Операції інкременту й декременту можуть бути перевантажені як для *префіксної*, так і для *постфіксної* форми запису.

Префіксна форма оголошується як метод класу, що не має аргументів, або як дружня функція з одним аргументом, що представляє собою посилання на об'єкт даного класу.

Постфіксна форма оголошується як метод класу, що має один аргумент типу **int**, або як дружня функція з двома аргументами: посилання на об'єкт даного класу й аргумент типу **int**. Аргумент типу **int** насправді не використовується. Фактично він має нульове значення.

```
class X
{ public:
    X & operator ++();              // Префіксна форма
    X & operator ++(int); };        // Постфіксна форма

class Y
{ public:
    friend Y & operator ++(Y &);          // Префіксна форма
    friend Y & operator ++(Y &, int); };  // Постфіксна форма
```

Можна узагальнити: **@x** інтерпретується як **x.opertor@()**, якщо **operator@** – функція-метод та **operator@(x)**, якщо **operator@** – дружня функція.

Аналогічно для **x@**: постфіксна операція **x@** інтерпретується як **x.opertor@(0)**, якщо **operator@** – функція-метод та **operator@(x, 0)**, якщо **operator@** - дружня функція.

Перевантаження бінарної операції

Якщо бінарна операція перевантажується за допомогою функції-метода класу, то у якості свого першого аргументу вона одержує неявно передану змінну класу, а другий аргумент – це єдиний аргумент у списку параметрів функції перетворення.

```
// Приклад невірного використання
clock i1, i2;
i2 = 6 + i1;              // Запис у такий спосіб є неприпустимим
```

Це означало б: «*передати об'єкту **6** повідомлення **'+'** з параметром **i1***». Чи можливо забезпечити такий запис? Так, якщо написати перевантаження операції **'+',** як дружню функцію.

При оголошенні **friend**-функції або звичайної функції в списку параметрів визначають обидва аргументи.

```
class clock
{       friend clock operator+(const clock &c1, const clock &c2);
};
clock  operator+(const clock &c1,const clock &c2)
{       clock temp(c1.sec+c2.sec); return temp;         }
```

Обидва параметри визначаються явно і є кандидатами для перетворення до типу **clock**. Використовуючи це визначення, можна записати:

```
int i=5; clock c(100);
c+i;    // припустимо: i - перетвориться в clock
i+c;    // припустимо: i - перетвориться в clock
```

У противагу цьому, можна перевантажити двомісний мінус **'-'** функцією-методом класу **clock**.

```
class clock
{       clock operator- (const clock &c)
          { clock temp(sec-c.sec); return temp; }  };
```

Тут застосовується перший неявний аргумент, який повинен мати тип **clock**. Ним є зменшуване перевантаженої операції **'мінус'**. Це може викликати асиметричне поводження двомісних операторів.

```
int i=5; clock c(100);
c-i;    // c.operator-(i) припустимо: i перетвориться в clock
i-c;    // i.operator-(c) неприпустимо: i не відноситься до clock
```

Можна визначити будь-яку двомісну операцію, наприклад, визначення операції множення **long** й **clock** може мати вигляд:

```
clock  operator*(long m, const clock &c)
{       clock  temp(m * c.sec); return temp;      }
```

В даному прикладі функція **operator*** має бути дружньою класу **clock**. Така реалізація вимагає застосовувати фіксований порядок запису операндів у виразах множення, який залежить від типу операндів. Для запобігання цього звичайно пишеться ще один перевантажений функціональний оператор.

```
clock operator*(const clock &c, long m)
{       clock temp(m * c.sec); return temp;       }
```

Загальне правило для бінарних операцій: **y@x** інтерпретується як **y.opertor@(x)**, якщо **operator@** – функція-метод й **operator@(y,x)**, якщо **operator@** – дружня функція.

Операція виклику функції

Оператор перевантаження операції виклику функції необхідно оголошувати, як нестатичну функцію-метод класу. Вона дозволяє користувачеві визначати типи і кількість параметрів перевантаженої операції:

```
class X
{       int a,b,c;
  public:
        X(int a1, int b1, int c1): a(a1),b(b1),c(c1) {}
        void operator() ( int i, int j, int k) { … };  // Перевантаження ( )
};
X ex(8,20,17);          // Викликається конструктор
ex(10,15,57);           // Викликається операція ()
```

Треба звернути увагу, що до об’єкта можна звертатися, як до функції. В подальшому це буде мати назву «**функціональний оператор**».

Перевантажені операції: функції-методи чи дружні функції

Якщо перевантажена операція реалізована як функція-метод класу, то їй або взагалі не передаються явно параметри, або передається один параметр. Дружній функції перевантаженої операції – передається один, або два параметри.

Якщо бінарна операція перевантажена як функція-метод, то її першим операндом є об'єкт, що приймає повідомлення. Отже, явно передається тільки один (другий) параметр операції.

Якщо унарна перевантажена операція реалізована як функція-метод, то її операндом є приймаючий повідомлення об'єкт. Таким чином, ця функція-метод не має явних параметрів.

Перевантажена операція не може мати більше двох параметрів.

Таблиця 2.1 – Приклади перевантаження операцій

Операція	**Функція-метод класу**	**Дружня функція**

Унарний мінус
 	
Обчислення адреси
Операція >
	class X {
	X operator-();
	
	X operator**&**();
	
	X operator**>**();
	// Помилка
	};
		class Y {
		friend Y operator-
		(const Y&);
		friend Y operator**&**(const Y&);
		
		friend Y operator**>**(const Y&);
		// Помилка
		};

Бінарний мінус
Побітове AND
	class X {
	X operator-(const X&);
	X operator**&**(const X&);
	};
		class Y {
		friend Y operator-(const Y&, const Y&);
		friend Y operator**&**(const Y&, const Y&);
		};

Перевантаження операції індексації

Перевантаження операцій індексації і привласнення пояснюється на прикладі класу **vect**, який визначено наступним чином:

```
class vect
{ private:
    int *p;
    int size;
  public:
    vect(int mysize);                    // створення вектора з mysize елеметів
    vect(const vect &v);                 // конструктор копіювання
    int & operator[ ](int i);            // перевантажене індексації
    vect& operator=(const vect &v);      // перевантажене привласнення
};

vect::vect(int n)              // створення вектора з mysize елеметів
{       p=new int[n]; size=n;
        for(int i=0; i < size; i++) p[i]=i;      }
```

```
vect::vect(const vect &v)   // конструктор копіювання
{       p=new int[v.size]; size=v.size;
        for(int i=0; i < size; i++) p[i]=v.p[i];        }
```

Перевантажена операція індексації **operator[](int i)** приймає цілий аргумент '***i***' (індекс) та перевіряє, чи знаходиться його значення в межах діапазону. Якщо так, це значення використовується для обчислення адреси індексованого елемента в пам'яті та повернення його у якості результату перевантаженої операції **'[ ]'**. Тип значення, що повертається, – посиланням на цілий тип **int &.**

```
int &vect::operator[ ](int i)
{       if(i < 0 || i >=size) { cerr << "Error in index\n"; exit(-1);}
        return p[i];    }
```

Функція перевантаження операції індексації має повертати посилання на цілий тип і вона має один параметр цілого типу – індекс. Функція перевантаження повинна бути нестатичної функцією-методом класу, для якого виконується перевантаження.

Необхідно ще раз наголосити, що перевантажена операція **'[ ]'** повинна повертати значення саме типу **int &**, а не **int**. Використання посилання у якості типу значення, що повертається, дозволяє використовувати перевантажену операцію індексування як із правої, так і з лівої сторони від операції привласнення. Крім того, недотримання цієї вимоги призведе до помилки при компіляції.

Перевантаження операції привласнення

У C++ операція привласнення **'='** за замовчуванням виконує рекурсивне поелементне копіювання одного об'єкта в інший. (У більш ранніх версіях використовувався механізм побітового копіювання.) Якщо операцію привласнення не перевантажено, результат виконання цієї операції над об'єктами типу **vect** буде таким (див. визначення класу вище):

```
vect v1(200), v2(200);
v2 = v1;        cout << v2[10];       // Результат = 10 (див.конструктор)
v1[10] =-50;  cout << v2[10];       // Результат =-50
```

Це обумовлено наявністю у класі елементів типу покажчик, значенням яких після створення об’єкту стає адреса динамічної пам'яті. Коли операцію привласнення не перевантажено, вона визначається за замовчуванням із семантикою, що представляє собою привласнення значення. Це називається «семантикою поверхневого копіювання» і не завжди припустимо. В розглянутому прикладі обидва вектори після привласнення займають одну і ту ж динамічну пам'ять, хоча після створення об’єктів кожному з них була розподілена власна ділянка (**p=new int[n]**). Перевантажувати операцію привласнення бажано, а іноді й необхідно, особливо, якщо клас містить покажчики на пам'ять, що розподіляється динамічно:

```
vect &vect::operator=(const vect &v)
{   int s=(size < v.size) ? size : v.size;
    for(int i=0; i<s; i++) p[i]=v.p[i];
    return *this;          // Дозволяє виконувати множинне привласнення
}
```

Перевантажена операція привласнення дає більш правильний результат:

```
vect v1(200), v2(200);
v2 = v1;         cout << v2 [10];       // Результат = 10
v1[10] =-50;  cout << v2[10];       // Результат = 10
```

Тип результату, що повертається, є посиланням на тип. Значенням, що повертає операція, є ***this**, тобто сам об’єкт (**this** – покажчик), що дозволяє виконувати множинне привласнення типу **v1=v2=v3**.

Читачам пропонується самостійно обґрунтувати різницю у результатах, які виводяться наступними двома фрагментами програм:

Приклад 1	Приклад 2

vect v1(200), v2(200), v3(200);
v2[100]=-111;
v1=v3=v2;
cout << " v2[100] = " << v2[100] << endl;
cout << " v1[100] = " << v1[100] << endl;
	vect v1(200), v2(200), v3(50);
	v2[100]=-111;
	v1=v3=v2;
	cout << " v2[100] = " << v2[100] << endl;
	cout << " v1[100] = " << v1[100] << endl;

Результат: v2[100] = -111 v1[100] = -111;	Результат: v2[100] = -111 v1[100] = 100;

За необхідності заборонити операцію привласнення для об’єктів іншого класу, її треба просто оголосити в закритій частині визначення класу.

Привласнення та ініціалізація

Операція виду '=' не завжди є операцією привласнення. Так, визначення нового об’єкта за допомогою конструкції

< ім’я класу> <об’єкт 2> = <об’єкт 1>;

не виконується, як привласнення, і навіть не називається привласненням. При створенні нового об’єкту зазначеним способом виконується операція його *ініціалізації*. Для виконання ініціалізації створюється спеціальний ***конструктор ініціалізації*** або ***конструктор копіювання***. Для кожного класу існує такий конструктор. Якщо користувач не перевизначає конструктор ініціалізації, використовується *конструктор ініціалізації за умовчанням*. Ініціалізація за умовчанням виконується шляхом простого поелементного (фактично, побайтового) копіювання вмісту об’єкта. Таке копіювання не є коректним в тому випадку, коли об’єкти містять посилання на інші об’єкти або змінні в динамічній пам’яті. В цьому випадку слід скористатися перевантаженим ***конструктором копіювання***, який має бути створений користувачем.

Конструктор копіювання викликається в наступних випадках:

– оголошення об’єкта у форматі **vect v2=v1**;

– повернення значення з функції. Значення, що повертається, - це новий об'єкт, який ініціалізується за допомогою конструктора;

– передача значень у функцію через параметри. Параметри - це локальні змінні і їх ініціалізація виконується за допомогою конструктора.

Конструктор копювання необхідно визначити у наступному вигляді: **vect::vect(&vect)**, тобто він повинен приймати один параметр типу посилання на власний клас.

Приклад коректного ***конструктора копіювання*** для класу **vect** може виглядати так:

```
vect::vect(const vect &v)   // конструктор копіювання
{       p=new int[v.size];
        size=v.size;
        for(int i=0; i < size; i++) p[i]=v.p[i];}
```

Повний текст програми, що містить всі розглянуті приклади наведений нижче.

Приклад 2.12:

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

class vect
{       int *p;
        int size;
public:
        vect(int n);                        // створення вектора з n елеметів
        vect(const vect &v);                // конструктор копіювання
        int & operator[ ](int i);           // перевантажена індексація
        vect& operator=(const vect &v);     // перевантажене
};                                          // привласнення

vect::vect(int n)                           // створення вектора з n елеметів
{       p=new int[n]; size=n;
        for(int i=0; i < size; i++) p[i]=i;     }

vect::vect(const vect &v)   // конструктор копіювання
{       p=new int[v.size]; size=v.size;
        for(int i=0; i < size; i++) p[i]=v.p[i];        }

int & vect::operator[ ](int i) // перевантажена індексація
{       if(i < 0 || i >=size) { cerr << "Error in index\n"; exit(-1);}
        return p[i];    }

vect &vect::operator=(const vect &v) // перевантажене привласнення
{       int s=(size < v.size) ? size : v.size;
        for(int i=0; i<s; i++) p[i]=v.p[i]; return *this;       }

void main()
{       vect v1(200), v2(200), v3(20);
```

```
v2 = v1;
cout << v2[10] << endl;              // Результат = 10
v1[10] =-50;
cout << v1[10] << endl;              // Результат =-50
cout << v2[10] << endl;              // Результат = 10

v2[100]=-111;
v1=v3=v2;
vect v4=v2;
cout << " v2[100] = " << v2[100] << endl;
cout << " v1[100] = " << v1[100] << endl;
cout << " v4[100] = " << v4[100] << endl;
v2[100]=-222;
cout << " v2[100] = " << v2[100] << endl;
cout << " v1[100] = " << v1[100] << endl;
cout << " v4[100] = " << v4[100] << endl;        }
```

Перевантажені операції new, delete, ->

Операції **new** й **delete** надають класу механізм керування динамічною пам'яттю.

Операції створення й знищення об'єктів у динамічній пам'яті можуть бути перевизначені в такий спосіб:

```
void *operator new(size_t size);
void  operator delete (void *);
```

де **void *** – покажчик на область пам'яті, виділену під об'єкт, **size** – розмір об’єкту в байтах.

Тип **size_t** є стандартним типом даних, що використовується замість **unsigned integer** в операціях з пам’яттю.

Тип значення, що повертається, має бути для перевантажених **new** – **void ***, а для **delete** – **void**. Неявно вони є статичними функціями-методами класу, отже не можуть бути а ні константними, ні віртуальними. Варіант реалізації цих функцій:

```
class T
{ public:
        void *operator new(size_t sz) { return malloc(sz); }
void operator delete(void *p) { free(p); }          };
```

Якщо операції **new** й **delete** перевантажені для класу, вони не можуть бути викликані для розподілення й звільнення пам'яті для масивів об'єктів даного класу (тобто, як **new T[ ]** та **delete [ ]**).

Операція **new** може бути перевантажена, щоб приймати додаткові аргументи (додатково до обов'язкового першого аргументу типу **size_t**, який повинен залишитися). При виклику перевантаженої операції додатково передані аргументи записуються в дужках після ключового слова **new**.

Приклад класу із перевантаженням операції **new**, якій передається додатковий аргумент для розміщення об'єкту за вказаною адресою:

```
class T
{ public:
        void * operator new(size_t sz, void *buf)
          { cout << "New new!\n"; return buf;  /* Розмістити T за адресою buf */   }
        void * operator new(size_t size)
           {cout << "Old new!\n"; return ::operator new(size); }  };

   char buffer[1000];
        T *p=new (buffer) T;
        T *p1 = new T;
```

Перший (не додатковий) аргумент **size_t size** в функцію передається, проте, тут не використовується. При створення власного механізму розподілення пам'яті він може бути застосований для обліку кількості вільної або вже зайнятої пам'яті.

Перевантажена операція **new** з додатковими аргументами приховує глобальну операцію **::operator new**. Тепер при звертанні до **new** зі звичайним синтаксисом відбудеться помилка.

```
T *p=new T;  // помилка: відсутній аргумент
```

Щоб знову було можливо використовувати звичайний синтаксис оператора **new**, в прикладі для класу **T** передбачено функцію-метод **new**, що має тільки один аргумент – **size_t size**. Вона явно викликає глобальну операцію **::operator new**.

Перевантажена операція '**->**' звичайно використовується в класах, які містять у якості свого елементу покажчик на інший клас. Перевантажена операція має повертати об'єкт або покажчик на об'єкт іншого класу.

```
class A
{ public:
```

```
        int  int_ptr;    };
class B
{   public:
   A *a;        // покажчик на клас A
   B(void) { a=new A; }      // конструктор
   A *operator->(void) { return a; }    };    // перевантаження ->

B b;
b->int_ptr=15;        // еквівалентно  b.a->int_ptr = 15;
```

Завдяки перевантаженню в даному прикладі ім'я об'єкту **b** може вільно використовуватися для звертання до відкритої частини класу **A**.

2.3. Успадкування та ієрархії класів. Просте та множинне успадкування

2.3.1. Механізм успадкування

Однією з переваг об'єктно-орієнтованого програмування є повторне використання створених класів, що може значно заощадити час і сили. Ця можливість в ООП називається ***успадкуванням***. В UML цьому терміну відповідає термін «***узагальнення***». Сутність ***успадкування*** полягає в наступному: якщо деякий клас вже створено, то можливі ситуації, коли новому класу потрібні деякі або навіть усі особливості вже існуючого класу, до яких необхідно лише додати один або декілька елементів даних або функцій. У таких випадках C++ дозволяє будувати новий об'єкт, використовуючи характеристики вже існуючого об'єкту. Іншими словами, новий об'єкт *успадковує* елементи існуючого класу (званого ***базовим класом***). Коли будується новий клас з існуючого, цей новий клас називається ***похідним класом***. Похідний клас отримує всі можливості базового класу, але його можна удосконалити за рахунок додавання власних атрибутів та методів. Базовий клас при цьому залишається незмінним. В даному розділі розглядається основні концепції *успадкування класів* в C++:

– якщо у програмі планується використовувати успадкування, то для породження нового класу потрібен *базовий клас*, тобто новий клас *успадковує* елементи базового класу;

– для ініціалізації елементів *похідного класу* програма повинна викликати конструктори базового і похідного класів;

– до елементів базового і похідного класів можна звертатися використовуючи операцію '**крапка**';

– крім загальних (**public** – доступних усім) і приватних (**private** – доступних тільки методам класу) елементів в С++ також використовуються захищені (**protected**) елементи, які доступні базовому і похідним класам;

– для розрішення конфлікту імен між елементами базового і похідного класів в програмі можна використовувати *оператор розширення (кваліфікації) області бачення* "**::**", вказуючи перед ним ім'я базового або похідного класу.

Успадкування є фундаментальною концепцією об'єктно-орієнтованого програмування, найбільш значущою, після класів. Виграш від нього полягає в тому, що успадкування дозволяє використовувати існуючий код кілька разів. Маючи написаний і відлагоджений базовий клас, його можна більше не модифікувати, при цьому механізм успадкування дозволяє пристосувати його для роботи в різних ситуаціях. Використання вже написаного коду заощаджує час і сили, а також збільшує надійність програми. Успадкування може допомогти і при початковій постановці задачі на програмування, і при розробці загальної структури програми. Програми, які розглядаються, показують, що на практиці успадкування реалізується дуже просто і може зберегти значні зусилля на програмування.

Базові і похідні класи

Для породження нового класу з існуючого в С++ використовується така форма запису:

```
class <ім’я похідного класу > :
      [ public | protected | private ] <ім’я базового класу>
{       <оголошення елементів похідного класу>    };
```

Механізм успадкування в С++ розглянемо на прикладі програми, що моделює роботу лічильника. Ця програма використовує у якості лічильника об'єкт класу **Counter**. При створенні лічильника його значення може бути ініціалізовано нулем або деяким числом за допомогою конструкторів. Збільшення значення лічильника виконується методом **operator++()**. Поточне значення можна прочитати за допомогою методу **get_count()**.

Можна припустити, що на створення і налагодження класу **Counter** витрачено багато часу і сил і він працює саме так, як необхідно. З часом може

з’ясуватися, що об’єктам класу **Counter** бракує однієї речі: в них відсутній метод для зменшення значення лічильника. Можливо, даний лічильник потрібно застосувати для підрахунку кількості відвідувачів деякої установи в конкретний момент часу. При вході відвідувача лічильник збільшує своє значення, при виході – зменшує.

Для вирішення цієї задачі можна було б вставити метод зменшення прямо в вихідний код класу **Counter**. Проте існує декілька причин, які доводять, що це робити недоцільно. По-перше, клас **Counter** прекрасно працює і на його налагодження витрачено певну кількість часу. Якщо втрутитися у вихідний код класу **Counter**, то його тестування і налагодження доведеться проводити спочатку, витрачаючи на це додатковий час.

По-друге, в деяких ситуаціях програміст просто не має доступу до початкового коду класу, наприклад, якщо він доступний, як частина бібліотеки класів у багатофайловій програмі.

Щоб уникнути цих проблем, можна застосувати успадкування для створення нового класу на базі **Counter**. Нижче представлено код програми, в якій з'являється новий клас **CountDn**, що додає операцію зменшення лічильника до класу **Counter**.

Приклад 2.13:

```
#include <iostream>
using namespace std;
class Counter                           // БАЗОВИЙ КЛАС
{ protected:
      unsigned int count;               // лічильник
  public:
      Counter() : count(0) {  }         // конструктор без аргументів
      Counter(int c) : count(c) {  }    // конструктор з 1-м аргументом
      unsigned int get_count() const    // константний метод
        { return count; }
      Counter operator ++ ()            // перевантаження ++
        { return Counter(++count); }            };

class CountDn : public Counter          // ПОХІДНИЙ  КЛАС
{ public:
      Counter operator -- ()                    // перевантаження --
        { return Counter(--count); }
};                      // { --count; return *this; }   // можливо і так

int main()
{       CountDn c1;                             // c1 - об’єкт класу CountDn
```

```
cout << "\nc1=" << c1.get_count();
++c1; ++c1; ++c1;                          // інкремент c1 (тричі)
cout << "\nc1=" << c1.get_count();
--c1; --c1;                                // декремент c1 (двічі)
cout << "\nc1=" << c1.get_count() << endl;
return 0;   }
```

Програма починається з визначення класу **Count**. Після опису класу **Count** в програмі визначено новий клас, **CountDn**. В нього включено новий метод **operator--()**, який зменшує лічильник. В той же час **CountDn** успадковує всі можливості класу **Counter**: конструктори і методи.

В першому рядку опису класу **CountDn** вказується, що він є *похідним* класом від **Counter**.

```
class CountDn : public Counter
```

Для цього використовується знак '**двокрапка**' (не плутати з подвійною двокрапкою, яка є операцією, що використовується для визначення області бачення), за ним йде ключове слово **public** і ім'я *базового* класу **Counter**. Таким чином встановлюється відношення між класами. Іншими словами, це говорить про те, що **CountDn** є *спадкоємцем* класу **Counter** (значення для успадкування ключового слова **public** буде розглянуто пізніше).

Для спрощення програми в її текст не включено перевантаження операцій постфіксного збільшення '++' і зменшення '--'.

Узагальнення на діаграмах класів UML

Як показано раніше, у UML *успадкування* називають ***узагальненням***, оскільки *базовий* клас – це більш загальна форма його *похідних* класів. Іншими словами, спадкоємець є більш спеціалізованою версією батьківського класу. Діаграму узагальнення для програми з прикладу 2.13 показано на рисунку 2.1.

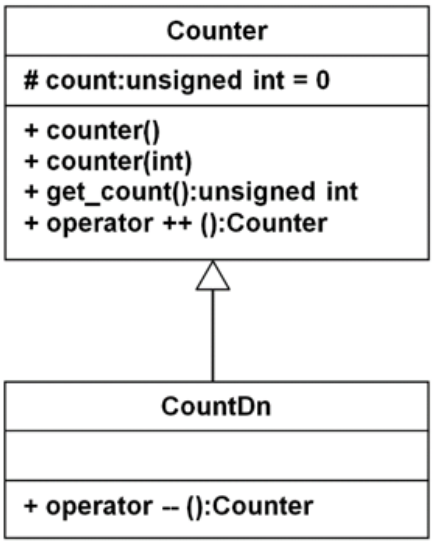


Рисунок 2.1 – Успадкування на діаграмі класів

На діаграмах UML узагальнення показується стрілкою з трикутним закінченням, що з'єднує базовий і похідний класи. Стрілка означає «*успадкування від*», «*похідний від*» або «*більш спеціалізована версія*». Напрям стрілки підкреслює, що похідний клас посилається на методи і поля базового класу, але при цьому базовий клас не має доступу до похідного.

Елементами діаграми є два класи **Counter** та **CountDn**, які представлені прямокутниками. Кожен клас поділений на три частини. У верхній частині записано назву класу, нижче розташовані *атрибути* (властивості, змінні), а ще нижче – *методи* (функції). Єдиний захищений (**protected**) атрибут **count** позначено префіксним символом '#'. Всі методи обох класів є відкритими (**public**), тому на діаграмі їм передує символ '+'.

Підстановка конструкторів базового класу

В головній функції **main()** програми з прикладу 2.13 створено об'єкт класу **CountDn**:

```
CountDn c1;
```

Цей рядок означає, що ***c1*** буде створений як об'єкт класу **CountDn** і ініціалізований значенням «нуль». Але в класі **CountDn** немає конструктора, яким же чином виконується ініціалізація? Виявляється, якщо в програмі не визначений конструктор похідного класу, то використовуватиметься *конструктор без параметрів* базового класу. Конструктори в класі-спадкоємці

автоматично викликають конструктор базового класу. У програмі конструктор класу **CountDn** відсутній, і компілятор використовує конструктор без параметрів класу **Counter**.

Така гнучкість компілятора – використання доступного методу замість відсутнього – звичайна ситуація, що виникає при успадкуванні.

Явний виклик конструктора базового класу

У попередньому прикладі конструктор базового класу в похідному класі було викликано автоматично, тобто неявно. У такий спосіб можливо виконувати ініціалізацію елементів базового класу лише за допомогою конструктора без параметрів. На практиці для ініціалізації частіше потрібно викликати конструктори з параметрами. Наступний приклад демонструє явний виклик конструктора базового класу у похідному. Це надає розробнику програм більш гнучкий спосіб налаштування створюваних об'єктів.

Наприклад, нехай є базовий клас **employee** (працівник):

```
class employee
{ public:
      employee(char *, char *, float);     // конструктор
      void show_employee();                // метод для перегляду
  private:
      char name[64];                       // ім'я працівника
      char position[64];                   // посада
      float salary;            };          // оклад
```

Також необхідно оголосити клас **manager**, який додає наступні елементи даних в клас **employee**:

```
      float annual_bonus;                  // щорічна премія
      char company_car[64];                // службовий автомобіль
      int stock_options;                   // пільги
```

В даному випадку можливо вибрати один з двох варіантів: по-перше, програма може створити новий клас **manager**, який дублює елементи класу **employee**, або, по-друге, програма може породити клас типу **manager** з базового класу **employee** шляхом успадкування. Мова C++ надає засоби реалізації обох варіантів створення нового класу **manager**, але перевагу слід віддати другому варіанту, тобто *успадкуванню*.

Для визначення нового (похідного) класу **manager** слід вказати ключове слово **class**, ім'я **manager**, двокрапку, обрати один із способів успадкування (**public**, **protected** або **private**), вказати ім'я базового класу **employee** і оголосити елементи нового класу, яких немає в класі **employee**:

```
class manager : public employee
{   /*  оголошення елементів класу manager  */   };
```

Ключове слово **public** перед ім'ям класу **employee** вказує, що загальні (**public**) елементи класу **employee** також є загальними і в класі **manager**. Наприклад, наступні оператори породжують клас **manager**.

```
class manager : public employee
{ public:
    manager(char *, char *, char *, float, float, int);       // конструктор
    void show_manager();                                      // метод для перегляду
  private:
    float annual_bonus;                    // щорічна премія
    char company_car[64];                  // службовий автомобіль
    int stock_options;              };     // пільги
```

При породженні класу з будь-якого базового класу приватні (**private**) елементи базового класу доступні похідному класу тільки через інтерфейсні функції базового класу. Таким чином, похідний клас не може звичайним чином безпосередньо звернутися до приватних елементів базового класу із використанням операції '**крапка**'.

Далі наведено повний текст програми, яка демонструє успадкування в C++ при створенні похідного класу **manager** з базового класу **employee**. В програмі реалізовано явний виклик конструктора базового класу у конструкторі похідного класу.

Приклад 2.14:

```
#include <iostream>
#include <string.h>
using namespace std;
// Оголошення базового класу:
class employee
{ public:
    employee(char *, char *, float);
    void show_employee(void);
  private:
    char name [ 64 ];
```

```
    char position[64];
    float salary;                 };

employee::employee(char *name, char *position, float salary)
{       strcpy(employee::name, name);
    strcpy(employee::position, position);
    employee::salary = salary;              }

void employee::show_employee()
{       cout << "Ім'я: " << name << endl;
    cout << "Посада: " << position << endl;
    cout << "Оклад: $" << salary << endl;        }

// Оголошення похідного класу:
class manager : public employee
{ public:
    manager(char *, char *, char *, float, float, int);
    void show_manager();
  private:
    float annual_bonus;
    char company_car[64];
    int stock_options;                  };

// Конструктор похідного класу
// викликає конструктор базового класу employee:
manager::manager(char *name, char *position, char *company_car,
                    float salary, float bonus, int stock_options) :
        employee(name, position, salary)
{       strcpy(manager::company_car, company_car);
        manager::annual_bonus = bonus;
        manager::stock_options = stock_options;       }
void manager::show_manager()
{       show_employee();
        cout << "Службове авто: " << company_car << endl;
        cout << "Щорічна премія: $" << annual_bonus << endl;
        cout << "Пільги: " << stock_options << endl;    }

void main()
{       employee worker((char*)"Джон Дой", (char *)"Програміст", 35000);
        manager boss((char *)"Джейн Дой", (char *)"Віце-президент ",
                "Lexus", 50000.0, 5000, 1000);
        worker.show_employee();
        boss.show_manager();     }
```

Як можна бачити, в програмі спочатку визначено *базовий* клас **employee**, а потім визначено *похідний* клас **manager**.

Треба звернути увагу на конструктор **manager**. Коли клас породжується з базового класу, у якому відсутній конструктор без параметрів (конструктор за умовчанням), *конструктор похідного класу повинен викликати конструктор базового класу*. Тому, якщо в базовому класі визначено хоча б один конструктор, в цьому базовому класі необхідно визначити також конструктор без параметрів. Щоб викликати конструктор базового класу, необхідно після конструктора похідного класу записати через двокрапку ім'я конструктора базового класу з необхідними параметрами:

```
manager::manager( char *name, char *position, char *company_car,
                  float salary, float bonus, int stock_options) :
                        employee(name, position, salary)
{       strcpy(manager::company_car, company_car);
        manager::annual_bonus = bonus;
        manager::stock_options = stock_options;       }
```

Конструктор для **employee** викликається, як частина ***списку ініціалізаторів***. Логічно створювати об'єкт базового класу перш, ніж може бути завершено створення повного похідного об'єкту. Також треба звернути увагу, що функція **show_manager** викликає функцію **show_employee**, яка є методом класу **employee**. Оскільки клас **manager** є похідним від класу **employee**, клас **manager** може звертатися до загальних елементів класу **employee**, начебто усі ці елементи були визначені усередині класу **manager**.

2.3.2. Керування доступом при успадкуванні

Захищені елементи класу

Однією з особливостей успадкування є те, що у похідному класі успадковані елементи можуть бути доступними не так, як для них це визначено у базовому класі. Для визначення доступності успадкованих елементів в заголовку похідного класу застосовуються ключові слова **public**, **protected** і **private** (***зовнішнє***, ***захищене*** і ***внутрішнє*** успадкування).

При визначені базових класів їх елементи оголошуються, як **public**, **private** і **protected** (*загальні*, *приватні* і *захищені*). Похідний клас може звертатися до *загальних* (**public**) елементів базового класу так, нібито їх визначено в самому похідному класі. З іншого боку, похідний клас не може безпосередньо звертатися

до *приватних* (**private**) елементів базового класу. Замість цього, для звернення до таких елементів похідний клас повинен використовувати *інтерфейсні функції* або *дружні функції*. *Захищені* (**protected**) елементи базового класу, можна сказати, займають місце посередині між *приватними* та *загальними*. Якщо елемент є захищеним (**protected**), то об'єкти похідного класу можуть звертатися до нього так, ніби він є загальним (**public**). Для решти частини програми доступ до захищених елементів закрито (як до приватних). Програми (поза межами похідного класу) можуть звертатися до захищених елементів тільки через *інтерфейсні* або *дружні* функції. Наступне визначення класу **book** використовує позначку **protected** для того, щоб дозволити похідним від класу **book** класам звертатися до елементів **title**, **author** і **pages** безпосередньо із використанням операції '**крапка**'.

```
class book
{ public:
        book(char *, char *, int);
        void show_book();
  protected:            // наступні елементи є захищеними
        char title [64];
        char author[64];
int pages;              };
```

Якщо програміст планує у майбутньому породжувати нові класи зі створюваного в даний час класу, він повинен оголосити захищеними (а не приватними) елементи, до яких майбутні похідні класи повинні звертатися безпосередньо, як до свої власних елементів.

Наступний приклад демонструє успадкування із застосуванням захищених елементів базового класу:

```
class student
{ protected:
        int id;
        char name[30];
  public:
        sudent(int i, char *nm);
        void print();     };

class grad_student : public student
{ protected:
        char thesis[30];
  public:
```

```
grad_student(int i,char *nm, char *t);
void print();     };
```

В даному прикладі **grad_student** – похідний клас, а **student** – базовий клас. Використання ключового слова **public** у заголовку визначення похідного класу означає, що **protected** і **public** члени класу **student** будуть успадковані як **protected** і **public** члени **grad_student** відповідно. Елементи **private** є недоступними. Успадкування із застосування ключового слова **public** в заголовку похідного класу називають ***загальним*** або ***зовнішнім успадкуванням***. Загальне успадкування також означає, що похідний клас **grad_student** є *підтипом* типу **student**. Іншими словами, студент, який закінчив інститут, також студент.

Похідний клас – це модифікація базового класу, яка успадковує загальні та захищені елементи базового класу. Таким чином, в останньому прикладі до класу **grad_student** з класу **student** успадковані елементи **id**, **name**, **print** з класу **student**. Функцію-метод **print** *перекрито*. Це означає, що похідний клас отримує реалізацію функції-елемента, яка відрізняється від аналогічної функції базового класу. *Перекриття* не слід плутати із *перевантаженням*, де одне й теж саме ім'я функції може мати різні значення для кожної унікальної *сигнатури*.

Такому механізму притаманні наступні переваги:

– код базового класу використовується багаторазово;

– тип **grad_student** використовує вже існуючий перевірений код зі **student**;

– ієрархія відображає відношення, які існують у предметній області;

– поліморфні механізми дозволяють коду користувача обробляти **grad_student** як підтип **student**, що спрощує код користувача, в той же час надаючи йому переваги при обробці відмінностей типів.

Правила доступу для об'єктів і класів

Елементи, які оголошено **private** (закритими), доступні функціям-методам і друзям класу.

Елементи, які оголошено **protected** (захищеними), доступні функціям-методам, друзям класу і функціям-методам похідних класів.

Елементи, які оголошено **public** (відкритими), доступні будь-яким функціям.

Коли базовий клас використовується як відкритий (**public**) базовий клас (***зовнішнє успадкування***), його відкриті елементи залишаються відкритими

елементами похідного класу, а захищені елементи – захищеними елементами похідного класу.

Коли базовий клас використовується як закритий (**private**) базовий клас (***внутрішнє успадкування***), його відкриті і захищені елементи стають закритими елементами похідного класу.

Коли базовий клас використовується як захищений (**protected**) базовий клас (***захищене успадкування***), його відкриті і захищені елементи стають захищеними елементами похідного класу.

Фактично, при *захищеному* та *внутрішньому* успадкуванні похідний клас виключає зі свого інтерфейсу інтерфейс базового класу, але сам може їм користуватися. В цьому вони схожі. Різницю між захищеним та внутрішнім успадкуванням «відчує» лише клас, який породжується з похідного. Читачеві пропонується самостійно довести цей факт.

Захищені елементи класу забезпечують доступ і захист

Як вже було сказано, програма не може звертатися безпосередньо до *приватних* (закритих) елементів класу. Для того, щоб звернутися до приватних елементів програма повинна використовувати інтерфейсні функції, які керують доступом до цих елементів. Використання для цих цілей дружніх функцій порушує інкапсуляцію і може вважатися відхиленням від основних принципів ООП. Успадкування спрощує програмування лише тоді, коли похідні класи можуть звертатися до елементів базового класу за допомогою операції '**крапка**'. В таких випадках програми можуть використовувати захищені елементи класу. Похідний клас може звертатися до захищених елементів базового класу безпосередньо із використанням операції '**крапка**'. Проте решта програми може звертатися до захищених елементів тільки за допомогою інтерфейсних функцій цього класу. Таким чином, захищені елементи класу знаходяться між загальними (доступними всій програмі) і приватними (доступними тільки самому класу) елементами.

Розрішення конфлікту імен

При породженні певного класу з іншого класу шляхом успадкування можливі ситуації, коли ім’я елемента в похідному класі співпадає з ім'ям елемента базовому класі. При виникненні такого конфлікту імен, компілятор

C++ завжди використовує елементи похідного класу всередині функцій похідного класу. Наприклад, класи **student** і **grad_student** використовують метод **print**. Якщо в тексті програми не вказано явно (за допомогою оператора глобального розрішення '**::**'), то функції класу **grad_student** будуть завжди використовувати елементи свого, тобто похідного, класу **grad_student**. Але, якщо функціям класу **grad_student** необхідно звертатися до методу **print** базового класу **student**, то вони повинні використовувати ім'я класу **student** і оператор глобального розрішення '**::**', наприклад, **student::print**. Припустимо, що в класі **grad_student** необхідно записати виклик обох методів **print**. Тоді необхідно використовувати наступні оператори:

```
print();                    // виклик методу класу grad_student
student :: print();         // виклик методу класу student
```

Висновок

Успадкування – це здатність *похідного* класу *успадковувати* характеристики існуючого *базового* класу. Це означає: якщо існує клас, елементи даних або функції якого можуть бути корисними для нового класу, можна налаштувати цей новий клас в термінах існуючого (базового) класу. Новий клас у свою чергу буде успадковувати елементи існуючого класу. Використання успадкування для побудови нових класів заощаджує значний час і сили на програмування. Об'єктно-орієнтоване програмування використовує успадкування, що дозволяє будувати складні об'єкти з невеликих об'єктів, якими легко управляти.

Приклад 2.15:

```
class A
{       int i,k;
  public:
        A(void) { i=0; k=0; }
        A(int i1,int k1=0): i(i1),k(k1) {}
        int geti() { return i; }
        void print() { cout << " A "<< i << " "<<k; }        };

class B: public A
{       float s;
  public:
        B(void): A() { s=0.; }
        B(int i1,int k1=0,float s1=0.): A(i1,k1),s(s1) {}
        float gets() { return s; }
```

```
        void print(){A::print(); cout << " s="<< s; }                    };

class C: private A
{ public:
        C(void):A() {}
        C(int i1,int k1=0):A(i1,k1) { }
        int geti() { return A::geti(); }
        void print() { A::print(); }                };

void main()
{       A a1, a2(10);
        B b1,b2(20,30,40);
        C c1,c2(50,60);       }
```

Підсумки

Успадкування в C++ дозволяє будувати/породжувати новий клас із існуючого класу. Будуючи у такий спосіб один клас з іншого, можна зменшити об'єм коду і час на створення програми. Далі буде показано, що C++ дозволяє породжувати клас з двох та більшої кількості базових класів. Використання декількох базових класів для породження класу має назву «множинне успадкування». Підсумком даного підрозділу має бути засвоєння викладених нижче концепцій.

1. *Успадкування* – це створення (породження) нового похідного класу з існуючого базового класу.

2. *Похідний* клас – це новий клас, а базовий клас – це клас, який вже існував до створення похідного класу.

3. При породженні одного (*похідного*) класу з іншого (*базового*) класу похідний клас *успадковує* елементи базового класу.

4. Рядок-заголовок оголошення *похідного* класу в C++ починається ключовим словом **class**, за яким треба вказати ім'я *похідного* класу, двокрапку та ім'я базового класу. Наприклад, **class grad_student : public student**.

5. Коли клас породжується з *базового* класу, *похідний* клас може звертатися до *загальних* елементів *базового* класу так, ніби ці елементи визначено в самому *похідному* класі. Для доступу до *приватних* даних *базового* класу *похідний* клас повинен використовувати *інтерфейсні* функції *базового* класу (або *дружні* функції *базового* класу).

6. Конструктор *похідного* класу повинен викликати конструктор *базового* класу, як елемент *списку ініціалізації*, вказуючи через двокрапку '**:**' ім'я конструктора *базового* класу із відповідними параметрами.

7. Щоб забезпечити *похідним* класам безпосередній доступ до певних елементів *базового* класу, одночасно захищаючи ці елементи від решти програми, в C++ використовуються *захищені* (**protected**) елементи класу. *Похідний* клас може звертатися до *захищених* елементів *базового* класу, ніби вони є *загальними*. Проте, для решти програми *захищені* елементи еквівалентні *приватним*.

8. Якщо у *похідному* і *базовому* класах оголошено елементи з однаковими іменами, то в межах функцій *похідного* класу C++ застосовуватиме елементи *похідного* класу. Якщо функціям *похідного* класу необхідно звернутися до елементів *базового* класу, треба використовувати *оператор глобального розрірення* '**::**', наприклад, **base_class :: member**.

2.4. Віртуальні функції і поліморфізм

Коли програмісти говорять про C++ і об'єктно-орієнтоване програмування, то дуже часто використовують термін *поліморфізм*. У загальному випадку ***поліморфізм*** є здатністю об'єкта змінювати форму. Якщо розділити цей термін на частини, то можна виявити, що «*полі*» означає багато, а «*морфізм*» відноситься до зміни форми. *Поліморфний об'єкт*, отже, є об'єктом, який може приймати різні форми.

Поліморфізм – це надання різних значень одному і тому ж повідомленню залежно від типу оброблюваних даних. *Перетворення* – це явна або неявна зміна значення залежно від типу. Перетворення забезпечує форму поліморфізму. Перевантаження функцій дає одному і тому ж імені функції різні значення. Одне і те ж ім'я має різні інтерпретації, які залежать від вибору функції. Вибрана функція задовольняє алгоритму відповідності сигнатур для C++. Така форма поліморфізму називається ***спеціальний поліморфізм***.

У цьому розділі розширюється поняття поліморфізму і розглядається, як використовувати поліморфні об'єкти в програмах для спрощення і зменшення коду. Будуть розглянуті наступні основні концепції:

– *поліморфізм* є здатністю об'єкта змінювати форму під час виконання програми;

– для створення *поліморфних об'єктів* в програмі повинні використовуватися *віртуальні* (**virtual**) *функції*;

– *віртуальна функція* – це функція базового класу, перед ім'ям якої стоїть ключове слово **virtual**;

– будь-який похідний від базового клас може використовувати або перевантажувати *віртуальні функції*;

– для доступу до *поліморфного об'єкта* слід використовувати покажчик на об'єкт базового класу.

2.4.1. Віртуальні функції і поліморфічні кластери

Зв’язування повідомлення з методом

Одне з пояснень терміну *поліморфічний* в словнику визначено, як такий, що «має, приймає або зустрічається в різних формах, символах або стилях». У застосуванні до об'єктно-орієнтованих мов, *поліморфізм* розглядається як істотна

властивість, що дозволяє деяке повідомлення передавати різними шляхами. Отже, повідомлення може мати безліч різних реалізацій.

Як правило, зв'язування повідомлення, що посилається об'єкту, з конкретним методом (функцією-методом) здійснювалося на етапі компіляції (тобто до запуску програми на виконання). Таке *раннє зв'язування* у багатьох випадках бажане, оскільки це дозволяє отримати найбільш оптимальний код. Окрім раннього зв'язування компілятор C++ виконує перевірку типів, гарантуючи, що кожному об'єкту буде надіслане законне повідомлення. Об'єктно-орієнтована мова пропонує програмістові вибір між перевіркою типів і *пізнім зв'язуванням* (зв’язуванням при виконанні програми).

Пізнє зв'язування дозволяє асоціювати повідомлення з методом під час виконання програми. Програміст визначає специфічні дії, які повинен виконувати об'єкт, отримавши повідомлення. Під час виконання, програма інтерпретує ці дії і зв'язує повідомлення з відповідним методом. Перегляд списку методів і вибір потрібного покладається не на програміста, а на програму. Це дозволяє писати більш надійний код і полегшує його використання і модифікацію.

Рисунок 2.2 – Зв’язування при компіляції і при виконанні

Віртуальні функції і поліморфічні кластери

Часто виникає необхідність передачі повідомлень об'єктам, що належать різним класам в ієрархії. Кожен клас в ієрархії відповідає на повідомлення по-своєму. В цьому випадку потрібна реалізація механізму *зв'язування при виконанні* програми. У C++ цей механізм реалізований для *віртуальних функцій* (оголошених з використанням ключового слова **virtual**).

У C++ зв'язування при виконанні охоплює підмножину функцій-методів класу, які називають ***віртуальними функціями***. *Віртуальна функція* оголошується в базовому або в похідному класі і, потім, перевизначається в

успадкованих класах. Сукупність класів (підкласів), в яких визначається і перевизначається віртуальна функція, називається ***поліморфічним кластером***, який асоціюється з деякою віртуальною функцією. В межах поліморфічного кластеру повідомлення зв'язується з конкретною віртуальною функцією-методом під час виконання програми.

Звичайну функцію-метод базового класу можна перевизначити в похідних класах. Проте без атрибуту **virtual** така функція буде зв'язана з повідомленням на етапі компіляції. Атрибут **virtual** гарантує зв'язування при виконанні в межах поліморфічного кластера.

Щоб добитися зв'язування при виконанні для деякого об'єкта, треба звертатися до його (об'єкта) віртуальних методів, через *покажчик* або *посилання* на його базовий клас. При *зовнішньому успадкуванні* покажчики і посилання на об'єкти похідних класів сумісні з покажчиками і посиланнями на об'єкти базового класу (тобто до об'єкту похідного класу можна звертатися, наче це об'єкт базового класу). Вибрана функція-метод залежить від класу, на об'єкт якого вказується, але не від типу покажчика.

C++ підтримує віртуальні (**virtual**) функції, які оголошені в базовому класі і перевизначені в похідному класі. Ієрархія класів, що утворена зовнішнім успадкуванням, утворює зв'язаний набір типів користувача, на які можна посилатися за допомогою покажчика базового класу. При звертанні до віртуальної функції через цей покажчик в C++ вибирається відповідне визначення функції під час виконання програми. Об'єкт, на який вказує покажчик, повинен містити в собі інформацію про тип, оскільки відмінності між об’єктами перевіряються динамічно. Це особливість типова для ООП коду. Кожен об'єкт «знає», як його необхідно викликати. Ця форма поліморфізму називається ***чистим поліморфізмом***.

Віртуальні функції дозволяють похідним класам визначати власні версії функцій базового класу. У C++ не віртуальні функції-методи класу з різною кількістю і типами параметрів є дійсно різними функціями, навіть якщо вони мають одне і теж ім'я. *Віртуальні функції* дозволяють перевизначати в похідному класі функції, які містяться в базовому класі, навіть якщо кількість і типи аргументів співпадають. Для віртуальних функцій заборонено перевизначати тип значення, що повертається цією функцією. Якщо дві функції з однаковим ім'ям матимуть різні *сигнатури* (кількість і типи параметрів), C++ вважатиме їх різними і проігнорує механізм віртуальних функцій. Для визначення *віртуальної*

функції служить ключове слово **virtual**. Віртуальна функція – обов'язково метод класу (не друг).

Ключове слово **virtual** потрібне в поліморфічному кластері тільки перед віртуальною функцією самого верхнього рівня. Для усіх інших перевизначень цієї функції ключове слово **virtual** необов'язкове:

```
class B
{ public:
    virtual void vf1();
    virtual void vf2();
    virtual void vf3();
    void f();     };

class D : public B
{ public:
    virtual void vf1();  // віртуальна функція
    void vf2(int);         // не віртуальна функція, інший список аргументів
    // char vf3();         // неприпустимо - змінено тип, помилка
    void f();     };

void main()
{    D d;
    B *bp=&d;          // покажчику на базовий клас привласнюється
                       // покажчик на об’єкт похідного класу
    bp -> vf1();       // виклик D::vf1
    bp -> vf2();       // виклик B::vf2
    bp -> f();    }    // виклик B::f
```

Механізм створення поліморфічного кластера віртуальної функції можна продемонструвати наступним прикладом.

Приклад 2.16:

```
class Parent
{ protected:
    char *lastName;
  public:
    Parent(void) { lastName = new char[5]; strcpy(lastName, "None"); }
    Parent(char *a)
        { lastName = new char[strlen(a)+1]; strcpy(lastName, a); }
    Parent(Parent &a)
        { lastName=new char[strlen(a.lastName)+1];
          strcpy(lastName,a.lastName); }
    char *getLastName(void) { return lastName; }
    void setLastName(char *a)
        { lastName=new char[strlen(a)+1]; strcpy(lastName,a); }
    virtual void answerName(void)
```

```
        { cout << "My last name is" << lastName << "\n"; }
    ~Parent(void) { delete lastName; }     };

class Child: public Parent
{ protected:
    char *firstName;
  public:
    Child(void) { firstName=new char[5]; strcpy(firstName, "None"); }
    Child(char *a,char *a1) : Parent(a)
        { firstName=new char[strlen(a1)+1]; strcpy(firstName,a1);    }
    Child(Child &a): Parent(a)
        { setLastName(a.getLastName());
          firstName=new char[strlen(a.firstName)+1];
          strcpy(firstName,a.firstName);   }
    char *getFirstName(void) { return firstName; }
    void setFirstName(char *a)
        { firstName=new char[strlen(a)+1]; strcpy(firstName,a);  }
    ~Child(void) { delete firstName; }
    virtual void answerName(void)
        { Parent::answerName();
          cout << "My first name is " << firstName << "\n"; }             };

class GrandChild: public Child
{ private:
    char *grandFatherName;
  public:
    GrandChild(char*a,char*a1,char*a2): Child(a,a1)
        { grandFatherName=new char[strlen(a2)+1];
          strcpy(grandFatherName,a2); }
    ~GrandChild(void) { delete grandFatherName; }
    virtual void answerName(void)
    { Child:: answerName
         ();
          cout << "My grandfather's name is " <<
          grandFatherName << "\n"; }       };

void main()
{   Parent *family[3];          // масив покажчиків на базовий клас
    Parent *p=new Parent((char *)"Іванов");
    Child *c=new Child((char *)"Іванов", (char *)"Сергій");
    GrandChild *g =
        new GrandChild((char *)"Іванов", (char *)"Андрій", (char *)"Володимир");
    family[0]=p;                // в масив покажчиків
    family[1]=c;                // на базовий клас
    family[2]=g;                // записуються покажчики спадкоємців
    for(int index=0; index<3; index++)
        family[index] -> answerName();   }
```

Ефект від віртуалізації можна краще зрозуміти, якщо прибрати у цій програмі ключове слово **virtual**. Тобто функції перестають бути віртуальним.

Результат роботи програми:

При оголошенні, як **virtual**	Без **virtual**
My last name is Іванов My last name is Іванов My first name is Сергій My last name is Іванов My first name is Андрій My grandfather's name is Володимир	My last name is Іванов My last name is Іванов My last name is Іванов

Для реалізації поліморфізму можливо використовувати *посилання* на об'єкти замість *покажчиків*.

```
void main()
{    Parent p((char *)"Іванов");
     Child c((char *)"Іванов", (char *)"Сергій");
     GrandChild g((char *)"Іванов", (char *)"Андрій", (char *)"Володимир");
     Parent &f0=p, &f1=c, &f2=g;
     f0.answerName(); f1.answerName(); f2.answerName();  }
```

Деякі тонкощі демонструє наступний приклад:

Приклад 2.17:

```
class P
{ private:
    virtual void method1(void) { }
    void method2(void){ }
    void method3(void){method1(); method2();} // this->method1,
                                               // this->method2
    void method5(void){ }
  public:
    void method6(void){method3();}
    void method4(void){ }
    void method7(void){method5(); }                };

class C : public P
{ private:
    void method1(void) { }
    void method5(void) { }
  public:
    C(){}
    void method4(void) { }
    void method7(void) { method5(); }      };

void main()
```

```
{   P pop;
    C me;
    pop.method6();   // P::method6  P::method3
                     // P::method1  P::method2
    me.method6();    // P::method6  P::method3
                     // C::method1  P::method2
    pop.method4();   // P::method4
    me.method4();    // C::method4
    pop.method7();   // P::method7  P::method5
    me.method7(); }  // C::method7  C::method5
```

Послідовність виклику методів вказана у коментарях. Треба звернути увагу на рядок, в якому повідомлення **method6()** надсилається об'єкту **me** (**me.method6();**). Відповідна функція **method6()** успадкована з класу **P**. Це призводить до виклику приватного методу **method3()** і, далі, приватних методів **method1()** та **method2()**. Оскільки **method1()** оголошено в базовому класі **P**, як віртуальний метод, система на етапі виконання програми зв'язує повідомлення **this->method1()** із **method1()** класу **C**.

Віртуальними (**virtual**) можуть бути тільки нестатичні функції-методи класу. Функція породженого класу автоматично стає віртуальною. Спеціальним обмеженням є те, що **деструктори** можуть бути віртуальними, а **конструктори** – ні.

Наступний приклад демонструє переваги використання віртуальних функцій для спрощення програми:

```
class shape
{ protected:
        double x,y;
  public:
        virtual double area() { return 0.0; }   };

class rectangle : public shape
{       double height, width;
  public:
        double area() { return (height*width); }  };

class circle : public shape
{       double radius;
  public:
        double area() { return (PI*radius*radius); }  };
```

Обчислення площини є локальною відповідальністю похідного класу. Код користувача, в якому використовується поліморфне обчислення площини, може бути таким:

```
shape *p[N];
.....
for(i=0; i<N; i++) tot_area+=p[i]->area();
```

В цьому прикладі головною перевагою є те, що код користувача можна не змінювати, навіть тоді, коли до системи додаються нові геометричні фігури. Управління змінами є локальним і розповсюджується автоматично, завдяки поліморфному характерові коду користувача.

Висновок

Поліморфний об’єкт – це такий об’єкт, який може змінювати форму під час виконання програми. Для створення поліморфного об’єкта в програмі необхідно використовувати покажчик на об’єкт базового класу. Цьому покажчикові на базовий клас необхідно присвоїти адресу об’єкта похідного класу. Кожен раз, коли у програмі покажчику присвоюється адреса іншого класу, об’єкт цього покажчика (який є поліморфним) змінює форму. Програми будують поліморфні об’єкти, базуючись на віртуальних функціях базового класу.

2.4.2. Чисті віртуальні функції та абстрактні базові класи

Базовий клас у ієрархії типу зазвичай містить низку віртуальних функцій, які забезпечують динамічну типізацію. Часто в базовому класі ці віртуальні функції фіктивні і мають пусте тіло. Певне значення їм надається тільки в породжених класах. В C++ для цієї мети застосовуються чисті віртуальні функції. ***Чиста віртуальна функція*** – віртуальна функція-метод класу, тіло якої не визначено. Запис такого визначення у класі наступна:

```
virtual <прототип функції> = 0;
```

Чиста віртуальна функція використовується для того, щоб «відкласти» рішення щодо реалізації функції. У ООП термінології це називається ***відстроченим (відтермінованим) методом***.

Клас, в якому міститься щонайменше одна чиста віртуальна функція – це ***абстрактний клас***. Для ієрархії типів корисно мати базовий абстрактний клас. Він містить загальні спільні властивості похідних класів, проте сам не може бути використаний для оголошення об'єктів. Він може використовуватися лише для оголошення покажчиків, які можуть звертатися до об'єктів підтипу, які породжено від *абстрактного базового класу*.

Такий клас може існувати з єдиною метою – бути батьківським по відношенню до похідних класів, об'єкти яких будуть реалізовані. Також він може слугувати ланкою для створення ієрархічної структури класів. Сказане можна продемонструвати наступним прикладом.

Приклад 2.18:

```
#include <iostream>
using namespace std;
class Base                          // абстрактний базовий клас
{ public:
    virtual void show() = 0;  };    // чиста віртуальна функція

class Derv1 : public Base           // похідний клас 1
{ public:
    void show() { cout << "Derv1\n"; }          };

class Derv2 : public Base           // похідний клас 2
{ public:
    void show() { cout << "Derv2\n"; }          };

void main()
{       //      Base bad;           // помилка!!! неможливо створити об'єкт
    Base* arr[2];                   // масив покажчиків на базовий клас
    Derv1 dv1;                      // об'єкт похідного класу 1
    Derv2 dv2;                      // об'єкт похідного класу 2

    arr[0] = &dv1;                  // запис адреси dv1 в масив
    arr[1] = &dv2;                  // запис адреси dv2 в масив

    arr[0]->show();                 // виконання show() з обох об'єктів
    arr[1]->show();         }
```

В цьому прикладі віртуальну функцію **show()** оголошено так:

```
    virtual void show() = 0;        // чиста віртуальна функція
```

Знак рівняння не має нічого спільного з операцією привласнення. Нульове значення для привласнення не використовується. Конструкція '=0' – це просто спосіб повідомити компілятору, що функція буде *чистою віртуальною*. Якщо тепер в **main()** спробувати створити об'єкт класу **Base**, то компілятор виведе повідомлення про помилку.

Якщо клас використовує чисту віртуальну функцію, він сам стає *абстрактним* і ніякі об'єкти з нього створити неможливо. Можна всі віртуальні функції базового класу зробити чистими. Такий клас буде описувати *інтерфейс* класу.

Переваги зв'язування при виконанні програми

В C++ переваги від різного реагування об'єкта на повідомлення можливо отримати в поліморфічному кластері. Можливо створити колекцію посилань або покажчиків на об'єкти. Ці посилання (покажчики) повинні бути оголошеними як посилання або покажчики на об'єкти базового класу або одного з похідних класів в поліморфічному кластері. Будь-яка спроба завантажити покажчик або посилання на об'єкт, що не входить у поліморфічний кластер, закінчиться невдачею (компілятор видасть повідомлення про невідповідність типів). Посилання і покажчики на об'єкти базового класу сумісні тільки із посиланнями або покажчиками **public**-похідного класу.

В мовах, які підтримують тільки зв'язування при компіляції, відповідальність за виклик відповідної версії функції покладено на програміста. Зазвичай для виконання такого контролю використовуються оператори множинного вибору, такі як оператори **switch**, **if else**. Кожен додатковий вибір, який вводиться у систему, необхідно включати у всі оператори множинного вибору. При цьому програма змінюється в багатьох місцях.

В мовах, які підтримують зв'язування при виконанні програми, (наприклад C++), при необхідності внести доповнення в програму, створюються нові похідні класи. Додаткові можливості реалізуються через віртуальні функції цих класів. Така можливість дозволяє мінімізувати модифікацію програми.

Віртуальні деструктори

C++ дозволяє оголошувати віртуальні деструктори, як звичайні функції-методи. Тоді деструктори всіх класів, що є похідними від класу, в якому оголошено віртуальний деструктор, також будуть віртуальними.

Використання віртуальних деструкторів дозволяє забезпечити виклик відповідного деструктора при знищенні об'єктів оператором **delete**, навіть якщо тип об'єкта, що знищується, невідомий на етапі компіляції. Це показано у наступному прикладі.

Приклад 2.19: Віртуальні деструктори

```
class Figure
{ public:
     Figure(){};
     virtual ~Figure(){};  };

typedef Figure *PFigure;

class Circle : public Figure
{ public:
     Circle(int centerx, int centery, int radius);
     virtual ~Circle(){};        };

class Rectangle : public Figure
{ public:
     Rectangle(int left, int top, int right, int bottom);
     ~Rectangle(){};             };

int main();
{       const ALLFigures = 2;
     PFigures figures[ALLFigures];          // масив покажчиків на фігури
     figures[0]=new Circle(100,100,10);
     figures[1]=new Rectangle(100,100,200,300);
     for(int count=0; count < ALLFigures; count++)
        delete figures[count];   }              // знищення об'єктів (фігур)
```

Підсумок

Поліморфізм – це здатність об'єкта змінювати форму під час виконання програми. Розглянуто послідовність дій, які необхідно виконати для створення поліморфних об'єктів. Основні концепції, які викладено, перелічено нижче.

1. Поліморфний об'єкт здатний змінювати форму під час виконання програми.

2. Поліморфні об’єкти створюються з класів, які є похідними від існуючого базового класу.

3. В базовому для поліморфного об’єкта класі необхідно визначити одну або декілька функцій як віртуальні (**virtual**).

4. Поліморфні об’єкти характеризуються використанням віртуальних функцій базового класу.

5. Для створення поліморфного об’єкта необхідно створити покажчик або посилання на об’єкт базового класу.

6. Щоб змінити форму поліморфного об’єкта, слід привласнити покажчику/посиланню на базови клас адреси поліморфного об’єкта похідного класу.

7. Чиста віртуальна функція – це віртуальна функція базового класу, для якої в базовому класі не визначені оператори. Замість них базовий клас привласнює такій функції значення **0**.

8. Похідні класи повинні забезпечити визначення функції для кожної чистої віртуальної функції базового класу.

2.4.3. Технічна реалізація віртуальних функцій

Перевірка помилок при використанні віртуальних функцій

В деяких об’єктно-орієнтованих мовах, які підтримують зв’язування на етапі виконання програми, перевірка помилкових повідомлень виконується саме при виконанні програми. Мова C++, завдяки віртуальним функціям, підтримує і статичний (при компіляції) контроль повідомлень і контроль зв’язування під час виконанні програми.

Приклад 2.20: Помилка при виклику віртуальної функції

```
class P
{ public:
      P(){}
      virtual void hello() {}   };

class C: public P
{ public:
      C() {}
      virtual void hello() { cout << "Hello world"; }        };

void main()
```

```
{       P *p; C c;
        p=&c;
        p->hello("Hello"); }              // Помилка!
```

Компілятор виведе повідомлення про помилку в рядку **p->hello("Hello")**. Компілятор може визначити, що віртуальна функція **hello** не має параметрів, навіть якщо повідомлення зв'язується із методом с методом **hello** класу **C** під час виконання програми. Віртуальні функції, які визначені у батьківському чи похідних класах, повинні мати однакові списки параметрів, тому що пізнє зв'язування не впливає на контроль типів параметрів.

Технічна реалізація віртуальних функцій

Об'єкт C++ при виконання програми займає безперервну ділянку оперативної пам'яті. Покажчик на об'єкт містить адресу початку цієї ділянки. При створенні об'єктів похідних класів їхні поля додаються до полів батьківських класів. Цю функцію виконує компілятор.

Коли викликається функція-метод класу (об'єкту надсилається повідомлення), цей виклик транслюється у звичайний виклик функції із *додатковим аргументом*, який містить *покажчик на об'єкт*.

Наприклад:

```
ClassName *object;
object->message(10);
```

перетворюється у

```
ClassName::message(object,10);
```

де **object** – додатковий аргумент, який неявно присутній у виклику функції **message(10)**.

Віртуальні функції реалізовано із використанням *таблиць віртуальних функцій*. Це може бути пояснено на наступному фрагменті коду.

```
class P
{       int value;
  public:
      virtual int method1(float r):
      virtual void method2(void);
      virtual float method3(char *s);  };
```

```
class C1 : public P
{ public:
     void method2(void);  };

class C2 : public C1
{ public:
     float method3(char *s);  };
```

Таблиця віртуальних функцій (**virtual_tabl**), будується для кожного об’єкта поліморфічного кластеру. Таблиця віртуальних функцій містить функції-методи кожного класу в поліморфічному кластері. Покажчик на цю таблицю мають всі об’єкти класів і підкласів поліморфічного кластеру.

Типовий об’єкт наведеного вище класу **P** виглядає, приблизно, так:

```
int value;
virtual_table * ->     P::method1 // адреса virtual_tabl
                       P::method2
                       P::method3
```

Типовий об’єкт класу **C1** виглядає, приблизно, так:

```
int value;
virtual_table * ->     P::method1
                       C1::method2
                       P::method3
```

Типовий об’єкт класу **C2**, виглядає приблизно, так:

```
int value;
virtual_table * ->     P::method1
                       C1::method2
                       C2::method3
```

Компілятор перетворює виклик віртуальної функції в непрямий виклик через **virtual_tabl**. Наприклад, виклик метода **method3** у фрагменті

```
P *p;
C2  *c = new C2;
p = c;
p->method3(“Hello”);
```

перетворюється у

```
(*(c->virtual_table[2]))(c,"Hello"); // виклик функції, адреса якої
                                     // у третьому рядку таблиці virtual_tabl
```

Якщо б методи не було визначено, як віртуальні, то рішення про те, який метод буде викликано, приймалося б статично, тобто при трансляції. При цьому, оскільки виклики виконуються через покажчики на базовий клас, було б викликано методи базового класу. При визначенні методів, як **virtual**, викликаються перевизначені методи, тобто динамічно.

2.5. Множинне успадкування

2.5.1. Конфлікт імен при множинному успадкуванні

Раніше було показано, як можливо побудувати якийсь клас шляхом успадкування характеристик іншого класу. Мова програмування C++ також дозволяє породжувати клас з декількох базових класів. Коли клас успадковує характеристики декількох класів, використовується ***множинне успадкування***. Мова C++ повністю підтримує *множинне успадкування*. Далі перелічено основні положення, які будуть розглянуто:

– якщо клас породжується з декількох базових класів, то таке успадкування називається ***множинним успадкуванням***;

– при множинному успадкуванні похідний клас отримує атрибути і методи двох або більшої кількості базових класів;

– при використанні множинного успадкування для породження класу конструктор похідного класу повинен викликати конструктори усіх базових класів;

– при породженні класу з похідного класу утворюється *ієрархія успадкування* (*ієрархія класів*).

Множинне успадкування являє собою потужний інструмент об’єктно-орієнтованого програмування. При побудові класу з декількох вже існуючих базових класів, можна значно заощадити час та зусилля на програмування.

Похідні класи: множинне успадкування

Ієрархію успадкування можна описати із використанням структури дерева, в якому кожен вузол представляє підклас, який здатен породжувати будь-яку кількість додаткових підкласів. Як і у випадку простого успадкування, визначники **private:**, **protected:**, **public:** в батьківських класах можна використовувати для управління доступом до екземплярів змінних та методів, які успадковані похідним класом (підкласом). Крім цього, специфікатори **public**, **protected** або **private** в заголовку оголошення похідного класу, як і при простому успадкуванні, визначають похідний клас, чиї об’єкти мають доступ до відкритих та захищених даних і функцій-методів базових класів.

Для наочного представлення ієрархії множинного успадкування можна застосувати *прямий ациклічний граф* (ПАГ) (орієнтований граф успадкування без петель). Підклас може успадковувати протокол одного або більшої кількості

батьківських класів. При цьому, крім специфікаторів **public**, **protected** та **private** в заголовках похідних класів, може використовуватися додаткова опція **virtual**.

Наступний приклад демонструє множинне успадкування. Клас **Derived** оголошується як похідний від двох базових класів **Base1** и **Base2**.

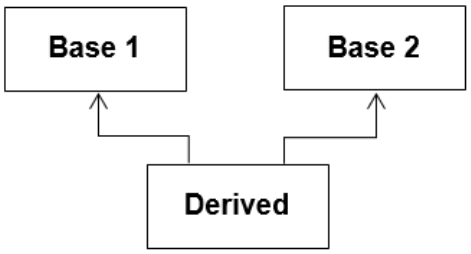


Рисунок 2.3 – Прямий ациклічний граф множинного успадкування

Приклад 2.21:

```
class Base1
{     int id;
  public:
    Base1() { cout << " Constructor Base1"; id=0; }
    Base1(int anid) { cout << " Constructor Base1"; id=anid; }
    void assignid(int anid) { id=anid; }
    int accessid() { return id; }        };

class Base2
{    char name[20];
  public:
    Base2() { cout << " Constructor Base2"; strcpy(name, "void"); }
    Base2(char *str) { cout << " Constructor Base2"; strcpy(name, str); }
    void assignName(char *str) { strcpy(name, str); }
    char *accessName() { return name; } };

class Derived: public Base1, public Base2
{    char ch;
  public:
    Derived() { cout << " Construct Derived"; ch='a'; }
    Derived(char c, int anid, char *str) : Base1(anid), Base2(str)
          { cout << " Construct Derrived"; ch=c; }
    void assign(char c) { ch=c; }
    friend void show(Derived& d); };

void show(Derived& d)
    { cout << " id " << d.accessid() << " name " <<
      d.accessName()  << " ch " << d.ch << endl;          }

void main()
```

```
{    Derived object1;                    show(object1);
   Derived object2('e', 26, "Robert: ");    show(object2);                }
```

При множинному успадкуванні базові класи записуються в заголовку похідного класу через кому із зазначенням типу успадкування (зовнішнє – **public**, внутрішнє – **private**, захищене – **protected**):

```
class Derived: public Base1, public Base2 { … };
```

В результаті роботи програми на екран буде виведено:

```
Constructor Base1 Constructor Base2 Construct Derived id 0 name void ch a
Constructor Base1 Constructor Base2 Construct Derived id 26 name Robert ch e
```

Коли **object1** оголошено за допомогою виразу **Derived object1;**, викликаються **void**-конструктори двох базових класів в тій послідовності, у якій визначено успадкування (**Base1**-конструктор перший, **Base2**-конструктор другий).

Конфлікт імен

Не можна виключити таку ситуацію, коли один або декілька базових класів містять елементи з однаковими іменами. Наприклад, що буде, якщо екземпляри змінних **name** та **id** в базових класах **Base1** і **Base2** перейменувати у **instance** і перенести до **protected** частини оголошень класів? З’явиться неоднозначність у визначенні функції **show(Derived& d)**. Компілятор не зможе відрізнити **d.instnace** із класу **Base1** від **d.instance** із класу **Base2**. Тоді як при зверненні через інтерфейсні функції все буде працювати нормально.

Якщо дані описано як **private** і, якщо для доступу до них використовуються функції-методи класу, то програма буде працювати, тому що протокол класу **Derived** може мати доступ до екземплярів змінних двох базових класів, тільки шляхом використання методів доступу **accessid**, **accessName**. Тільки при спробі звернутися до екземплярів безпосередньо, як до власних, виникне помилка внаслідок конфлікту імен.

У наступному прикладі два базових класи містять поля з однаковими іменами.

Приклад 2.22:

```
class Base1
{ protected:
    int id;
  public:
    Base1() { id=0; }
    Base1(int anid) { id=anid; }      };

class Base2
{ protected:
    int id;
  public:
    Base2() { id=0; }
    Base2(int anid) { id=anid; }      };

class Derived: public Base1, public Base2
{    char ch;
  public:
    Derived() { ch='a'; }
    Derived(char c, int anid, int aid):Base1(anid), Base2(aid)
          { ch=c; }
    void assign(char c) { ch=c; }
    friend void show(Derived& d);           };

void show(Derived& d)
{    cout << "id Base1 " << d.Base1::id;
    cout << " id Base2 " << d.Base2::id;
    cout << " ch " << d.ch << endl;          }

void main()
{    Derived object1;          show(object1);
     Derived object2('e',120,-150); show(object2);                }
```

Результати роботи програми:

```
id Base1 0 id Base2 0 ch a
id Base2 120 id Base2 -150 ch e
```

Об’єкти класу **Derived** містять по два екземпляри змінної **id** кожний. Одна **id** з класу **Base1**, а інша – **id** з класу **Base2**. Для доступу до змінної **id** використовується *дружня* (**friend**) по відношенню до класу **Derived** функція **show()**. Функція отримує через список параметрів посилання на об’єкт класу **Derived** і використовує його для доступу до екземплярів змінної **id** із використанням операції '**крапка**' і *кваліфікатора області бачення* '**::**' **d.Base1::id**.

Наступний приклад демонструє, що доступ до змінної **id** в ***функції-методі*** класу може бути виконано через **Base1::id** або **Base2::id**.

Приклад 2.23:

```
class Base1
{ protected:
      int id;
  public:
      Base1() { id=0; }
      Base1(int anid) { id=anid; }              };

class Base2
{ protected:
      int id;
  public:
      Base2() { id=0; }
      Base2(int anid) { id=anid; }              };

class Derived: public Base1, public Base2
{       char ch;
  public:
      Derived() { ch='a'; }
      Derived(char c, int anid, int aid):Base1(anid), Base2(aid)
                  { ch=c; }
      void assign(char c) { ch=c; }
      void show();                };

void Derived::show()
      {   cout << "id Base1 " << Base1::id ;
          cout << " id Base2 " << Base2::id ;
          cout << " ch" << ch << endl;        }

void main()
{       Derived object1;                      object1.show();
      Derived object2('e',120,-150);          object2.show();               }
```

Таким чином, доступ до тієї чи іншої змінної в *функції-методі* класу може бути виконано через кваліфікатор **Base1::id** або **Base2::id**, а в *дружній функції* через посилання (або покажчик) на об’єкт **d.Base1::id** або **d.Base2::id**.

Ще один приклад демонструє можливу помилку звернення при наявності конфлікту імен.

Приклад 2.24:

```
class Base1
{ public:
```

```
        int cost();      };

class Base2
{ public:
        int cost();      };

class Derived: public Base1, public Base2
{ public:
        int tot_cost() { return (Base1::cost()+Base2::cost()); } };

void main()
{       Derived der;
        der.cost();  }  // помилка! неоднозначність!
```

Компілятор видасть повідомлення про помилку – «неоднозначне звернення». Цю проблему можна усунути або правильною кваліфікацією **cost** із використанням оператора розрішення контексту (кваліфікатора області бачення) '**::**', або додаванням власного елемента **cost** до породженого класу.

2.5.2. Порядок виклику конструкторів

При використанні множинного успадкування в протоколі похідного класу необхідно викликати конструктори базових класів для ініціалізації полів даних, які успадковані від двох або більшої кількості базових класів, та ініціалізувати власні елементи об’єктів. Порядок ініціалізації такий:

1) Ініціалізація елементів базових класів здійснюється в порядку, визначеному порядком оголошення успадкування.

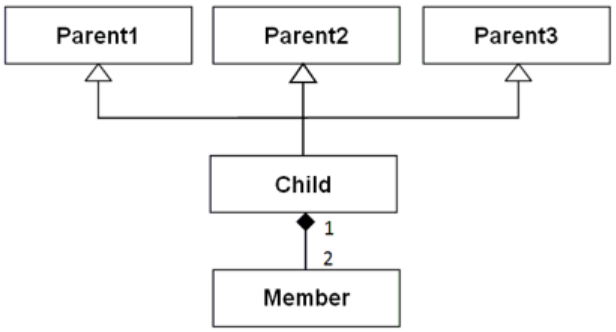


Рисунок 2.4 – Діаграма класів для демонстрації порядку виклику конструкторів

2) Елементи класів будуть ініціалізовані в порядку оголошення.

3) **void**-конструктори базових класів, які явно не вказані в списку ініціалізації, викликаються після конструкторів базових класів, які ініціалізуються явно, в тому порядку, в якому їх записано в оголошенні класу. Ці **void**-конструктори викликаються перед будь-якими конструкторами полів даних.

Порядок виклику конструкторів при множинному успадкуванні можна пояснити на прикладі програми (Приклад 2.25), для якої відповідна діаграма класів показана на рисунку 2.4. На діаграмі можна бачити, що крім множинного успадкування в ієрархію класів включено також відношення асоціації, яке відображає той факт, що до класу **Child** входять у якості елементів даних два об'єкти класу **Member**.

Приклад 2.25:

```
class Parent1
{     int p1;
  public:
    Parent1() { cout << “Construct Parent1 free parameter”; p1=0; }
    Parent1(int ap):p1(ap) { cout << “Construct Parent1 1 parameter”; }
    ~Parent1() { cout << “destruct Parent1”; }              };

class Parent2
{    int p2;
  public:
    Parent2() { cout << “Construct Parent2 free parameter”; p2=0; }
    Parent2(int ap):p2(ap) { cout << “Construct Parent2 1 parameter”; }
    ~Parent2() { cout << “destruct Parent2”; }              };

class Parent3
{    int p3;
  public:
    Parent3() { cout << “Construct Parent3 free parameter”; p3=0; }
    Parent3(int ap):p3(ap) { cout << “Construct Parent3 1 parameter”; }
    ~Parent3() { cout << “destruct Parent3”; }              };

class Member
{    int m;
  public:
    Member() { cout << “Construct Member free parameter”; m=0; }
    Member(int ap):m(ap)
          { cout << “Construct Member 1 parameter” << ap; }
    ~Member() { cout << “destruct Member”; }              };

class Child: public Parent3, public Parent2, public Parent1
{    Member mem1, mem2 ;
  public:
```

```
    Child(void) { cout << “Construct child free parameter”; }
    Child(int v1): mem1(v1)
    { cout << “Construct Child 1 parameter”; }
    Child(int v1, int v2, int v3, int m1, int m2): mem2(m1), Parent1(v1),
          Parent3(v3), Parent2 (v2), mem1(m2)
          { cout <<“Construct child 5 parameter”; }
    ~Child(void) { cout << “destruct Child”; }                };

void main()
{     Member m(16); // Construct Member 1 parameter
    Child child1;        // Construct Parent3 free parameter
                  // Construct Parent2 free parameter
                  // Construct Parent1 free parameter
                  // Construct Member free parameter
                  // Construct Member free parameter
                  // Construct child free parameter
    Child child2(5);
                  // Construct Parent3 free parameter
                  // Construct Parent2 free parameter
                  // Construct Parent1 free parameter
                  // Construct Member 1 parameter
                  // Construct Member free  parameter
                  // Construct Child 1 parameter
    Child child3(10, 20, 30, 40, 50);         }
                  // Construct Parent3 1 parameter
                  // Construct Parent2 1 parameter
                  // Construct Parent1 1 parameter
                  // Construct Member 1 parameter 50
                  // Construct Member 1 parameter 40
                  // Construct child 5 parameter
```

Повний вивід програми наведений нижче. Крім інформації про виклик конструкторів можна бачити також дані про знищення об’єктів шляхом виклику відповідних деструкторів.

```
Construct Member 1 parameter

Construct Parent3 free parameter
Construct Parent2 free parameter
Construct Parent1 free parameter
Construct Member free parameter
Construct Member free parameter
Construct child free parameter

Construct Parent3 free parameter
Construct Parent2 free parameter
Construct Parent1 free parameter
```

```
Construct Member 1 parameter
Construct Member free parameter
Construct Child 1 parameter

Construct Parent3 1 parameter
Construct Parent2 1 parameter
Construct Parent1 1 parameter
Construct Member 1 parameter 50
Construct Member 1 parameter 40
Construct child 5 parameter

destruct Child
destruct Member
destruct Member
destruct Parent1
destruct Parent2
destruct Parent3
destruct Child
destruct Member
destruct Member
destruct Parent1
destruct Parent2
destruct Parent3
destruct Child
destruct Member
destruct Member
destruct Parent1
destruct Parent2
destruct Parent3
destruct Member
```

2.5.3. Віртуальні базові класи

В прямому ациклічному графі успадкування клас може з’явитися більше ніж один раз. Це можна пояснити на прикладі ПАГ множинного успадкування, який зображено на рисунку 2.5. Елементи даних (екземпляри змінних) класу **Parent** з’являються двічі в класі **Grand_Child**. Перший їх набір успадковується через **Child1**, другий – через **Child2**. Таке успадкування буває *небажаним*. Віртуальні базові класи забезпечують механізм для уникнення дублювання елементів в класі, такому як **Crand_Child**. В інших випадках множинного успадкування дублювання успадкованих елементів даних може бути *бажаним*. Подальші приклади пояснюють сказане.

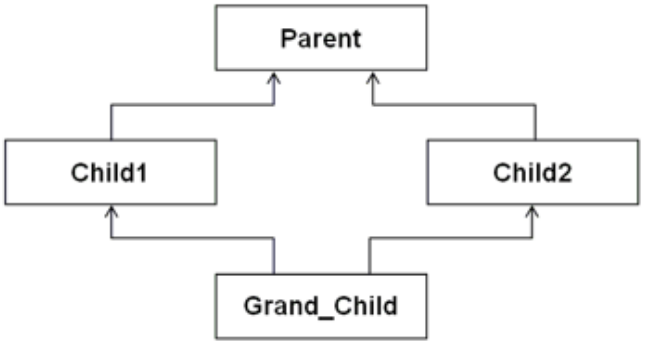


Рисунок 2.5 – Успадкування із дублюванням елементів класу **Parent** в класі **GrandChild**

Нехай клас **Parent** буде називатися **Vehicle** (транспортний засіб). В його протоколі міститься закрите поле даних **int topSpeed** (максимальна швидкість). Клас **Child1** буде називатися **Boat** (корабель), а клас **Child2** – буде називатися **Plane** (літак). Тоді клас **GrandChild** буде називатися **SeaPlane** (аквалан). Для **SeaPlane** ***бажано*** успадкувати **Boat::topSpeed** і **Plane::topSpeed**. Об'єкт класу **SeaPlane** має різну граничну швидкість в залежності від того, як він пересувається: морем чи повітрям (рис. 2.6).

З іншого боку, можна припустити, що клас **Parent** має назву **DomesticAnimal** (домашня тварина), клас **Child1** має назву **Cow**, клас **Child2** називається **Buffalo**, а клас **GrandChild** називається **Beefalo**. Нехай до протоколу класу **DomesticAnimal** включено екземпляри змінних **int weight** (вага), **float price** (ціна), **char color[20]** (колір). *Небажано*, щоб протокол класу **Beefalo** мав по два екземпляри змінних **weight**, **price**, **color**. Уникнути дублювання дозволяє застосування *віртуальних базових класів*.

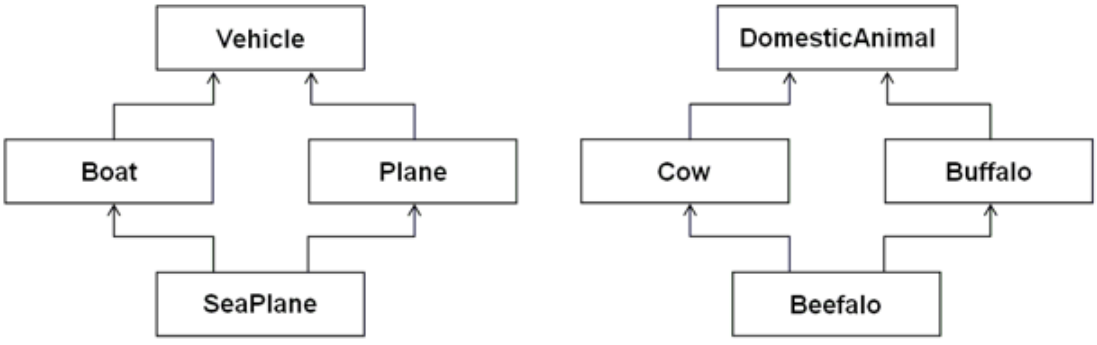


Рисунок 2.6 – Приклади *бажаного* та *небажаного* дублювання при успадкуванні

Приклад 2.26 демонструє застосування механізму *віртуальних базових класів* для уникнення *небажаного* дублювання при множинному успадкуванні. Ключове слово **virtual** вказує, що клас **DomesticAnimal** є *віртуальним базовим класом* для похідних класів.

Приклад 2.26:

```
class DomesticAnimal
{ protected:
    int weight; float price; char color[20];
  public:
    DomesticAnimal() {weight=0; price=0.; strcpy(color,"none"); }
    DomesticAnimal(int aweight, float aprice, char *acolor)
          { weight=aweight; price=aprice; strcpy(color, acolor); }
    virtual void print()
          { cout << "weight="<< weight << " price=" <<
            price << " color=" << color<<endl; }              };

class Cow: public virtual DomesticAnimal
{ public:
    Cow() { }
    Cow(int aweight, float aprice, char *acolor)
          { weight=aweight; price=aprice; strcpy(color, acolor); }
    void print() { cout << “Cow has properties\n”; DomesticAnimal::print(); }          };

class Buffalo: public virtual DomesticAnimal
{ public:
    Buffalo() { }
    Buffalo(int aweight, float aprice, char *acolor)
          { weight=aweight; price=aprice; strcpy(color, acolor); }
    void print() { cout << “Buffalo has properties\n”;
                        DomesticAnimal::print(); }              };

class Beefalo: public Cow, public Buffalo
{ public:
    Beefalo(int aweight, float aprice, char *acolor)
          { weight=aweight; price=aprice; strcpy(color, acolor); }
    void print() { cout << "Beefalo has properties\n";
          DomesticAnimal::print(); }            };

void main()
{     Cow aCow(1400, 375.0, (char *)"black and white");
    Beefalo aBeefalo(1700, 525.0, "brown and white");
    DomesticAnimal& myCow=aCow;
    DomesticAnimal& myBeefalo=aBeefalo;
    myCow.print();
    myBeefalo.print();                  }
```

Результат роботи:

```
Cow has propeties
weight=1400 price=375 color=black and white
Beefalo has propeties
weight=1700 price=525 color=brown and white
```

Код наведеної вище програми вирішує проблему класу **Beefalo**, який успадковує поля даних **weight**, **price**, **color**. Обєкти класу **Beefalo** мають по одному полю даних для ваги, ціни і кольору.

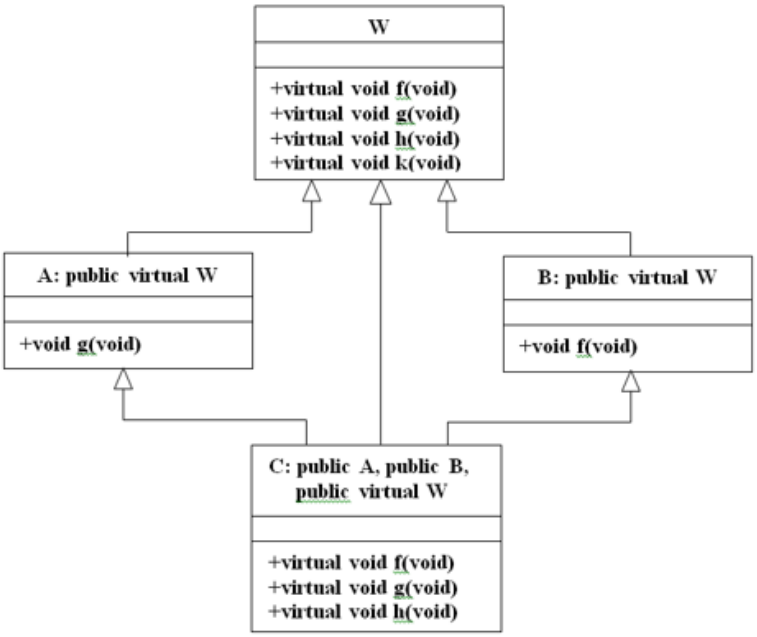
Віртуальні базові класи ініціалізуються (викликається **void**-конструктор) перед будь-якими невіртуальними базовими класами та у тій послідовності, в якій вони з'являються в ПАГ успадкування при перегляді його знизу-вгору і зліва-направо.

Якщо віртуальний базовий клас має хоча б один конструктор, то він повинен мати **void**-конструктор.

Ключове слово **virtual** в класі **Cow** і класі **Buffalo** запобігає багаторазовому копіюванню полів даних **weight**, **price**, **color** з предків класу **Beefalo**.

Наступний приклад демонструє деякі особливості застосування віртуальних функцій віртуальних базових класів.

Приклад 2.27:




```
class W
{ public:
```

```
    virtual void f() { cout << " W::f"; }
    virtual void g() {cout << " W::g"; }
    virtual void h() {cout << " W::h"; }
    virtual void k() {cout << " W::k"; }              };

class A: public virtual W
{ public:
    void g() { cout << " A::g";}                      };

class B: public virtual W
{ public:
    void f() { cout << " B::f"; }                     };

class C: public A, public B, public virtual W
{ public:
    void h() { cout << "C::h"; }
    void f() { B::f(); }
    void g() { A::g(); }                 };

void main()
{      C *pc=new C;
    pc->f();            // C::f() ( B::f)
    pc->g();            // C::g() ( A::g)
    pc->h();            // C::h
    ((A *)pc)->f();     // C::f (B::f)
    ((W *)pc)->f();     // C::f (B::f)

    B *pb=new B;
    pb->f();                    // B::f
    pb->g();                    // W::g
    ((W *)pb)->f();             // B::f
    A *pa=new A;
    pa->f();                    // W::f
    pa->g();                    // A::g
    ((W *)pa)->g();}            // A::g
```

Вивід програми записано у коментарях.

У виклику **((A *)pc)->f()** виконується функція **f** класу **C**. Те ж саме має місце і у виклику **((W *)pc)->f()**. Таблиця віртуальних функцій використовує визначення **f** в класі **C**. Це свідчить про те, що при виклику «верхньої» віртуальної функції по одному шляху ПАГ може закінчитися викликом функції по іншому шляху.

Якщо віртуальний базовий клас і похідний клас розділяють ім’я якогось поля, функції-метода або перерахування, то ім’я в похідному класі приховує ім’я в віртуальному базовому класі. Наприклад:

```
class V { public: int i; void f(); };
class A: virtual public V {public: int i; int f(); };
class B: virtual public V { };
class C: public A, public B
{ void g() { i=f(); } };            // Вірно A::i та A::f() приховують V::i та V::f()
```

2.6. Обробка виключень в C++

2.6.1. Варіанти обробки помилок, що не пов'язані з використанням виключень

Повне та остаточне зникнення помилок в програмах – мрія кожного програміста. На жаль, ця мрія залишається тільки мрією, в усякому разі в найближчі часи. Тому істотною частиною процесу створення будь-якої реальної програми є пошук і виправлення помилок.

Перш ніж перейти до розгляду засобів мови C++ для обробки помилок, зупинимося на тому, які помилки потребують обробки.

Помилки компіляції. Доки вони всі не виправлені, програма не готова і її неможливо запустити на виконання.

У багатьох випадках мова C++ допомагає уникати помилок, виконуючи суворий контроль на стадії компіляції. Загалом, чим більший контроль на стадії компіляції, тим менше помилок залишається при виконанні програми.

Сучасні компілятори мають потужні засоби запобігання помилкам. Найпоширеніші з них перелічено далі.

1. Контроль типів. Використання неприпустимих операцій, змішування несумісних типів будуть знайдені компілятором.

2. Обов'язкове оголошення імен до їх використання. При зміні визначення змінної або функції неважко викрити всі місця, де вона використовується. Неможливо викликати функцію з невірною кількістю аргументів.

3. Обмежена бачність імен. Зменшується можливість конфліктів імен, неправильного перевизначення імен.

Далі розглядаються тільки помилки, що виникають *під час виконання програми*.

Перший вид помилок, який завжди спадає на думку – це ***помилки програмування***. До них відносяться також помилки у алгоритмі, в логіці роботи програми, та чисто програмістські помилки. Низку можливих помилок кожен може згадати з власного досвіду, але ще більше їх буде спостерігатися при розробці складних програм.

Найважливішим засобом зменшення ймовірності помилок є власне об'єктно-орієнтований підхід до програмування, який підтримує мова C++. Поряд з перевагами об'єктного програмування, про які зазначалось раніше, побудова програми з класів дозволяє налагоджувати окремо один від одного та

будувати програми з надійних складових, використовуючи ті ж самі класи багаторазово.

Незважаючи на всі ці позитивні якості мови, залишається великий «простір» для створення помилок в програмах.

Розуміння того факту, що ідеальних програм не існує, допомагає розробляти більш надійні програми. Найважливішим є контроль даних – необхідно перевіряти в програмі все, що може містити помилку. Якщо в програмі передбачається деяка умова, не зайвим буває її перевірити, хоча б у початковій версії програми, до того, як стане ясним з досвіду, що ця умова дійсно виконується. Хорошим правилом є перевірка покажчиків, що передаються у якості аргументів, на рівняння їх нулю, перевірка того, що індекси не виходять за межі масиву тощо.

Вирішальною якістю, що зменшує кількість помилок, є *уважність*, *акуратність* й *досвід*.

Другим видом помилок є «***передбачені***», заплановані помилки. Якщо розробляється програма діалогу з користувачем, ця програма мусить розумно реагувати й обробляти помилкові натискання клавіш. Програма читання тексту повинна враховувати можливі синтаксичні помилки. Програми передавання даних по каналах зв'язку мають обробляти вади і можливі збої при передаванні. Такі помилки – це, взагалі кажучи, не помилки з погляду програми, а заплановані ситуації, які вона обробляє.

Третій вид помилок також, деякою мірою, передбачений. Це ***виключні ситуації***, які можуть мати місце навіть якщо в програмі немає помилок. Наприклад, нестача пам'яті для створення нового об'єкту. Або збій диску при вибірці інформації з бази даних.

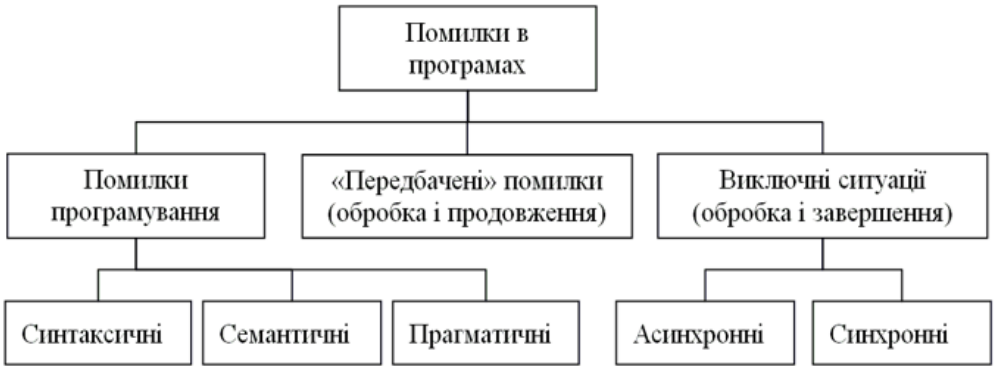


Рисунок 2.7 – Класифікація помилок в програмах

Саме обробка цих двох останніх видів помилок («***передбачених***» та ***виключних ситуацій***) й розглядається в даному розділі. Межа між ними досить умовна. Наприклад, для більшості програм збій диску – виключна ситуація, але для операційної системи збій диску має бути передбаченим та мусить бути обробленим. Два види помилок краще можна розмежувати за реакцією програми, яка має бути передбачена. Якщо після планових помилок програма повинна продовжити працювати, то після виключних ситуацій необхідно лише зберегти вже обчислені дані й завершити програму.

Примітка*: Значення слова «прагматизм». Напрямок у філософії, згідно з яким об'єктивність істини заперечується, істинним визнається лише те, що дає корисні практичні результати. Слідування в усьому вузькопрактичним інтересам, міркуванням користі та вигоди*.

Підтвердження та сигнали

Спочатку розглянемо *підтвердження*, як засоби, що призначені для подолання помилкових умов і виключень. Одна з точок зору на цю проблему полягає в тому, що виключення виникають при порушенні договірної гарантії, наданої постачальником (розробником) коду, відносно того, як має застосовуватися цей код. При такій моделі користувач гарантує дотримання умов застосування коду, а постачальник гарантує, що код при цих умовах працює правильно. В цій методології підтвердження забезпечують гарантії правильної роботи.

Також буде охарактеризовано пакет **signal.h** для **ANSI C**, який обробляє *асинхронні виключення* апаратних засобів. Ці виключення залежать від системи і, як правило, генеруються апаратними засобами та компонентами операційної системи при виявлені спеціальних умов. Пакет **signal.h** також обробляє *синхронні виключення* із використанням функції **raise()**.

Використання підтверджень (assert.h)

Той факт, що обчислення завершилося із вірним результатом, частково можна розглядати як доказ того, що програма працює правильно. Крім того, вірний результат залежить від коректності вихідних даних, тому той, хто активізував обчислення, несе відповідальність за забезпечення правильного вводу. Так можна охарактеризувати *передумову*. Обчислення, якщо воно

завершилося успішно, задовольняє *постумові*. *Підтвердження* (або перевірка) виконання *передумов* і *постумов* підвищує надійність програми.

Зусилля із забезпечення повністю формального доказу правильності – ідеалізація, яка зазвичай не виконується. Незважаючи на це, такі підтвердження можуть контролюватися під час виконання для того, щоб надавати досить корисну діагностику. Система ретельно продуманих підтверджень часто примушує програміста уникати помилок і «пасток». Різні спільноти користувачів C і C++ все більше і більше підкреслюють важливість використання таких підтверджень.

Стандартний бібліотечний файл **assert.h** забезпечує доступ до *макрокоманди* **assert**:

```
void assert(int <вираз>);
```

Якщо **<вираз>** оцінюється, як *неістинний* (повертає значення **0** або **false**), то виконання програми переривається із виводом діагностичного повідомлення. Підтвердження не перевіряються, якщо визначено макрокоманду **NDEBUG**. Її треба визначити перед підключенням **assert.h**.

Застосування підтверджень можна пояснити на прикладі конструктора класу **vect** з одним параметром, який перевіряє умови та результат спроби створити масив заданого розміру **n**. Варіант такого конструктора без використання підтверджень може мати вигляд:

```
vect::vect(int n) {  if(n<1) { cerr << " illegal vect size " << n; exit(1); }
                     size=n; p=new int[size];  }
```

Такий конструктор можна замінити на:

```
vect::vect(int n) {  assert(n > 0);                  //  передумова
                     size=n; p=new int[size];
                     assert(p!=NULL);    }           // постумова
```

Використання підтверджень, які заміняють перевірки деяких умов, робить методологію більш однорідною. Це сприяє покращенню практики боротьби із помилками. Проте, *недоліком* даного підходу можна вважати те, що системи підтверджень не припускає повторень або іншої стратегії відновлення для продовження виконання програми.

Використання сигналів (signal.h)

Файл **signal.h** забезпечує стандартний механізм для безпосередньої обробки *асинхронних виключень*, що визначаються системою. Виключення визначаються в файлі **signal.h** і являють собою залежні від системи значення цілих чисел. Наприклад,

```
#define SIGINT 2    /*  сигнал переривання  */
#define SIGFPE 8   /* виключення операцій над числами з плаваючою  крапкою */
#define SIGABRT 22        /*  сигнал аварійного завершення */
```

Ці виключення можуть відрізнятися у різних системах. Але частіше вони трактуються однаково. Наприклад, натиснення **Ctrl+C** на клавіатурі в більшості систем генерує сигнал переривання (**SIGINT**), яке зазвичай завершує поточний процес користувача.

Для того щоб згенерувати *явне* виключення, можна застосувати функцію **raise()**, прототип якої знаходиться в **signal.h**. Наприклад,

```
raise(SIGFPE);       // генерувати сигнал виключення,
                     // для операцій з плаваючою крапкою
```

Визначені в файлі **signal.h** виключення можна *обробляти* із використанням функції **signal()**. Вона зв'язую *функцію-обробник* із сигналом переривання. Також вона може використовуватися для того, щоб ігнорувати сигнал або повторно встановлювати дії за умовчанням.

```
signal(SIGABRT, my_abrt);        // виклик my_abrt() при отриманні SIGABRT
signal(SIGABRT, SIG_DFL);        // дії за умовчанням при виникненні SIGABRT
signal(SIGFPE, SIG_IGN);         // ігнорувати сигнал SIGFPE
```

Це має назву ***встановлення обробника***. Встановлення обробника замінює системний обробник, на визначений користувачем. В наступному прикладі ці механізми використовуються для обробки сигналу **SIGINT**, який передається у програму при вводі з клавіатури **Ctrl+C**. Після виникнення переривання (**Ctrl+C**) обробник запитує користувача, чи повинна програма продовжувати виконання.

Приклад 2.28:

```
#include <signal.h>
#include <time.h>
void cntrl_c_handler(int sig);       // Функція-обробник
void main()
```

```
{    double long i=0, j;
   cout << "Count to j million. Enter j: "; cin >>j; j*=1000000;
   signal(SIGINT, cntrl_c_handler);        // Встановлення обробника
   // Функція-обробник зв'язується з перериванням SIGINT (ctrl+C).
   // Після виникнення наступного переривання замість дій
   // за умовчанням система викличе cntrl_c_handler():
   cout << (double)clock()/CLOCKS_PER_SEC << " start time\n";
   while(1)
   {    i++;
      if(i > j)
      { cout << (double)clock()/CLOCKS_PER_SEC <<" end loop\n";
        cout << "HIT " << j/1000000 << " MILLION" << endl;
        raise(SIGINT);        // Явне генерування виключення.
                              // Викликається обробник cntrl_c_handler()
        cout << "\nEnter j: ";
        cin >> j;
        j*=1000000;   i=0;
        cout << (double)clock()/CLOCKS_PER_SEC << " start loop\n";
        }        }                // Кінець циклу while(1) і main()
}
//  Наступна функція-обробник обробляє виключення SIGINT
void cntrl_c_handler(int sig)
{        char c;
//       При виникненні переривання користувача запитують,
//       чи повинна програма продовжувати виконання
        cout << " INTERRUPT" << " type Y to continue :"; cin >> c;
        if(c=='Y')
        // Повторне встановлення обробника. Без цього
        // система повертається до обробки за умовчанням:
          signal(SIGINT, cntrl_c_handler);
        else exit(0);                    }
```

Розглянуту методику можна застосувати для обробки виключень, що виникають у конструкторі класу **vect**. Для цього наведені раніше підтвердження можна замінити умовними операторами, які встановлюють сигнали **SIGUSR1** та **SIGUSR2** (ці сигнали зарезервовані для користувачів).

```
const int SIGHEAP=SIGUSR1;    // встановлення глобальної константи для SIGUSR1
const int SIGSIZE = SIGUSR2;    // встановлення глобальної константи для SIGUSR2
vect::vect(int n)
{        if(n<=0) { raise(SIGSIZE);
        cout << " Enter vect size n: "; cin >> n; }
        p= new int[n];
        if(p==NULL) { raise(SIGHEAP); p=new int[n]; }          }
```

Далі необхідно визначити обробники відповідних сигналів **vect_size_handle** та **vect_heap_handler**. Наприклад:

```
void vect_size_handler(int sig)
{       char c;
        cout << "SIZE ERROR, Enter Y to continue:"; cin >> c;
        if(c=='Y') { signal(SIGSIZE,vect_size_handler); }
        else exit(0);   }

void vect_heap_handler(int sig) {
        // Можливі  дії:
        // повернути пам’ять системі
        // або коректно завершити виконання
}
```

Сигнали і підтвердження часто можуть замінюватися локальними перевірками і викликами функцій, які забезпечують більшу гнучкість. Проте, підтвердження та сигнали забезпечують більш однорідну методологію і пом’якшують спроби надмірного відновлення, яке, у свою чергу, у більшості випадків свідчить про неправильне програмування та неакуратність. Підтвердження перевіряють правильність виконання попередніх угод. Сигнали і обробники дозволяють відновлення в глобальному контексті для непередбачених умов. Сигнали обмежені низкою *асинхронних* умов, які залежать від системи, проте ця схема легко розширюється встановленням глобальних даних, які скидаються під час виклику умови **raise**.

2.6.2. Обробка виключних ситуацій. Блок try, оператори catch, throw

Виключення – це виникнення помилкових умов (наприклад, ділення на нуль при виконанні арифметичних операцій). Як правило, ці умови завершують програму користувача з системним повідомленням про помилку. C++ надає програмісту можливість відновлювати програму при наявності цих умов і продовжувати її виконання.

C++ забезпечує вбудований механізм обробки помилок, який має назву «***обробка виключних ситуацій***». Завдяки цьому механізму можна спростити управління і реакцію на помилки, що виникають під час виконання програми. Обробка виключних ситуацій в C++ будується за допомогою трьох ключових слів: **try**, **catch**, **throw**.

Оператори програми, під час виконанні яких треба забезпечити обробку виключних ситуацій, розміщуються в блоці **try{…}**. Якщо помилку виявлено

всередині блоку **try**, відповідна виключна ситуація генерується за допомогою оператору **throw**. Перехоплення і обробка виключної ситуації виконується за допомогою конструкції **catch**.

Будь-який оператор, який генерує виключну ситуацію, повинен виконуватися всередині блоку **try**. (Функції, які викликаються всередині блоку **try** також можуть генерувати виключні ситуації). Будь-яку виключну ситуацію необхідно перехоплювати оператором **catch**, що розміщується безпосередньо за блоком **try**, який згенерував відповідне виключення.

```
try
{       /*  Оператори, в яких перевіряються виключення  */   }

catch (type1 arg)
{       // Оператори, які виконуються
        // при виникненні виключення типу type1
}
catch (type2 arg)
{       // Оператори, які виконуються
        // при виникненні виключення типу type2
}
.........
catch (typeN arg)
{       // Оператори, які виконуються
        // при виникненні виключення типу typeN
}
```

Блок **try** повинен містити ту частину програми, в якій необхідно відслідковувати помилки. Це може бути декілька операторів всередині однієї функції, а також і всі оператори функції **main()** (що призведе до відстеження помилок в усій програмі).

Коли виключна ситуація виникає, вона перехоплюється відповідним оператором **catch**, який обробляє цю ситуацію. З блоком **try** може бути зв’язаний більш ніж один оператор **catch**. Який саме оператор **catch** використовується, залежить від *типу виключної ситуації*. Тобто, якщо тип даних, вказаний в операторі **catch**, ***відповідає*** типу виключної ситуації, то виконується даний оператор **catch**. Всі інші оператори блоку **try** пропускаються. Якщо виключну ситуацію перехоплено, то аргумент **arg** отримує її значення. Можна перехоплювати будь-які типи даних, включаючи і типи, що створені користувачем.

Загальний формат оператору **throw**:

```
throw <виключна ситуація>;
```

де **<виключнаї ситуація>** – це виключна ситуація яку генерує оператор **throw**. Оператор **throw** повинен виконуватися або всередині блоку **try**, або у будь-якій функції, яку цей блок викликає (прямо чи непрямо).

Простим прикладом, що пояснює механізм генерування і обробки виключень, може бути наступна програма.

Приклад 2.29:

```
void main()
{       try{    throw 10;
          }     // Неприпустимо ставити крапку з комою (;)
   catch(int i)
       {        cout << " error " << i << endl;
       }        // Неприпустимо ставити крапку з комою (;)
       return;          }
```

На екрані з'явиться повідомлення:

```
error 10
```

В C++ реалізовано чутливий до контексту механізм обробки виключних ситуацій. Виходячи з цього, він може бути більш поінформованим ніж обробник з **signal.h** і може забезпечити більш складне відновлення. Проте він не може обробляти асинхронні виключення, які визначено в **signal.h**.

Раніше застосування *підтверджень* і *сигналів* було продемонстровано на прикладі конструктора класу **vect**. Аналогічний конструктор може бути розроблений також із застосування засобів *обробки виключень* C++.

```
vect::vect(int n)
{   if(n <1) throw(n);
    p=new int[n];
    if(p==NULL) throw("FREE STORE EXHAUSTED");   }

void g(int n)
{   try
    {     vect a(n), b(n); ...  …  …    }

    catch(int n) { ... }            // реакція на неправильний розмір масиву
    catch(char *error) { ... }  // дії при перевищенні вільної пам'яті
}
```

Перший **throw()** має цілий аргумент і відповідає *сигнатурі* **catch(int n)**. Якщо розмір масиву у параметрі конструктора вказано неправильно, цей обробник буде виконувати відповідні дії, наприклад, вивід повідомлення про помилку і аварійне завершення роботи. Аргумент другого **throw()** – це покажчик на рядок символів, що відповідає сигнатурі **catch(char *error)**.

Розглянутий конструктор класу **vect** перевіряє тільки одну змінну на припустимість значення. Це виглядає дещо штучно, оскільки конструктор міг би заборонити будь-яку неправильну ініціалізацію і встановлювати відповідні виключення, дозволяючи обробнику виправляти помилки. У більш загальній формі конструктор об’єкту може мати вигляд:

```
Object::Object(<аргументи>)
{   if(<неприпустимий аргумент 1>)
                throw <вираз 1>;
    if(<неприпустимий аргумент 2>)
                throw <вираз 2>;
        // спроба створити об’єкт
}
```

Конструктор **Object** у такому вигляді забезпечує низку виразів для визначення забороненого стану. Теперь в блоці **try** можна використовувати цю інформацію для виправлення або переривання неправильного коду.

```
try     {       /*  стійкий до помилок код  */   }
catch(<оголошення 1>) { /* виправлення помилки 1 */ }
catch(<оголошення 2>) { /* виправлення помилки 2 */ }
catch(<оголошення K>) { /* виправлення помилки K */ }
        // тепер змінні стану перевірено або виправлено
```

Далі синтаксис механізмів обробки виключних ситуацій розглядається більш детально.

Встановлення виключень

Синтаксично оператор **throw** використовується в двох формах.

```
throw;
```

та

```
throw <вираз>;
```

Оператор **throw** *встановлює виключення*, яке визначається ***виразом***. При наявності вкладених один в одного блоків **try**, для вибору обробника **catch** використовується оператор **throw** з блоку **try**, найнижчого рівня вкладеності.

Зауваження:

1. Якщо генерується виключна ситуація, для якої немає відповідного оператора **catch**, можливе аварійне завершення програми.

2. Оператор **throw** без аргументу **повторно** встановлює поточне виключення. Як правило такий формат використовується, коли для подальшої обробки виключення потрібен інший обробник, який викликається з першого.

Встановлене виключення – статичний, тимчасовий об'єкт, який зберігається до тих пір, доки не виконано вихід з гілки обробки виключної ситуації. **Вираз** *перехоплюється обробником* **catch**, який може використовувати захоплене значення.

```
void foo()
{       int i;
        . . .
        throw i;  }
main()
{       try {    foo();  }
        catch(int n) { . . . }          }
```

Значення цілого числа, яке передане через **throw i**, зберігається до завершення роботи обробника з відповідною сигнатурою **catch(int n)**. Це значення доступне для використання всередині обробника у вигляді аргументу **n**.

В програмах допускається перевстановлення виключень. Перевстановлення виключення може мати наступний вигляд

```
catch(int n)
{       . . .
        throw;          // переустановлення виключення
}
```

В даному прикладі встановлений **вираз** мав цілий тип, тому перевстановлене виключення також являє собою цілий об'єкт, який обробляється найближчим обробником даного типу.

Концептуально встановлений **вираз** завжди передає інформацію в обробники. Часто обробники не мають потреби в цій інформації. Наприклад, обробнику, який виводить повідомлення і аварійно завершує роботу, не потрібна

ніяка інформація від оточення. Проте, користувачу може статися у пригоді додаткова інформація для виводу її на екран або для прийняття рішення стосовно дій обробника. В такому випадку можливо формувати інформацію у вигляді об'єкта певного класу помилок:

```
enum error { bounds, heap, other };
class vect_error
{       error e_type;
        int ub, index, size;
  public:
        vect_error(error, int, int);            // помилка першого типу
        vect_error(error, int);         }       // помилка другого типу
```

В даному прикладі використання в операторі **throw** об'єкта типу **vect_error** може бути більш інформативним по відношенню до обробника, ніж тільки встановлення виразів простих типів:

```
. . .
throw vect_error(bounds, i, ub);
. . .
```

Блоки повторних спроб try

Синтаксично, блок **try** має наступний формат:

```
try
< складений оператор >
< обробники >
```

Блок **try** – контекст для прийняття рішення про те, які обробники викликаються для встановленого виключення. Порядок, в якому визначено обробники, визначає порядок, в якому обробники перевіряються для встановленого виключення відповідного типу.

В C++ блоки **try** можуть бути вкладеними. Якщо в поточному блоці **try** немає відповідного обробника, буде обрано обробник з найближчого зовнішнього блоку **try**. Якщо його не знайдено і там, тоді використовується поведінка за умовчанням.

Обробники catch

Синтаксис обробників використовує наступний формат:

```
catch (< формальний параметр >)
< складений оператор >
```

catch виглядає подібно оголошенню функції з одним параметром без типу, що повертається.

```
catch (char *message)
{       cerr << message << endl; exit(1);         }

catch ( ... )
{       // дія, яку треба виконати за умовчанням
        cerr << " That's all folks " << endl; exit(2);       }
```

В списку аргументів допускається сигнатура виду '**…**', яка відповідає будь-якому параметру (дія за замовчанням). Крім цього, формальний параметр може бути абстрактним оголошенням. Це означає, що він може мати інформацію про тип без імені змінної.

Обробник викликається відповідним оператором **throw**. При цьому, фактично, відбувається вихід з блоку **try**. Система викликає функції вивільнення, які включають деструктори для будь-яких об'єктів, локальних для блока **try**. Частково створений об'єкт буде мати деструктори, які викликатимуся на будь-які частини створених підоб'єктів.

Специфікація виключення

Специфікація виключень дозволяє перелічити в оголошенні функції всі виключення, які вона може генерувати. Це гарантує, що інші виключення функція генерувати не буде. Така специфікація записується за списком формальних параметрів функції. Вона складається з ключового слова **throw**, за яким йде список типів виключень в дужках.

Синтаксично специфікація виключення представляє собою частину оголошення функції і має форму

```
<заголовок функції> throw (<список типів>)
```

Список типів – це список типів, які може мати вираз **throw** всередині функції. Якщо цей список порожній, компілятор може припустити, що безпосередньо або побічно, **throw** не буде виконуватися функцією.

```
void foo() throw(int, over_flow);
```

```
void noex(int i) throw();
```

Якщо специфікація виключення присутня у визначенні функції, тоді виникає припущення, що такою функцією може бути встановлено будь-яке з перелічених виключення. Хорошим стилем програмування вважається показувати за допомогою специфікації, які очікуються виключення. Порушення цих специфікацій призведе до помилки під час виконання програми.

2.6.3. Обробка виключень в класах

Виключення в класах та класи виключень

Розглянуті механізми боротьби з помилками знайшли ефективне застосування в об'єктно-орієнтованому програмуванні.

Як відомо, об'єктно-орієнтовані додатки створюють об'єкти певних класів і працюють з ними. У нормальній ситуації виклики методів не призводять ні до яких помилок. Але при певних обставинах в програмі виникає помилка, яку викриває метод. Він інформує додаток про те, що трапилось. При використанні виключень така подія називається ***генеруванням виключної ситуації***. В об'єктно-орієнтованому додатку для цього мається окрема секція коду, у якій розташовано операції, що обробляють помилки. Цей код називають ***обробником виключних ситуацій*** або ***обробником виключень***. В ньому перехоплюються генеровані методами виключення. Будь-який код у додатку, який використовує об'єкти класу, розміщується в ***блоці повторних спроб***. Відповідно, помилки, які виникають в даному блоці, будуть зафіксовані обробником виключень. Код, який не має ніякого відношення до класу, не включається до блоку повторних спроб. Розглянуту схему обробки виключень в класах показано на рисунку 2.8.

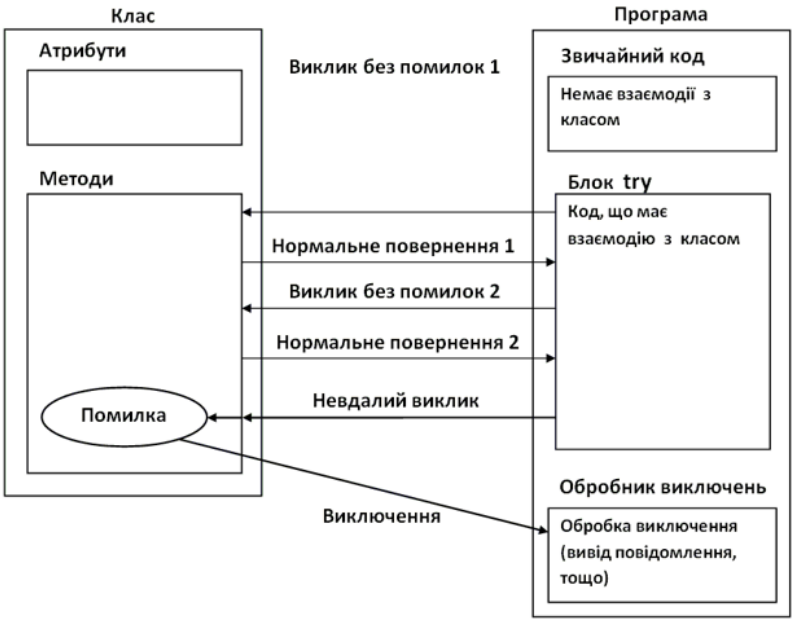


Рисунок 2.8 – Обробка виключних ситуацій в класах

Механізм виключень в класах використовує вже розглянуті ключові слова: **catch**, **throw** і **try**. Крім того, в об'єктно-орієнтованих додатках, як правило, створюється новий тип сутності, який називається ***класом*** **exception** (класом виключень). Наступний приклад – це ще не завершена програма, а тільки схема, яка демонструє синтаксис.

```
class AClass                    // звичайний клас
{ public:
        class AnError           // клас exception
        { … };
        void Func()             // метод класу AClass
        {   if( /* умова виникнення помилки */ )
              throw AnError();  // генерування  виключення
        }               };
/*********************************************************
int main()                      // додаток
{       try                     // блок повторних спроб
        {  AClass obj1;
           obj1.Func();         // тут може виникнути помилка
        }
        catch(AClass::AnError)  // обробник виключень
        { ( /* обробка помилки */ }
        /*продовження після обробки виключень */
        return 0;       }
```

Програма починається зі створення класу **AClass**, в якому можуть виникати помилки. Клас виключень, **AnError**, визначено у загальнодоступній частині класу **AClass**. Методи класу **AClass** необхідно ретельно дослідити на предмет можливого виникнення помилок. Якщо такі буде знайдено, то за допомогою оператора **throw** і конструктора класу виключень необхідно згенерувати виключні ситуації:

```
throw AnError():      //'throw' і конструктор класу AnError
```

В **main()** всі оператори, які будь-яким чином стосуються класу **AClass**, розміщуються всередині блоку повторних спроб. Якщо якісь з цих операторів призводять до того, що виникають підозри на наявність помилки в методі класу **AClass**, то виконується генерування виключення, і управління програмою переходить до обробника виключень, який розміщується безпосередньо за блоком повторних спроб.

Простий приклад виключення в класі

Наступна програма (Приклад 2.30) – це вже реально працююча програма, що реалізує ідею виключень в класах. В даному прикладі створюється стек для зберігання цілих чисел. Програма здатна викривати дві важливі помилки, а саме: додаток може спробувати вибрати зі стеку або записати у нього забагато елементів, що призведе до отримання неправильних даних, або до перевищення розміру масиву. В програмі використовується механізм виключень, щоб обробляти ці ситуації.

Приклад 2.30:

```
#include <iostream>
using namespace std;
const int MAX = 3;  // в стеку максимум 3 цілих числа
//**************************************************************************
class Stack
{ private:
        int st[MAX]    // стек: масив цілих чисел
        int top;                    // індекс вершини стеку
  public:
        class Range  // клас виключень для Stack
        {         /* тіло класу порожнє */              };
//--------------------------------------------------------------------------------
        Stack()                     // конструктор
                { top = -1; }
```

```
//--------------------------------------------------------------------------------
        void push(int var)          // занести число в стек
        { if(top >= MAX-1)          // якщо стек повний,
                throw Range();      // генерувати виключення
          st[++top] = var;          // занести число в стек
        }
//--------------------------------------------------------------------------------
        int pop()                   // взяти число зі стеку
        { if(top < 0)               // якщо стек пустий,
                throw Range();      // виключення
          return st[top--];         // взяти число зі стеку
        }
};
//*************************************************************************
int main()
{    Stack s1;
   try
   { s1.push(11);
     s1.push(22);
     s1.push(33);
   //  s1.push(44);                 // Стек заповнено!
     cout  <<  "1: “ << s1.pop() << endl;
     cout  <<  "2: “ << s1.pop() << endl;
     cout <<  "3: “ << s1.pop() << endl;
     cout <<  "4: “ << s1.pop() << endl;   // Стек порожній!
   }
   catch(Stack::Range)                  //Обробник
   { cout << "Виключення: стек переповнений або порожній\n"; }
   cout<<"Продовження після обробки або нормальний вихід)\n";
   return 0;
}
```

В даному прикладі стек навмисно зроблено маленьким, щоб простіше було викликати виключні ситуації.

Розгляд програми дозволяє виділити в ній чотири частини, які безпосередньо стосуються виключень.

1. В специфікації класу визначено клас виключень **Range**.

2. Також в класі розміщено оператори **throw**, які генерують виключення.

3. В **main()** є частина коду, яка може призвести до виключних ситуацій, – це *блок повторних спроб*.

4. Також є код, в якому ці ситуації обробляються, – *обробник виключень*.

Клас виключень

В розглянутому прикладі клас виключень визначено всередині класу **Stack**:

```
class Range // клас виключень для Stack
{       /* тіло класу порожнє */           };
```

Як можна бачити, тіло класу порожнє, і тому його об’єкти не матимуть ні даних, ні методів. Все, що потрібно для перехоплення виключень в цій простій програмі, – це ім'я класу **Range**. Воно використовується для зв’язування виразу генерування виключення із обробником виключень (класу не обов’язково бути порожнім, як буде показано далі).

Генерування виключення

В класі **Stack** виключення виникає, коли додаток намагається вибрати значення з порожнього стеку або занести значення у вже повністю заповнений стек. Щоб повідомити додаток про виконання неприпустимої операції зі стеком класу **Stack**, методи цього класу перевіряють вказані умови за допомогою оператора **if** і генерують виключення, якщо ці умови виконуються. В програмі виключення генеруються в двох місцях, обидва рази за допомогою виразу

```
throw Range();    // генерувати виключення
```

Друге слово цього простого речення неявно запускає конструктор класу **Range** для створення відповідного об’єкту цього класу. Що стосується першого слова, то воно передає управління обробнику помилок (виключень).

Блок повторних спроб

Всі оператори в **main()**, в яких можуть виникати помилки, тобто, в яких виконується маніпулювання даними об’єктів класу **Stack**, загорнуто в фігурні дужки, перед яким стоїть ключове слово **try**:

```
try
{       // код, що працює з об’єктами, в яких можуть
        // виникати помилки
}
```

Обробник виключень

Код, в якому містяться операції із обробки помилок, загортається у фігурні дужки і починається з ключового слова **catch**. Ім'я класу виключень необхідно кваліфікувати ім'ям класу, елементом якого він являється. В даному прикладі це **Stack::Range**.

```
catch(Stack::Range)
{       /*  код-обробник помилок  */   }
```

Така конструкція, як вже відомо, називається ***обробником виключень***. Її необхідно розміщувати безпосередньо за блоком повторних спроб. В розглянутому прикладі обробник виключень займається тільки виводом на екран повідомлення про помилки, що дає можливість користувачеві знати, чому його програма завершила роботу. Управління програмою «провалюється» на дно обробника помилок і програма може продовжувати роботу безпосередньо з цього місця. Обробник може передати управління і зовсім в інше місце, проте частіше він просто завершує роботу програми.

Послідовність подій

Підбиваючи підсумки треба ще раз звернути увагу на послідовність дій у програмі при виникненні помилки.

1. Код нормально виконується поза межами блоку повторних спроб.
2. Управління переходить до блоку повторних спроб.
3. Виконання якогось оператора в цьому блоці призводить до виникнення помилки в методі.
4. Метод генерує виключення.
5. Управління переходить до обробника помилок, який розташований безпосередньо за блоком повторних спроб.

Безперечною *перевагою* такого підходу до обробки помилок є цілком прозорий і чіткий кінцевий алгоритм. В будь-якому операторі блоку повторних спроб може виникнути помилка, проте все залишається під контролем. Не треба турбуватися про отримання значення, що повертається кожним з операторів, тому що це виконується автоматично. В даний конкретний приклад свідомо вставлено два оператора з помилками. Перший:

```
//  s1.push(44);                // Стек заповнено!
```

призведе до виключної ситуації, якщо видалити знаки коментаря перед ним, а другий:

```
cout <<  "4: “ << s1.pop() << endl;        // Стек порожній!
```

призведе до виключної ситуації, якщо закоментувати перший оператор. В обох випадках буде виведене те ж саме повідомлення:

```
Виключення: стек переповнений або порожній
```

Обробка декількох виключення

Можна спроектувати клас таким чином, щоб він генерував стільки виключень, скільки потрібно. Це демонструє наступна програма, яка відрізняється від попередньої тим, що вона генерує окремі виключення при переповненні стеку і спробі вилучення з порожнього стеку.

Приклад 2.31:

```
// Демонстрація обробника двох виключень
#include <iostream>
using namespace std;
const int MAX = 3;   // в стеку максимум 3 цілих числа
//********************************************************************************
class Stack
{ private:
    int st[MAX];        // стек: масив цілих чисел
    int top;            // індекс вершини стеку
  public:
    class Full { };     // клас виключення «Переповнення стеку»
    class Empty { };    // клас виключення «Порожній стек»
    Stack()             // конструктор
      { top = -1; }
    void push(int var)          // занести число в стек
    { if(top >= MAX-1)          // якщо стек повний,
    throw Full();               // генерувати виключення Full
    st[++top] = var; }
    int pop()                   // взяти число зі стеку
    { if(top < 0)               // якщо стек пустий,
    throw Empty();              // генерувати виключення Empty
      return st[top--]; }               };
//********************************************************************************
int main()
{    Stack s1;
    try
```

```
   { s1.push(11);
     s1.push(22);
     s1.push(33);
   // sl.push(44);                              // Стек заповнено!
     cout  <<  "1: “ << s1.pop() << endl;
     cout  <<  "2: “ << s1.pop() << endl;
     cout  <<  "3: “ << s1.pop() << endl;
     cout  <<  "4: “ << s1.pop() << endl;   // Стек порожній!
   }
   catch(Stack::Full)
   { cout << "Помилка: Переповнення стеку" << endl; }
   catch(Stack::Empty)
   { cout << "Помилка: Порожній стек " << endl; }
   return 0;                        }
```

Коли існує багато помилкових умов, може бути зручним використання ієрархії класів для створення набору зв’язаних типів, які можна використовувати у якості виразу встановлення виключення.

```
class Object_Error
{ public:
   Object_Error(<аргументи>);              // Конструктор з параметрами
// < атрибути, що містять опис встановленого виключення >
   virtual void repair()
   { cerr << “Відновлення неможливе “ << endl; abort(); }
};

class Object_Error_S1: public Object_Error
{ public:
   Object_Error_S1(<аргументи>);       // Конструктор з параметрами
// < додані атрибути, що містять опис встановленого виключення >
   void repair();                                // перевизначення, для забезпечення
                                                 // відповідного відновлення
};
. . .     // Інші породжені класи – при необхідності
```

Такі ієрархії дозволяють відповідній впорядкованій множині обробників **catch** обробляти виключення в логічній послідовності. Тип базового класу у переліку **catch** необхідно розміщувати після типу породженого класу.

Порядок вибору обробника catch

З урахуванням сказаного можна визначити порядок, у якому обирається обробник **catch** для встановленого виключення відповідного типу.

Вираз **throw** відповідає аргументу **catch**, якщо він:

– точно відповідає;

– загальний базовий клас породженого типу представляє собою те, що встановлюється;

– типом об’єкта встановленого типу є покажчиком, який може бути перетворений у тип покажчика, що є аргументом **catch**.

Запис обробників у порядку, який запобігає їхньому виклику може бути помилковим. Наприклад:

```
catch (void *)                  // відповідає будь-якому char *
catch(char*)                    // ніколи не викликається
catch(BaseTypeError&)           // буде викликатися завжди для DerivedTypeError
catch(DerivedTypeError&)        // ніколи не викликається
```

Клас bad_alloc

В стандарт C++ включено декілька вбудованих класів виключень. Найчастіше у програмах використовується клас **bad_alloc**, який генерує виключення при виникненні помилки виділення пам’яті оператором **new**. В деяких версіях C++ це виключення має назву **xalloc**. Наступна програма показує приклад штучного виклику і обробки виключної ситуації **bad_alloc**, що дає можливість продовжувати виконання програми при невдалій спробі отримати від системи занадто багато додаткової пам’яті.

Приклад 2.32:

```
#include <iostream>
using namespace std;
long const _SIZE = 100000;                  // розмір одного блоку пам’яті

int main()
{   char* ptr[_SIZE];                       // масив покажчиків на блоки пам’яті
    long i;
    int er=0;

    try
    { for( i=0; i<_SIZE; i++ )
                ptr[i] = new char[_SIZE];   }       // запит SIZE байтів

    catch(bad_alloc)          // обробник виключення bad_alloc
    {     cout << "\nBad_alloc: неможливо виділити пам'ять.\n";
          er=1;  }

    if( er )
    { cout << "\n Помилка! Кількість блоків = " << i << endl;
```

```
    for( long k=0; k<i; k++)
        delete [ ] ptr[k];                          // вивільнення пам’яті
    }
    else
    { cout << "\n Пам'ять виділено.\n";
    for( long k=0; k<_SIZE; k++)
        delete [ ] ptr[k];                          // вивільнення пам’яті
    }
    return 0;
}
```

Всі оператори, якимось чином пов’язані з **new**, розміщуються в блоці повторних спроб **try**. Обробник **catch**, який записано безпосередньо за цим блоком, обробляє помилку, тобто виведе відповідне повідомлення. Далі вивільняється виділена пам'ять і програма завершується (з помилкою чи без).

Відновлення після помилок

Відновлення при виникненні помилок – головним чином має відношення до правильності написання програми. Обробка виключень має безпосереднє відношення і до відновлення при виникненні помилок і до механізму передачі управління. Програміст, який розробляє програмний продукт, повинен гарантувати, що при задовільному вхідному стані його програмне забезпечення виконає правильний вивід. Як реалізовано викриття і виправлення помилок – справа розробника.

Відновлення при виникненні помилок базується на передаванні управління. Неконтрольована передача управління веде до хаосу. При відновлені припускається, що виключення порушило обчислення. Продовжувати обчислення стає небезпечним. Корисна обробка виключень тягне за собою упорядковане відновлення при виникненні помилки.

В більшості випадків програма, яка спричиняє виключення, повинна виводити діагностичне повідомлення і коректно завершити роботу. При спеціальних формах обробки, наприклад, при роботі в режимі реального часу та при обчисленнях, які не повинні мати помилок, існує необхідність в тому, щоб система не призупинялась. В таких випадках спроби відновлення цілком виправдані.

Можливо узгодити, для яких класів корисно передбачати помилкові умови. В більшості випадків – це такі умови, коли об’єкт отримує значення атрибутів з діапазону заборонених станів – тобто значення, які йому не дозволено мати. Для

таких випадків система встановлює виключення із обробкою за умовчанням, яке полягає у завершенні програми. Це аналогічно вбудованим типам, які встановлюють виключення, що визначені системою, наприклад, такими як **SIGFPE** (помилка в операціях з числами у форматі із плаваючою крапкою).

Внутрішня суперечливість підіймає наступні питання: який вид втручання треба вважати допустимим для продовження виконання програми, і визначення, куди необхідно передавати управління? C++ використовує модель завершення, яка примушує завершувати поточний блок **try**. При такому режимі можливо або повторити код, ігноруючи виключення, або підставити результат за умовчанням и продовжити виконання. При повторенні коду більш ймовірно отримати правильний результат.

Досвід показує, що зазвичай код слабо коментується. Важко уявити програму, в якій використовується занадто багато підтверджень. Підтвердження і прості встановлення обробки виключень, які завершують обчислення, представляють собою паралельні технології. Добре спланований набір помилкових умов, які викриваються класом – важлива частина проекту. Якщо при нормальному виконанні програми занадто часто з’являються помилки і виникають переривання – це ознака того, що програму недостатньо обмірковано і вона мала забагато прорахунків на початковій стадії розробки.

2.7. Області дії та простори імен

2.7.1. Роздільна компіляція

В C++ підтримується декілька способів зберігання даних в пам'яті. Програміст має можливість обирати, як довго дані будуть зберігатися в пам'яті (*класи пам'яті*) і те, які частини програми будуть мати доступ до даних (*діапазон доступу* і *зв'язування*). *Простори імен* C++ забезпечують додатковий контроль над доступом до даних. Великі програми зазвичай складаються з декількох файлів з вихідним кодом. Такі програми можуть спільно використовувати деякі дані. Оскільки в таких програмах застосовується *роздільна компіляція* файлів програми, доцільно розпочати саме з цього питання.

Сучасні мови програмування підтримують створення великих програмних продуктів, які складаються з багатьох десятків, або навіть тисяч класів. Програма, що розроблена мовою C++, може бути написана колективом програмістів протягом декількох років. Далі розглянуто властивості мови, які дозволяють створювати великі програми.

Програма – це перш за все текст, написаний певною мовою програмування. За допомогою компілятора і редактору зв'язків текст перетворюється у ***виконуваний код*** – форму, яка дозволяє комп'ютеру виконувати програму.

Якщо розглянути цей процес більш ретельно, то можна з'ясувати, що обробка вихідних файлів виконується в три етапи. *Спочатку* файл обробляється *препроцесором*, який виконує директиви **#include**, **#define** і ще декілька інших. Після цього етапу програма все ще представлена в вигляді текстового файлу, хоча й зміненого порівняного з початковим. Потім, на *другому етапі*, *компілятор* створює так званий ***об'єктний файл***. Програму вже переведено у машинні інструкції, проте вона ще не повністю готова до виконання. В об'єктному файлі знаходяться посилання на різні системні функції і на стандартні функції мови програмування. Наприклад, виконання операції **new** полягає у виклику повної системної функції. Навіть якщо в програмі в явному вигляді не записано жодної функції, необхідним є, як мінімум, один виклик системної функції, яка завершує програму і вивільняє всі надані їй ресурси.

На *третьому етапі* до об'єктного файлу приєднуються всі функції, на які він посилається. Ці функції також мали пройти етап трансляції, тобто були переведеними на машину мову в форматі об'єктних файлів. Цей процес називається ***компоновкою*** і його результат – це ***виконуваний файл***.

Системні функції і стандартні функції мови С++ у формі об'єктних модулів (після компіляції) доступні для використання у вигляді бібліотек. ***Бібліотека*** – це, можна сказати, архів об'єктних модулів, з якими зручно компонувати програму.

Головна мета багатоетапної компіляції програм – можливість компонувати програму з багатьох файлів. Кожний файл являє собою деякий закінчений фрагмент програми, який може посилатися на функції, змінні або класи, які визначені в інших файлах. Компоновка об'єднує фрагменти в одну програму, яка є «самодостатньою», тобто містить все необхідне для виконання.

Сучасні системи програмування мають засоби автоматизації процесу компоновки. Наприклад, в системі Microsoft Visual C++ достатньо лише включити потрібні файли в один проект для того, щоб вони компонувались разом. В системі UNIX, як правило, необхідно більше ручної праці для компонування багатьох файлів в один виконуваний. За повними інструкціями треба звернутися до літератори з конкретної системи програмування.

Мова програмування С++, як і С, забезпечує можливість (більш того, навіть заохочує програміста) розміщувати функції, що утворюють програму, в окремих файлах. Після цього, можна компілювати файли окремо один від одного, і потім зв'язувати їх на завершальному етапі в єдину програму. Якщо змінюється тільки один файл, то можна перекомпілювати тільки його і зв'язати із раніше відкомпільованими версіями інших файлів. Цей механізм спрощує роботу з великими програмами. Крім того, в більшості середовищ С++ забезпечує додаткові засоби, які спрощують управління. В Unix і Linux, наприклад, мається програма (утиліта) **make**, яка відслідковує, від яких файлів залежить програма, а також те, коли в цих файлах останній раз виконувались зміни. Якщо при запуску утиліти **make** вона з'ясує, що один або декілька вихідних файлів з часу останньої компіляції було змінено, то вона, у відповідності із збереженою схемою компіляції створює нову версію перекомпільованої програми. Середовища Borland C++, Microsoft C++ та Metrowerks CodeWarrior IDEs (інтегровані середовища розробки) надають в розпорядження користувача схожі засоби, доступ до яких можливий із меню **Project** (Проект).

Основні положення роздільної компіляції будуть розглянуті на прикладі простої програми перетворення Декартових координат у полярні. Деталі реалізації, які залежать від конкретної версії компілятора не розглядатимуться, а

головну увагу буде зосереджено на найбільш загальних аспектах процесу проектування багатомодульної програми.

Компонентами програми, що розглядається, є оголошення і визначення структурних типів, функцій користувача і головної функції програми – функції **main()**. Нехай, текст програми потрібно розділити таким чином, щоб визначення функцій користувача розміщувались в одному файлі, а функція **main()** в іншому. Неможливо просто «розрізати» вихідний текст програми на дві частини тому, що в функції **main()** і в функціях користувача використовуються ті ж самі оголошення структур. Тому необхідно помістити такі оголошення в обидва файли функцій. Проте просте копіювання цих оголошень в усі файли, де вони застосовуються, у подальшому може призвести до помилки. Навіть якщо оголошення структури буде скопійовано правильно, дуже легко забути, що при змінах структур оновлення необхідно вносити в обидва набори оголошень. Тобто простий механічний розділ тексту програми на декілька файлів породжує нові проблеми.

В C і C++ для вирішення цієї проблеми використовується директива препроцесору **#include**. Замість того щоб розміщувати оголошення структур в кожному файлі, їх можна записати у *файл-заголовок*, а після цього включити цей файл-заголовок в кожний файл з вихідним кодом. В такому разі, щоб змінити оголошення структури, необхідно буде зробити це тільки один раз у файлі-заголовку. Крім того, в файл-заголовок можна записувати і прототипи функцій. Таким чином, вихідну програму можна розділити три частини:

– *файл-заголовок* із оголошеннями структур і прототипів функцій, в яких використовуються ці структури;

– *файл з вихідним кодом функцій*, які працюють із структурами;

– *файл з вихідним кодом*, в якому виконується виклики цих функцій.

Таку стратегію можна успішно застосовувати для організації складних програм. Якщо, наприклад, необхідно написати іншу програму, в якій використовуються ті ж самі функції, достатньо включити до неї файл-заголовок і додати файл із функціями до проекту (або в список **make** для UNIX). До того ж така організація програми відповідає принципам ООП. Перший файл, а саме – заголовок, містить визначення типів користувача (класів). Другий файл містить код функцій для управління типами, що визначені користувачем. Разом обидва файли утворюють *пакет*, який можна використовувати в різних програмах.

До файла-заголовка не слід включати визначення функцій або оголошення змінних: якщо для простих програм це не створить ніяких проблем, то у більш складних випадках проблеми можливі. Наприклад, якщо у файлі-заголовку містяться визначення функцій, і цей файл включено в два інших файли, які є складовими однієї програми, то у одній програмі можуть опинитися два визначення тієї ж самої функції, що вважається помилкою, якщо ця функція не є вбудованою (**inline**).

У файлах-заголовках, як правило, розміщуються:

– прототипи функцій;
– символьні константи визначені директивою **#define** або **const**;
– оголошення структур;
– оголошення класів;
– оголошення шаблонів;
– вбудовані (**inline**) функції.

Розміщувати оголошення структур у файлах-заголовках дуже зручно, тому що вони не створюють змінні; вони тільки повідомляють компілятор, як створювати змінну типу структури, коли вона оголошується у файлі з вихідним кодом.

Аналогічно оголошення шаблонів – це не код, який необхідно компілювати, а інструкції для компілятора, які вказують, як генерувати визначення функцій і класів, щоб вони відповідали викликам шаблонних функцій і оголошенням шаблонних класів, які зустрічаються у вихідному коді. Даним, які оголошено із специфікатором **const**, і вбудованим функціям притаманні спеціальні можливості зв’язування. Завдяки цим можливостям розміщення таких даних і функцій в файлах-заголовках не призводить до будь-яких проблем.

Код програми перетворення Декартових координат у полярні у вигляді трьох окремих файлів одного проекту наведено у прикладі 2.33. Треба звернути увагу, що при включенні в програму файла-заголовка його назву записано у вигляді '**coordin.h**' замість **<coordin.h>**. Якщо ім’я файлу записано в кутових дужках, компілятор C++ шукає цей файл в тій частині файлової системи, в якій розташовано стандартні системні файли-заголовки. Але, якщо ім’я файла загорнуто у парні лапки, компілятор спочатку шукає файл у поточному робочому каталозі або у каталозі з вихідним кодом (або в іншому подібному місці, що залежить від конкретного компілятора); і тільки після цього пошук буде продовжуватися серед стандартних файлів-заголовків. Таким чином, при

включенні в програму власних файлів-заголовків необхідно використовувати лапки, а не кутові дужки.

Приклад 2.33:

Файл-заголовок:

```
// coordin.h  -  оголошення структур і прототипів функцій
// Структури:
#ifndef COORDIN_H_
#define COORDIN_H_
struct polar
{ double distance;            // відстань від початку координат
  double angle;       };      // напрямок від початку координат (кут)

struct rect
{ double x;             // відстань від початку
                        // координат по горизонталі
  double y;    };       // відстань від початку координат по вертикалі прототипи
// Прототипи:
polar rect_to_polar(rect xypos);
void show_polar(polar dapos);
#endif
```

Файл з текстом функції **main()**:

```
// file1.cpp - головна функція main ()
#include <iostream>
#include "coordin.h"   // оголошення структур і прототипів функцій
using namespace std;
int main ()
{ rect rplace; polar pplace;
  cout << "Enter the x and y values: ";
  while (cin >> rplace.x >> rplace.y)       // якщо ввести не число, cin повертає false
  { pplace = rect_to_polar(rplace); show_polar(pplace);
    cout << "Next two numbers  (q to quit):  " ;            }
  cout << "Bye!\n";
  return 0;             }
```

Файл з текстом функцій користувача:

```
// file2.cpp
// функції, що викликаються в файлі file1.cpp
#include <iostream>
#include <cmath>
#include "coordin.h"    // оголошення структур і прототипів функцій
using namespace std;

// перетворення прямокутних координат у полярні
```

```
polar rect_to_polar(rect xypos)
{ polar answer;
  answer.distance = sqrt  (xypos.x * xypos.x + xypos.y * xypos.y);
  answer.angle = atan2(xypos.y, xypos.x);
  return answer;       }         // результат - полярні координати

// вивід полярних координат з приведенням кутів до градусів
void snow_polar(polar dapos)
{ const double Rad_to_deg = 57.29577951;
  cout << "distance = " << dapos.distance;
  cout << ", angle = " << dapos.angle * Rad_to_deg;
  cout << " degrees\n " ;
}
```

На рисунку 2.9 схематично показані дії, які необхідно виконати, щоб відкомпілювати програму та отримати виконуваний файл. Треба звернути увагу, що при роботі в UNIX компіляція і компоновка виконується однією командою (**gcc**). Засоби розробки Microsoft Visual C++, по суті, виконують те ж саме, проте ці дії ініціюються по-іншому – переважно за допомогою елементів меню, які дозволяють створювати проекти і асоціювати з ними файли програмних кодів.

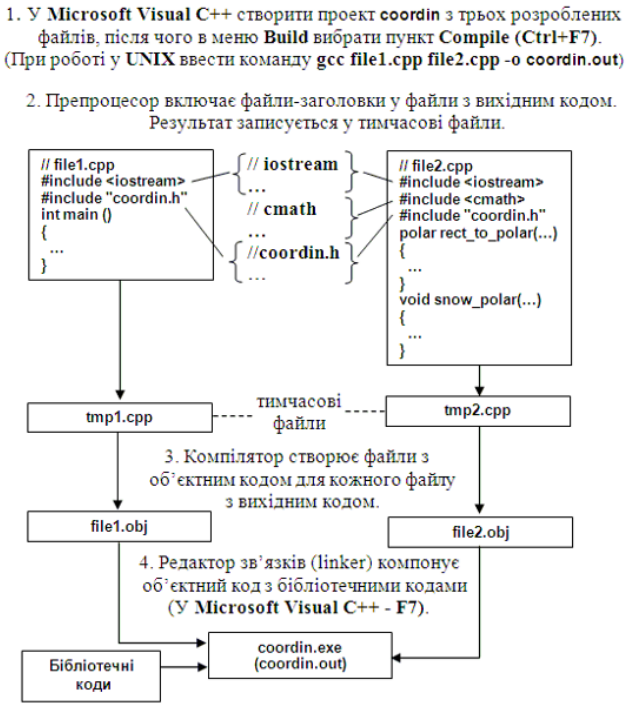


Рисунок 2.9 – Компіляція та компоновка багатофайлової програми

Управління файлами-заголовками

Файл-заголовок необхідно включати в файл з текстом програми тільки один раз. Це просте правило дуже легко порушити і несподівано включити файл-заголовок у код декілька разів, навіть не підозрюючи цього. Наприклад, можна створити файл-заголовок, до складу якого входить інший файл-заголовок. Стандартна для C/C++ методика, яка дозволяє робити можливими багаторазові включення файлів-заголовків, базується на застосуванні директиви препроцесору **#ifndef** («якщо не визначено»). Так, в наведеному нижче фрагменті кода

```
#ifndef COORDIN_H_
        …
#endif
```

оператори, які розташовані між директивами **#ifndef** та **#endif**, обробляються тільки в тому випадку, коли ім'я **COORDIN_H_** не було визначено раніше директивою препроцесору **#define.**

Як правило оператор **#define** використовується для створення символьних констант, наприклад, наступним чином:

```
#define MAXIMUM 4096
```

Також достатньо просто використовувати **#define** для визначення константних імен, як показано нижче:

```
#define COORDIN_H_
```

Методика, яку застосовано в прикладі 2.33, передбачає загортання вмісту файлу в оболонку **#ifndef**:

```
#ifndef COORDIN_H_
#define COORDIN_H_
   //  в це місце включається вміст файлу
#endif
```

Коли компілятор (а точніше, його препроцесор) вперше переглядає файл, ім'я **COORDIN_H_** ще не повинно бути визначено (ім'я утворено з імені файлу та декількох символів підкреслення таким чином, щоб отримане ім'я не було схожим з іншими іменами, які могли бути визначені десь в іншому місці). У такому випадку компілятор буде шукати це ім'я між директивами **#ifndef** та **#endif**, що й потрібно. В процесі пошуку компілятор зчитує рядок, в якому визначається **COORDIN_H_**. У подальшому, якщо зустрінеться ще одне

включення **coordin.h** в той же файл, компілятор відмічає, що ім’я **COORDIN_H_** вже визначено, і переходить до рядку після **#endif**. Таким чином буде проігноровано вміст всіх файлів, крім першого. Така схема використовується в більшості стандартних файлів-заголовків C і C++.

2.7.2. Класи пам'яті, діапазони доступу та зв’язування

Після обговорення програми, що складається з декількох файлів, можна перейти до розгляду схем розподілення пам’яті, тому що категорії пам’яті і спільне використання даних різними файлами щільно зв’язані між собою. В C++ використовується три різних *класи пам’яті* (*схеми зберігання даних*), які різняться між собою тим, як довго дані зберігаються в пам’яті.

Змінним, які оголошено всередині визначення функції (до них відносяться і параметри функції), призначається *автоматичний клас пам’яті*. Такі змінні створюються, як тільки програма передає управління функції або блоку, в якому вони визначені, а надана їм пам’ять вивільняється, коли управління передається за межі функції або блока.

Змінні, які визначено поза межами будь-якої функції або з використанням ключового слова **static,** мають *статичний клас пам’яті.* Вони існують протягом всього часу виконання програми.

Пам’ять, яку виділено оператором **new**, зберігається поки її не буде вивільнено за допомогою оператору **delete** або поки програма не завершиться - в залежності від того, яка подія настане раніше. Ця пам'ять відноситься до класу *динамічної пам’яті* (іноді застосовується термін «вільне сховище» (**free store**)).

Клас пам’яті змінних безпосередньо впливає на інші, не менш важливі їх характеристики, які пов’язані з тим, коли змінні різних типів знаходяться в *діапазоні доступу* (області бачення), тобто можуть використовуватися програмою, а також *зв’язування*, що визначає, які дані спільно використовуються різними файлами програми.

Діапазон доступу і зв’язування

Діапазон доступу визначає, наскільки «далеко» конкретне ім’я можна «бачити» в файлі (в одиниці трансляції). Наприклад, змінну, яку визначено в функції, можна використовувати тільки в цій функції, а не в будь-якій іншій,

проте змінна, яку визначено в файлі перед визначенням всіх функцій, може бути задіяна в усіх функціях.

Зв'язування визначає певний спосіб спільного використання імені в різних одиницях трансляції. Ім'я з *зовнішнім зв'язуванням* може використовуватися різними файлами, а ім'я із *внутрішнім зв'язуванням* – функціями в межах одного файлу. Імена автоматичних змінних (яким призначено автоматичний клас пам'яті) не мають жодного зв'язування, тому для них не припускається спільне використання в програмі.

Змінна в C++ може мати один з декількох можливих *діапазонів доступу*. Змінна, яка має *локальний діапазон доступу* (***область бачення блоку***), відома тільки в межах того блоку, в якому її визначено. Блок – це група операторів між парою фігурних дужок. Наприклад, блоком вважається тіло функції, проте в тіло функції також можуть бути вкладені і інші блоки. Змінна, яка має ***глобальний діапазон доступу*** (його ще називають ***областю бачення файлу***), відома в усьому файлі, починаючі з місця її визначення. Автоматичні змінні мають локальний діапазон доступу, а статичні змінні можуть мати різний діапазон доступу, в залежності від того, як їх було визначено. Імена, які використовуються в ***діапазоні доступу прототипу функції***, відомі тільки в межах круглих дужок, які містять список аргументів (тому не має значення, що вони собою представляють, неважливо, чи є вони, або їх немає). Елементи, які оголошені в класі, мають ***діапазон доступу класу***. Змінні, які оголошено в просторі імен, мають ***діапазон доступу простору імен*** (***глобальний діапазон доступу*** є частковим випадком діапазону доступу простору імен).

Функції C++ можуть мати *діапазон доступу класу* або *діапазон доступу простору імен*, включаючи *глобальний діапазон доступу*, проте не можуть мати локального діапазону доступу. (Функцію неможна визначати всередині блоку, тому що, якщо б вона мала локальний діапазон доступу, вона була б відома тільки для самої себе, і, як наслідок, її неможливо було б викликати з іншої функції).

В C++ різні способи зберігання даних розрізняються по притаманним їм *часу зберігання* в пам'яті, *діапазону доступу* і *зв'язуванню*. З тієї точки зору далі розглядаються властивості класів пам'яті змінних C++.

Автоматичний клас пам'яті

Параметрам функцій і змінним, які оголошені всередині функцій, за умовчанням призначається ***автоматичний клас пам'яті***. Вони характеризуються також локальним діапазоном доступу і не мають зв'язування. Це означає, що при оголошенні двох змінних з одним і тим же ім'ям **day** – однієї в функції **main(),** а другої в функції **fun()**, створюються дві незалежні змінні, кожна з яких буде відома тільки в тій функції, в якій цю змінну оголошено. Те, що буде зроблено зі змінною **day** в функції **fun()**, ніяким чином не впливатиме на змінну **day** в **main(),** и навпаки. Кожна змінна розміщується в пам'яті після того, як починає виконуватися та функція, в якій цю змінну визначено. Така змінна припиняє своє існування після того, як виконання її функції завершується (точніше, змінна розміщується в пам'яті, коли починається виконуватися блок, в якому її визначено, проте область бачення починається тільки після точки оголошення).

При визначенні змінної всередині блоку час її існування і область бачення обмежені цим блоком. Наприклад, якщо на початку функції **main()** було оголошено змінну **tel**, а всередині **main()** створюється новий блок і в ньому визначено змінну з ім'ям **sign**, то змінну **tel** можна бачити всюди в **main()**, а змінну **sign** – тільки у внутрішньому блоці. Змінна **sign** буде існувати з моменту свого визначення тільки доки виконання програми не дійде до кінця блоку (приклад 2.34).

Приклад 2.34:

```
int main ()
{    int tel = 5;
   {                              // змінна sight розміщується в пам'яті
      cout << "Hello\n";
      int sign = -2;              // початок області дії змінної sign
      cout << sign << ' ' << tel << endl;
   }                              // термін дії змінної sign закінчується
   cout << tel << endl;           // змінна tel все ще існує
}
```

А тепер можна подивитися, що вийде, якщо у внутрішньому блоці замість змінної **sign** визначити ще одну змінну **tel** (приклад 2.35). В результаті в програмі опиниться дві змінні з одним і тим же ім'ям, одна у зовнішньому блоці, а інша – у внутрішньому. У такій ситуації програма інтерпретує ім'я **tel** як змінну, що є локальною по відношенню до блоку під час виконання операторів в цьому блоці. В таких випадках говорять, що нове визначення приховує старе. Нове визначення

потрапляє в діапазон доступу, в той же час як старе з нього вилучається. Коли програма передає управління за межі блоку, до діапазону доступу повертається перше визначення.

Приклад 2.35:

```
int main ()
{    int tel = 5;
   {                          // в пам'яті розміщується нова змінна tel
      cout << "Hello\n";
      int tel = -2;           // початок дії нової змінної tel
      cout << tel << endl;
   }                          // термін дії нової змінної tel закінчується
   cout << tel << endl;       // змінна tel знову має значення 5
}
```

Програма у наступному прикладі показує, як автоматичні змінні локалізуються у функції або блоці, в яких їх визначено.

Приклад 2.36:

```
#include <iostream>
using namespace std;
void fun(int x);

int main()
{    int day = 31;
   int year = 1999;
   cout << "In main(), day = " << day <<  ",  &day =";
   cout << &day << "\n";
   cout << "In main(), year = " << year << ",  &year =";
   cout << &year <<  "\n";
   fun (day) ;
   cout << "In main(),  day = " << day << ",  &day =" ;
   cout << &day << "\n";
   cout << "In main(), year = " << year << ",  &year =";
   cout << &year << "\n";
   return 0;            }

void fun (int x)
{    int day = 5;
   cout << " In fun(),  day = " << day << " &day =";
   cout << &day <<  "\n" ;
   cout << " In fun(), x = " << x << ",  &x =";
   cout << &x << "\n";

   {    // початок блоку
        int day = 113;
```

```
        cout << "  In block,  day = " << day;
        cout << ",  &day = " << &day << "\n";
        cout << "  In block, x = " << x << ", &x =";
        cout << &x << "\n";
    }     // кінець блоку

    cout << " Post-block day = " << day;
    cout << ", &day = " << &day << "\n";            }
```

Результати виконання програми:

```
In main(), day = 31,  &day =0x6dfefc
In main(), year = 1999,  &year =0x6dfef8
 In fun(),  day = 5 &day =0x6dfecc
 In fun(), x = 31,  &x =0x6dfee0
  In block,  day = 113,  &day = 0x6dfec8
  In block, x = 31, &x =0x6dfee0
 Post-block day = 5, &day = 0x6dfecc
In main(),  day = 31,  &day =0x6dfefc
In main(), year = 1999,  &year =0x6dfef8
```

Як можна бачити, кожна з трьох змінних **day** має свою власну, відмінну від інших, адресу, а програма використовує лише ту змінну, яка у поточний час знаходиться у діапазоні доступу. Таким чином, змінній **day** у внутрішньому блоці функції **fun()** привласнюється значення **113**, що ніяк не впливає на інші змінні з тим же ім’ям.

На момент початку роботи **main()** програма виділяє пам'ять для змінних **day** та **year**, і обидві ці змінні потрапляють у діапазон доступу. В момент, коли програма звертається до функції **fun()**, ці змінні залишаються в пам’яті, але зникають з діапазону доступу. Дві нові змінні, **x** та **day**, також розміщуються в пам’яті і потрапляють до діапазону доступу. Коли програма доходить до внутрішнього блоку в функції **fun()**, нова змінна **day** виходить з області бачення і виштовхується більш новим визначенням. Змінна **x** залишається в діапазоні доступу, тому що в блоці не визначено нової змінної **x**. Коли управління передається за межі цього блоку, пам’ять для самої нової змінної **day** вивільняється, і **day** номер **2** знову повертається до діапазону доступу. Як тільки виконання функції **fun()** завершується, її змінні **day** і **x** припиняють існування, а до діапазону доступу повертається первинні змінні **day** і **year**.

Доречи, можна за допомогою ключового слова **auto** мови C++ (і C) вказувати клас пам’яті явно:

```
int fun(int n)
{       auto float x1;
...
}
```

Ключове слово **auto** можна використовувати тільки при роботі зі змінними, які є автоматичними за умовчанням, тому програмісти майже ніколи його не застосовують. Час від часу воно використовуються для того, щоб код був більш зрозумілим користувачеві. Наприклад, можна спеціально скористатися цім словом, якщо необхідно показати, що створюється автоматична змінна, яка перекриває глобальне визначення.

Ініціалізація автоматичних змінних

Ініціалізувати автоматичну змінну можна будь-яким виразом, значення якого буде відомим в момент, коли виконання програми дійде до її оголошення.

```
int w;          // змінна w не має значення
int x = 5;      // ініціалізація за допомогою константного виразу
int y = 2 * x;  // використання раніше визначеної змінної x
cin >> w;
int z = 3 * w;  // використання введеного значення w
```

Автоматичні змінні і робота зі стеком

Для отримання більш повного уявлення про автоматичні змінні необхідно знати, як компілятор мови C++ формує такі змінні.

Кількість автоматичних змінних під час роботи програми постійно змінюється внаслідок нетривалого часу їх життя. Окремі функції починаючи і закінчуючи свою роботу викликають створення і знищення автоматичних змінних. В таких умовах програма повинна управляти автоматичними змінними в процесі свого виконання. Як правило з цією метою застосовується підхід, який базується на виділенні спеціальної області пам’яті для реалізації – ***стеку***, призначеного для управління створенням і знищенням змінних. Ця область отримала назву «***стек***», тому що нові дані записуються, умовно кажучи, над старими (тобто у суміжні, а не в ті ж самі комірки пам’яті), а після того, як програма завершила роботу з ними, вони видаляються зі стеку. За умовчанням розмір стеку залежить від реалізації, поте у загальному випадку у компілятора є засоби для зміни розміру стеку. Програма відслідковує місцезнаходження стеку за допомогою двох покажчиків. Один з них вказує на *базу стеку*, с якої

починається ділянка відведеної під стек пам’яті, а другий – на *вершину стеку*, яка представляє собою наступну комірку вільної пам’яті. В момент, коли виконується звертання до функції, її автоматичні змінні додаються до стеку, а покажчик вершини встановлюється на вільну комірку пам’яті, наступну за тільки що розміщеними змінними. Коли виконання функції припиняється, покажчик вершини знову отримує значення, яке він мав перед звертанням до функції, в результаті чого вивільняється пам’ять, яка використовувалась для зберігання нових змінних.

Стек реалізує архітектуру **LIFO** (**last in** – **first out** – останнім увійшов, першим вийшов). Це означає, що змінна, яка потрапила у стек останньою, вилучається з нього першою. Такий механізм спрощує передачу аргументів. При виклику функції значення її аргументів розміщуються у вершині стеку і виконується перевстановлення покажчика вершини. Функція використовує опис своїх формальних аргументів для пошуку адреси кожного аргументу.

Наприклад, на рисунку 2.10 показано функцію **fib()**, якій в момент виклику передаються 4-байтове значення **int** та 8-байтове значення **long**. Ці значення потрапляють до стеку. Коли функція **fib()** починає виконуватися, вона зв’язує імена **r** і **t** з цими двома значеннями. Як тільки виконання функції **fib()** завершується, покажчик вершини стеку відновлює посилання на попередню позицію. Нові значення не видаляються, але тепер вони не зв’язані з аргументами, і ділянка пам’яті, яку вони займають, буде задіяна наступним процесом, під час виконання якого нові значення будуть розміщуватися в стеку (на рисунку представлена дещо спрощена ситуація, тому що при виклику функції можлива передача додаткової інформації, наприклад адреси повертання).

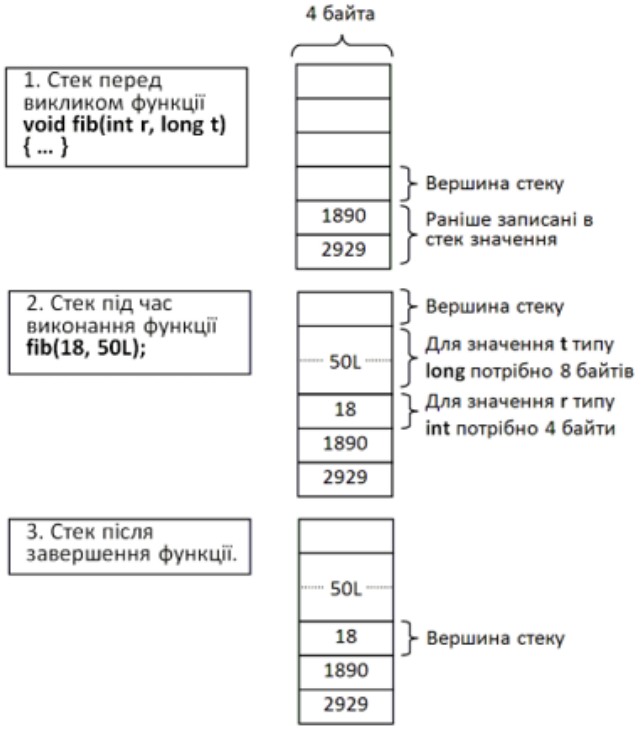


Рисунок 2.10 – Передавання аргументів через стек

Змінні типу register

Мовою C++, як і C, підтримується ключове слово **register**, яке призначене для оголошення локальних ***регістрових*** змінних. Регістрова змінна - це один з видів автоматичної змінної, тому вона характеризується автоматичним класом пам'яті, локальним діапазоном доступу, і не має зв'язування. Ключове слово **register** надає компілятору «*підказку*» про те, що для обробки конкретної змінної замість стеку йому пропонується використовувати регістр центрального процесору. Сенс полягає в тому, що центральний процесор отримує доступ до значень в регістрах значно швидше, ніж виконується доступ до пам'яті в стеку. Для оголошення регістрової змінної треба перед типом змінної вказати ключове слово **register**:

```
register int count_fast;       // запит на створення регістрової змінної
```

Слід пояснити, чому було використано слово «*підказку*», а не «вимогу». Компілятор зовсім не обов'язково повинен виконувати підказку. Наприклад, регістри вже можуть бути зайнятими або запрошується створення типу, який не підходить для регістра. В даний час сучасні компілятори достатньо «розумні», щоб обходитися без будь-яких вказівок. Наприклад, якщо застосовується цикл

for, то компілятор може самостійно прийняти рішення використовувати регістр для зберігання та зміни індексу циклу.

Якщо змінна зберігається у регістрі, то вона не має адреси пам'яті, тобто, до регістрової змінної неможливо застосувати адресні операції:

```
void g(int *)
{       ...     }
int x;
register int y;
g(&x);          // правильно
g(&y);          // не дозволено
```

Наведене обмеження дійсне в будь-якому випадку, тому зазначення ключового слова **register** в оголошенні, не гарантує використання регістру для змінної.

Всі змінні, такі як звичайна локальна змінна, локальна змінна, яку оголошено з використанням **auto**, і локальна змінна, яку оголошено з використанням **register**, мають автоматичний клас пам'яті, локальний діапазон доступу і не мають зв'язування.

```
short w;                // автоматична змінна за умовчанням
auto short p;           // явне оголошення автоматичної змінної
register int m;         // регістрова змінна
```

Оголошення локальної змінної без специфікатора - те ж саме, що й оголошення її зі специфікатором **auto**, і така змінна буде оброблена звичайним чином через розміщення у стеку пам'яті. Застосування специфікатора **register** вказує на те, що змінна буде часто використовуватися, і компілятор може обрати замість стеку що-небудь інше для її зберігання, наприклад, регістр центрального процесору.

Змінні з статичним класом пам'яті

В мові C++, як і в C, реалізовані змінні із ***статичним класом пам'яті*** з трьома типами зв'язування: ***зовнішнім зв'язуванням***, ***внутрішнім зв'язуванням*** і ***без зв'язування***. Змінні всіх трьох типів існують протягом усього часу виконання програми, їм властива більша тривалість, ніж автоматичним змінним. Кількість статичних змінних постійна протягом виконання програми і тому сама програма не має потреби у спеціальних механізмах, подібних стеку, для

маніпулювання ними. Компілятор просто резервує фіксований блок пам'яті для зберігання всіх статичних змінних, і ці змінні доступні програмі протягом усього часу її виконання. Більше того, якщо статичну змінну явно не проініціалізовано, то компілятор встановлює її значення в нуль. Елементи статичних масивів і структур встановлюються рівними нулю за умовчанням.

Далі розглядаються властивості змінних трьох типів із статичним класом пам'яті і процес їх створення. Щоб створити статичну змінну з ***зовнішнім зв'язуванням***, її треба оголосити за межами будь-якого блоку. Щоб створити статичну змінну **з *внутрішнім зв'язуванням***, її оголошують за межами всіх блоків із використанням модифікатора **static**. Для створення статичної змінної ***без зв'язування*** її необхідно оголосити всередині будь-якого блоку з використанням модифікатора **static**. Наступний фрагмент демонструє всі три випадки:

```
int global = 1000;              // статичний клас пам'яті, зовнішнє зв'язування
static int one_file =50;        // статичний клас пам'яті, внутрішнє зв'язування
int main ()
{
        ...
}

void functl(int n)
{   static int count =0;        // статичний клас пам'яті, немає зв'язування
    int llama = 0;
        ...
}

void funct2 (int q)
{       ...     }
```

Всі статичні змінні (в розглянутому фрагменті – це **global**, **one_file** і **count**) починають своє існування з моменту початку виконання програми і до її завершення. Змінна **count**, яку оголошено в функції **functl()**, має *локальний діапазон доступу* і *не має зв'язування*, що означає, що її можна використовувати тільки в функції **functl()**, так же, як і автоматичну змінну **llama**. Проте на відміну від **llama**, змінна **count** залишається в пам'яті, навіть коли функція **functl()** не виконується. Обидві змінні **global** і **one_file** мають ***діапазон доступу файлу (глобальний)***, що означає можливість їх використання від точки оголошення до кінця файлу. Зокрема, обидві змінні можна використовувати в функціях **main()**, **functl()** і **funct2()**. Зважаючи на те, що змінна **one_file** має внутрішнє зв'язування,

її можна використовувати тільки в файлі, де міститься даний код, а **global**, завдяки зовнішньому зв'язуванню, також можна задіяти і в інших файлах програми.

Всі змінні зі статичним класом пам'яті, задовольняють двом особливостям ініціалізації.

1. При оголошенні статичної змінної без ініціалізації всі біти змінної встановлюються в **0**.

2. Для ініціалізації статичної змінної можна використовувати тільки константний вираз.

У константному виразі можуть бути задіяні літеральні константи, константи **const** і **enum**, а також оператор **sizeof**. Наступний фрагмент коду ілюструє ці особливості.

```
int x;                          // змінна x встановлюється в 0
int y = 49;                     // 49 - це константний вираз
int z = 2 * sizeof(int) + 1;    // константний вираз
int m = 2 * z;                  // невірно, z - не константа
int main() {. . .}
```

Далі статичний клас пам'яті розглядається більш ретельно.

Статичний клас пам'яті, зовнішнє зв'язування

Змінні з зовнішнім зв'язуванням також називають ***зовнішніми змінними***. Вони обов'язково мають ***статичний клас*** пам'яті і ***діапазон доступу файлу***. Зовнішні змінні визначаються за межами функцій і тому є зовнішніми по відношенню до будь-якої функції. Наприклад, їх можна оголосити перед функцією **main()**. Зовнішньою змінною можна скористатися в будь-якій функції, яка розміщується в файлі за визначенням зовнішньої змінної. Таким чином, зовнішні змінні вважаються також глобальними по відношенню до автоматичних змінних, котрі є локальними. Тим не менш, якщо в програмі визначено автоматичну змінну з тим же ім'ям, що і зовнішня змінна, саме автоматична змінна потрапляє в діапазон доступу, коли програма виконує цю конкретну функцію. Представлена в прикладі 2.38 програма ілюструє перелічені аспекти. Крім того, в програмі показано, як можна скористатися ключовим словом **extern** для повторного оголошення зовнішньої змінної, яку вже раніше було оголошено, і як використовувати операцію визначення діапазону доступу '**::**' для реалізації

доступу до прихованої в усіх інших випадках зовнішньої змінної. Програма у прикладі 2.37 складається з одного файлу, вона не ілюструє особливості зовнішнього зв'язування; це зроблено в наступному прикладі.

Приклад 2.37:

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

double warming = 0.3;       // зовнішня змінна warming (потепління)

// прототипи функцій
void update(double dt);
void local();

int main()                  // використання глобальної змінної
{     cout << "Global warming " << warming << " degrees.\n";
    update(0.1);            // виклик функції для зміни рівня підігріву
    cout << "Global warming " << warming << " degrees.\n";
    local();                // виклик функції для локального підігріву
    cout << "Global warming " << warming << " degrees.\n";
    return 0;           }

void update(double dt)          // модифікує глобальну змінну
{    //extern double warming;   // необов'язкове оголошення
    warming += dt;
    cout << "Updating global warming to " << warming;
    cout << "degrees.\n";           }

void local()                    // використовує локальну змінну
{    double warming = 0.8;      // скриває зовнішню змінну
    cout << "Local warming = " << warming << " degrees.\n";
    // Доступ до глобальної змінної за допомогою операції "::"
    cout << "But global warming = " << ::warming;
    cout << " degrees.\n";         }
```

Результат виконання програми:

```
Global warming is 0.3 degrees.
Updating global warming to 0.4 degrees.
Global warming is 0.4 degrees.
Local warming = 0.8 degrees.
But global warming = 0.4 degrees.
Global warming is 0.4 degrees.
```

Примітки до програми

Результати виконання програми свідчать про те, що обидві функції, **main()** і **update()**, мають доступ до зовнішньої змінної **warming**. Значення, які функція **update()** привласнює змінній **warming**, проявляються у наступних випадках застосування цієї змінної.

В функції **update()** міститься повторне оголошення змінної **warming** з використанням ключового слова **extern**. Це ключове слово означає «*використовувати змінну з даним ім'ям, яку визначено раніше, як зовнішню*». Дане оголошення не є обов'язковим, тому що при його відсутності функція **update()** працює так же. Його можна розглядати лише як документальну констатацію того факту, що функцію розроблено для використання даної зовнішньої змінної. Перше оголошення

```
double warming = 0.3;
```

називається ***описовим оголошенням*** або просто ***визначенням***. В результаті під конкретну змінну розподіляється пам'ять. Повторне оголошення тієї ж змінної

```
extern double warming;
```

називається ***посилальним оголошенням*** або просто ***оголошенням***. Воно не виконує виділення пам'яті, тому що посилається на існуючу змінну. Ключовим словом **extern** можна скористатися тільки при оголошенні змінної в будь-якій іншій частині програми (або в інших файлах). До того ж у посилальних оголошеннях забороняється ініціалізація змінних:

```
extern double warming = 0.5;      // НЕПРАВИЛЬНО
```

Ініціалізація змінної в оголошенні допускається тільки за умови, що оголошення розміщує цю змінну в пам'яті, тобто при описовому оголошенні. Врешті решт, термін «*ініціалізація*» означає занесення значення у ділянку пам'яті при виділенні цієї ділянки для конкретної змінної.

На прикладі функції **local()** можна бачити, що при оголошенні локальної змінної с тим же ім'ям, що й у глобальної, наявність локальної версії приховує глобальну версію цієї змінної. В функції **local()**, наприклад, при виведенні значення змінної **warming** використовується її локальна версія.

В мові C++ у порівнянні з C засоби опису змінних отримали додатковий розвиток: в C++ підтримується *операція визначення діапазону доступу* '::' (*кваліфікатор області бачення*). Коли ця операція записується перед ім'ям змінної, використовується її глобальна версія. Таким чином, функція **local()** виведе значення **warming** як 0,8, а **::warming** – як 0,4.

Більш детально операція визначення області бачення розглядається нижче.

В таблиці 2.2 представлено зведені дані щодо властивостей класів пам'яті при їх використанні без просторів імен.

Таблиця 2.2 – П'ять способів зберігання змінних

Спосіб зберігання	**Клас пам'яті**	**Діапазон доступу**	**Зв'язування**	**Де і як оголошено**
Автоматичний	Автоматичний	Блок	Немає	В блоці (необов'язково з ключовим словом **auto**)
Регістровий	Автоматичний	Блок	Немає	В блоці з ключовим словом **register**
Статичний без зв'язування	Статичний	Блок	Немає	В блоці з ключовим словом **static**
Статичний з зовнішнім зв'язуванням	Статичний	Файл	Зовнішнє	Поза межами будь-якої функції
Статичний з внутрішнім зв'язуванням	Статичний	Файл	Внутрішнє	Поза межами будь-якої функції з ключовим словом **static**

Локальна чи глобальна змінна?

Перед програмістом завжди може виникнути питання: яким змінним слід надавати перевагу в конкретній програмі - глобальним чи локальним? На перший погляд глобальні змінні здаються більш привабливими. До таких змінних мають доступ всі функції і тому не треба турбуватися про те, як правильно передавати аргументи. Проте такий легкий доступ дістається великою ціною – за рахунок зниження надійності програм. Досвід програмування свідчить, що чим більше дані ізольовано від небажаного доступу, тим краще це впливатиме на їх *цілісність*. В більшості випадків слід користуватися локальними змінними і передавати дані функціям тільки при необхідності, а не дозволяти такий доступ до цих даних, який забезпечують глобальні змінні. Застосування об'єктно-орієнтованого програмування дозволяє зробити ще один крок в напрямку ще більшої ізоляції даних (*інкапсуляція*).

Проте й у глобальних змінних мається своя область застосування. Зокрема, в програмі не виключається існування блоку даних, який використовується декількома функціями, наприклад, масив з назвами місяців або список атомних ваг хімічних елементів. Для представлення постійних даних більш підходить клас зовнішньої пам'яті. В цьому випадку можна також скористатися ключовим словом **const** для захисту даних від змін.

```
const char * const months[12] =
{       "January", "February", "March", "April",
        "May", "June", "July", "August" ,
        "September", "October", "November",
        "December"          };
```

Перше ключове слово **const** захищає від змін рядки, а друге слово **const** гарантує, що кожний покажчик у масиві буде вказувати на той рядок, на який він вказував при створенні масиву.

Статичний клас пам'яті, внутрішнє зв'язування

Застосування модифікатора **static** до змінної з *діапазоном доступу файлу* забезпечує для неї *внутрішнє зв'язування*. Різниця між внутрішнім і зовнішнім зв'язуванням стає більш значущою у багатофайлових програмах. В цьому контексті статична зовнішня змінна є локальною по відношенню до файлу, в якому вона міститься. Для неї є характерним внутрішнє зв'язування. Проте звичайна зовнішня змінна має зовнішнє зв'язування, що означає можливість її використання в інших файлах. При зовнішньому зв'язуванні один і тільки один файл може містить зовнішнє визначення змінної. В усіх інших файлах, в яких буде використовуватися ця змінна, у посилальному оголошенні необхідно записувати ключове слово **extern**. Наприклад:

```
// file1.cpp
#include <iostream>
using namespace std;

// прототипи функцій
#include "mystuff.h"

// визначення зовнішньої змінної
int process_status = 0;

void promise ();
```

```
// file2.cpp
#include <iostream>
using namespace std;

// прототипи функцій
#include "mystuff.h"

// посилання на зовнішню змінну
extern int process_status;

int manipulate(int n)
```

int main()
{
…
}
void promise()
{
…
}
	{
	…
	}
	
	char * remark(char * str)
	{
	…
	}

В цьому файлі визначено змінну **process_status**, що примушує комп'ютер виділити для неї память	В цьому файлі застосовано оголошення з **extern**, щоб повідомити програму про необхідність використовувати змінну **process_status** з іншого файлу

Якщо в файлі відсутнє оголошення **extern** для змінної, в ньому неможливо буде використовувати визначену десь в іншому місці зовнішню змінну:

```
// файл 1
int errors = 20;              // глобальне оголошення
…
//-------------------------------------------------------------
// файл 2
…                             // відсутнє оголошення extern int errors
//
void froobish()
{   cout << errors;           // помилка!
...
```

Якщо в файлі дається визначення другої зовнішньої змінної з тим же ім'ям, то це помилка:

```
// файл 1
int errors = 20;              // зовнішнє оголошення
…
//-------------------------------------------------------------
// файл 2
int errors;                   // неправильне оголошення
void froobish()
{   cout << errors;           // невдала спроба скористатися змінною errors
…
```

Правильний підхід полягає у використанні ключового слова **extern** в другому файлі:

```
// файл 1
```

```
int errors = 20;        // зовнішнє оголошення
…
//-----------------------------------------------
// файл  2
extern  int  errors;   // посилання на змінну errors  з  файлу 1
void froobish()
{   cout  <<  errors;  // використання змінної errors з файлу 1
…
```

Проте, якщо в файлі оголошено статичну зовнішню змінну (з внутрішнім зв'язуванням) з тим же ім'ям, що й зовнішня змінна, яку вже оголошено в іншому файлі, то до діапазону доступу цього файлу потрапляє статична версія:

```
// файл 1
int errors = 20;        // зовнішнє оголошення
…
//-------------------------------------------------------
// файл 2
static int errors =5;  // відома тільки файлу 2
void froobish()
{   cout << errors;    // використання змінної errors з файлу 2
…
```

Зауваження: В багатофайловій програмі зовнішню змінну дозволяється оголосити тільки в одному файлі. В усіх інших файлах, де використовується ця змінна, її необхідно оголосити з ключовим словом **extern**.

Для забезпечення спільного використання даних різними частинами багатофайлової програми слід застосовувати звичайні зовнішні змінні. Для спільного використання даних різними функціями, які зосереджено в одному файлі, необхідно використовувати статичну зовнішню змінну (існує альтернативний метод, який забезпечується за допомогою просторів імен, і в стандарті мови C++ вказано, що використання **static** для створення внутрішнього зв'язування у майбутньому буде припинено). Крім того, якщо перевести змінну з діапазоном доступу файлу в категорію статичних, то необхідно буде також попіклуватися про відсутність конфлікту її імені з зовнішніми змінними в інших файлах.

В програмі, представленої у прикладі 2.38, показано, як в C++ виконується обробка зовнішнього і внутрішнього зв'язування. В першому файлі програмі (**twofile1.cpp**) оголошено зовнішні змінні **tom** і **dick**, а також статична зовнішня змінна **harry**. Функція **main()** в цьому файлі виводить адреси цих трьох змінних, а потім звертається до функції **remote_access()**, яку визначено в другому файлі

(**twofile2.cpp**). Крім визначення функцій **remote_access()**, в даному файлі застосовується ключове слово **extern**, щоб забезпечити спільне використання змінної **tom** з першого файлу. Далі в ньому оголошується статична змінна з ім'ям **dick**. Модифікатор **static** робить цю змінну локальною по відношенню до файлу і таким чином перекриває глобальне визначення. Після цього визначається зовнішня змінна з ім'ям **harry**. Це можна зробити без конфлікту зі змінною **harry** з першого файлу, тому що перша **harry** має тільки внутрішнє зв’язування. Після цього функція **remote_access()** виводить адреси трьох перелічених вище змінних, так що мається можливість порівняти їх з адресами відповідних змінних з першого файлу. Для отримання результату обидва файли необхідно скомпілювати і скомпонувати.

Приклад 2.38:

Файл **twofile1.cpp** – зовнішні і статичні зовнішні змінні:

```
#include <iostream>                 // компілюється з file2.cpp
using namespace std;
int tom = 3;                        // визначення зовнішньої змінної
int dick = 30;                      // визначення зовнішньої змінної
static int harry = 300;             // статична змінна, внутрішнє зв’язування

void remote_access();               // прототип функції

int main ()
{    cout << "main() reports the following adresses:\n";
    cout << &tom << " = &tom, " << &dick << " = &dick, ";
    cout << &harry << " = &harry\n";
    remote_access();
    return 0;            }
```

Файл **twofile2.cpp** – змінні з внутрішнім і зовнішнім зв’язуванням:

```
#include <iostream>
using namespace std;
extern int tom;                 // змінну tom визначено в іншому файлі
static int dick = 10;           // перекриває зовнішню змінну dick
int harry = 200;                // визначення зовнішньої змінної,
                                // без конфлікту зі змінною harry
                                // з файлу twofile1.cpp
void remote_access()
{    cout << "remote_access() reports the following adresses:\n";
    cout << &tom << " = &tom, " << &dick << " = &dick, ";
    cout << &harry << " = &harry\n" ;               }
```

Результати виконання програми:

```
main() reports the following addresses:
0x0041a020 = &tom, 0x0041a024 = &dick, 0x0041a028 = &harry
remote_access() reports the following addresses:
0x0041a020 = &tom, 0x0041a050 = &dick, 0x0041a054 = &harry
```

Статичний клас пам’яті, без зв’язування

Вище розглянуто змінні, які мають діапазон доступу файлу, з зовнішнім і внутрішнім зв’язуванням. Залишилося розглянути третій варіант змінних зі статичним класом пам’яті – *локальні змінні, які* ***не мають зв’язування***. Такі змінні створюються шляхом застосування модифікатора **static** до змінної, яку визначено всередині блоку. Модифікатор **static** всередині блоку забезпечує змінній статичний клас пам'яті і *локальний діапазон доступу*. Це означає, що, незважаючи на те, що змінна відома лише в межах цього блоку, вона існує навіть тоді, коли блок неактивний. Таким чином, статична локальна змінна може зберігати своє значення між викликами функції. І ще одне важливе зауваження: якщо в програмі передбачена ініціалізація статичної локальної змінної, то ініціалізація виконується тільки один раз – при запуску програми, а у наступних викликах функції ініціалізація змінної не виконується (на відміну від автоматичних змінних).

Наступний приклад програми демонструє використання локальної статичної змінної без зв’язування.

Приклад 2.39:

```
// Використання статичної локальної змінної
#include <iostream>
using namespace std;
const int ArSize = 10;                  // константа
void strcount(const char * str);        // прототип функції

int main()
{    char input[ArSize];
    char next;
    cout << "Enter a line: \n";
    cin.get(input, ArSize);
    while (cin)
    {     cin.get(next);
          while (next != '\n')    // кінець рядка?
          cin.get(next);
          strcount(input);
```

```
        cout << "Enter next line (empty line to quit):\n";
        cin.get(input, ArSize);                 }
    cout << "Bye\n";
    return 0;             }

void strcount(const char * str)
{    static int total = 0;         // статична локальна змінна
    int count = 0;                 // автоматична локальна змінна
    cout <<  "\"" << str << "\" constants ";
    while (*str++)                 // підрахунок довжини рядка str
    count++;
    total += count;
    cout << count << " characters\n";
    cout << total << " characters total\n";           }
```

Програма зчитує рядок у масив символів. В програмі, крім іншого, представлений один з способів розділення рядка, що вводиться, якщо його довжина перевищує розмір призначеного для нього масиву. Метод вводу **cin.get(input, ArSize)** запитує рядок, зчитує його повністю або до **ArSize-1** символу (в залежності від того, яка подія наставатиме раніше), залишаючи символ нового рядка в черзі вводу. Далі рядок вводиться посимвольно. Якщо зустрінеться символ нового рядка, значить, в поточному рядку не залишилося символів. Враховується також і те, що при спробі ввести порожній рядок метод **get(char *, int)** потоку **cin** повертає **false**.

Результат виконання програми:

```
Enter a line:
nice pants
"nice pants" contains 9 characters
9 characters total
Enter next line (Empty line to quit)
thanks
"thanks" contains 6 characters
15 characters total
Enter next line (Empty line to quit)
parting in such sweet sorrow
"parting i" contains 9 characters
24 characters total
Enter next line (Empty line to quit)
ok
"ok" contains 2 character"
26 characters total
Enter next line (Empty line to quit)
Bye
```

Треба звернути увагу на те, що оскільки розмір масиву дорівнює **10**, програма не зчитує більш ніж **9** символів рядка. Також важливим вважається, що для автоматичної змінної **count** кожний раз при виклику функції встановлюється значення **0** (ініціалізація). В той же час для статичної змінної **total** значення **0** встановлюється тільки один раз, при запуску програми. Після цього значення змінної **total** оновлюється між викликам функцій, завдяки чому є можливість рахувати всі введені за час роботи програми символи.

Специфікатор const

В C++ (але не в C) специфікатор **const** додає невеличких змін у класи пам'яті, що використовуються за умовчанням. В той час як глобальна змінна за умовчанням має зовнішнє зв'язування, *глобальна змінна* **const** за умовчанням має *внутрішнє зв'язування*. Іншими словами, в C++ глобальне визначення **const** розглядається як статичне глобальне визначення.

```
const int fingers = 10;  // те ж, що і static const int fingers;
int main(void)
…
```

Це зроблено з метою полегшення управління константами у багатомодульних програмах. Припустимо, що в деякому файлі-заголовку міститься визначено набір констант, які повинні використовуватись в декількох файлах однієї програми. Після включення препроцесором цього файлу-заголовку в кожний вихідний файл, в усіх вихідних файлах з'являться такі визначення.

Якщо б визначення глобального специфікатора **const** мало зовнішнє зв'язування, як звичайні змінні, то це було б помилкою, тому що глобальну змінну можна оголосити тільки в одному файлі. Іншими словами, представлене вище оголошення міг би містити тільки один файл, а в інших необхідно було б передбачити посилальні оголошення з використанням ключового слова **extern**. Більш того, тільки оголошення без **extern** забезпечує ініціалізацію значень:

```
// якщо б const мала зовнішнє зв'язування,
// необхідно було б вказувати extern
extern const int fingers;              // ініціалізація неможлива
extern const char * warning;
```

В результаті необхідно було б мати один набір визначень для одного файлу і другий набір - для інших файлів. Проте завдяки тому, що для даних **const** з

зовнішнім визначенням характерне внутрішнє зв'язування, можна використовувати одне і те ж визначення в усіх файлах.

Також треба зазначити, що при оголошенні імені з **const** в межах конкретної функції або блоку воно має діапазон доступу блоку. Тобто дану константу можна використовувати тільки тоді, коли програма виконує код цього блоку і, крім того, можна створювати константи в функції або у блоці, не піклуючись про можливі конфлікти з іменами констант, визначених десь в інших місцях програми.

Функції і зв'язування

Функціям також призначається клас пам'яті, проте вибір для них є більш обмеженим, ніж для змінних. В C++, як і в C, не дозволяється оголошувати одну функцію всередині іншої. Таким чином, всі функції автоматично отримують статичний клас пам'яті, а це означає, що вони існують, поки виконується програма. За умовчанням функції вважаються зовнішніми в тому розумінні, що для них характерне зовнішнє зв'язування и що їх можна використовувати в різних файлах. Фактично можна використовувати ключове слово **extern** у прототипі функції, щоб вказати, що дану функцію визначено в іншому файлі, проте це необов'язково. Крім того, можливо скористатися ключовим словом **static**, щоб надати функції внутрішнє зв'язування, обмежуючи таким чином її використання одним файлом. Ключове слово **static** можна застосувати як до прототипу, так і до визначення функції:

```
static int private(double x);
      ...
static int private(double x)
{                ...              }
```

Таке визначення робить відомою функцію тільки в межах даного файлу і не забороняє використання цього ж імені для будь-якої функції в іншому файлі. Як і у випадку зі змінними, статична функція перекриває зовнішнє визначення з файлу, в якому міститься статичне оголошення. Таким чином, в файлі, що містить статичне оголошення функції, ця версія функції буде застосовуватися навіть якщо мається зовнішнє оголошення функції з таким же ім'ям.

В C++ існує «*правило унікальності визначень*», згідно якому кожна програма повинна містити тільки одне визначення кожної невбудованої функції.

Для функцій з зовнішнім зв'язуванням це означає, що тільки один файл багатофайлової програми може включати визначення функції. Проте кожен файл, в якому використовується функція, повинен мати прототип (оголошення) функції.

Виключенням з цього правила є вбудовані функції: оголошення вбудованих функцій можна записувати у файл-заголовок. Таким чином, кожний файл, до якого включається файл-заголовок, буде мати оголошення вбудованих функцій. Проте C++ вимагає, щоб всі оголошення вбудованих функцій були ідентичними.

Схеми розподілення пам'яті і динамічна пам'ять

До цього моменту було розглянуто п'ять схем розподілення пам'яті C++, які використовуються для виділення пам'яті змінним (включаючи масиви і структури). Жодна з них не може бути застосована до пам'яті, яка надається шляхом використання оператора **new** мови C++ (або функції мови C **malloc()**). Такий вид пам'яті має назву «*динамічна пам'ять*». Як відомо, управління динамічною пам'яттю здійснюється операторами **new** і **delete**, і для неї не працюють правила визначення діапазонів доступу і зв'язування. Таким чином, динамічна пам'ять може отримуватися з однієї функції, а звільнятися – з іншої. На відміну від автоматичної пам'яті, динамічна пам'ять не підпорядковується правилу **LIFO** (last in first out). Порядок надання і вивільнення пам'яті залежить від того, коли і як було застосовано оператори **new** і **delete**. Як правило, компілятор використовує такі фрагменти пам'яті: один – для статичних змінних (цей фрагмент пам'яті може складатися з розділів або секцій), один – для автоматичних змінних (стек) і один (або декілька) – для динамічної пам'яті (фрагменти з області пам'яті, яка має назву **heap** – «купа»).

Незважаючи на те, що концепція схем розподілення пам'яті не працює по відношенню до динамічної пам'яті, її можна застосовувати до змінних типу покажчика, які використовуються для адресування динамічної пам'яті. Наприклад, якщо всередині функції записано такий оператор:

```
float * p_fees = new float [20];
```

то 80 байтів пам'яті (виходячи з того, що для типу **float** використовується чотири байти), наданих оператором **new**, залишаються зайнятими поки оператор **delete**

не звільняє їх. Проте покажчик **p_fees** припиняє існування, коли закінчується виконання функції, в якій записано це оголошення. Якщо потрібно, щоб виділені їй 80 байтів стали доступними іншій функції, слід передати або повернути адресу цієї пам'яті іншій функції. З іншого боку, якщо зробити це ж оголошення зовнішнім, то покажчик **p_fees** стане доступним всім функціям, які записано в файлі за таким оголошенням. Скориставшись конструкцією

```
extern float * p_fees;
```

в другому файлі, можна зробити даний покажчик доступним і в цьому файлі. Проте, треба звернути увагу на те, що оператор, який використовує **new**, щоб надати значення **p_fees**, має знаходитися в функції, тому що для ініціалізації статичних змінних можна застосовувати тільки константний вираз:

```
float * p_files;          // = new float[20] - тут неможливо
int main ()
{         p_fees = new float[20];
    ...
}
```

2.7.3. Простори імен, як загальне середовище реалізації програмного проекту

Простори імен у зв'язку з модульністю і ієрархією

В сучасних мовах програмування, які призначені для розробки складних багатокомпонентних програмних проектів, одним з важливих методів організації коду і забезпечення можливості його модифікації і повторного використання, являється групування програмних об'єктів з урахуванням їх логічних зв'язків. Механізм забезпечення реалізації модульності програмних проектів на основі логічної організації як правило пов'язаний з використанням так називаних ***просторів імен***.

Перш ніж розпочати аналізувати реалізацію просторів імен в C++, слід відмітити, що використання просторів імен не обов'язково передбачає об'єктну орієнтованість розроблених програмних компонентів. Проте можна вважати справедливим твердження, що активне використання в проектуванні просторів імен обумовлене перш за все розвитком об'єктно-орієнтованого програмування.

Непрямим підтвердженням цього факту може бути історія розвитку мови C++: перші реалізації C++ не підтримували просторів імен.

Як вже відомо, в програмі на C використовуються наступні *області бачення* програмних об’єктів: блок, функція, структура, одиниця трансляції (модуль), глобальний простір імен. Використання єдиного глобального простору породжує конфлікти імен у випадках, коли треба об’єднувати фрагменти незалежно розробленого коду.

В мові C++ одночасно з названими областями бачення використовуються *області бачення класу*. Клас також є, по суті, простором імен, в якому визначені елементи класу. Проте,якщо необхідно забезпечити простий механізм групування і приховування імен, класи вважаються занадто складним рішенням.

Простори імен надають механізм, який узагальнює глобальні оголошень в C і C++. ***Простір імен*** – це іменована або неіменована відкрита область дії імен. Це означає, що оголошення програмного об’єкта в просторі імен не приховує його для звернення з інших областей бачення, а лише вимагає кваліфікації імені простору імен для доступу до програмного об’єкта (змінної, константи, типу, функції, іншого простору імен).

Програмний проект: модулі і інтерфейси

Інший аспект використання просторів імен щільно пов'язаний з модульністю при проектуванні. Побудова модуля завжди передбачає процес логічного групування оголошень і визначень у зв’язку з вирішуваними задачами і використовуваними даними. Простори імен надають механізм відображення логічного групування програмних об’єктів засобами мови програмування. Таким чином, необхідно розрізнювати два поняття: модуль-файл, як сегмент *фізичної організації* багатомодульної програми, і модуль-простір імен, як сегмент *логічної організації*.

Коли один модуль використовує інший, немає необхідності знати всі подробиці реалізації, тому простір імен часто існує у двох проявах:

– простір імен, як зовнішній інтерфейс для користувача. Такий простір імен повинен містити оголошення і визначення, необхідні (і достатні) для правильного використання функціональності, яка надається модулем;

– простір імен, як загальне середовище реалізації програмного проекту. Такий простір імен повинен містити всі оголошення і визначення, які використовуються під час розробки.

Простори імен в C++

Та особливість мови C++, що в неї поєднуються процедурне програмування в стилі C і об’єктно-орієнтоване програмування, пояснює той факт, що реалізація просторів імен в C++ навряд чи може розглядатися у якості універсального рішення. Проте основні елементи реалізації концепції логічного групування програмних об’єктів в тому вигляді, в якому вони підтримуються в C++, подібним чином (із урахуванням особливостей синтаксису) реалізовані і в інших мовах.

Імена в C++ можна надавати змінним, функціям, структурам, перерахуванням, класам, а також елементам класів і структур. Коли програмні проекти стають занадто громіздкими, збільшується ймовірність виникнення конфлікту імен. З проблемою конфлікту імен можна зіткнутися, якщо для створення бібліотеки класу використовується більше ніж одне джерело. Наприклад, в двох бібліотеках визначено класи з іменами **List**, **Tree** і **Node**, але при цьому виконано правила сумісності. Якщо, наприклад, в програмі використовується клас **List** з першої бібліотеки і клас **Tree** – з другої, то при цьому кожний з них очікує взаємодію з власною версією класу **Node**. Такі конфлікти називаються ***проблемами простору імен***.

Стандарт C++ передбачає засоби, які забезпечують досконале управління діапазонами доступу імен.

В мові C++ реалізовано можливість створення ***іменованих просторів імен*** шляхом створення області визначення нового виду. Основне призначення такої області – забезпечити область пам'яті для оголошення імен. Імена з одного простору не конфліктують з тими ж іменами, які оголошено в інших просторах імен. При цьому існують механізми, які дозволяють іншим частинам програми використовувати оголошені в просторі імен об’єкти даних. Наприклад, у наведеному нижче програмному коді використовується нове ключове слово **namespace** для створення двох просторів імен **Jack** і **Jill**.

```
namespace Jack
{       double pail;
```

```
        void fetch ();
        int  pal;
        struct Hill { ... };
}
namespace Jill
{       double bucket(double n) { ...  }
        double fetch;
        int pal;
        struct Hill { ... };
}
```

Простори імен можуть знаходитися на глобальному рівні або всередині інших просторів імен, але вони не можуть бути розміщені в блоці. Внаслідок цього ім'я, яке оголошене в просторі імен, має зовнішнє зв'язування за умовчанням (поки воно не посилається на константу або вбудовану функцію).

Крім просторів імен, які оголошені користувачем, існує ще один простір імен – *глобальний.* Це відповідає області оголошення на рівні файлу, тому глобальні змінні тепер належать *глобальному простору імен*.

Імена з одного простору імен не конфліктують з іменами з іншого простору імен. Тобто, ім'я **fetch** в просторі імен **Jack** цілком законно може існувати поруч з ім'ям **fetch** в просторі **Jill**, а **Hill** з простору **Jill** може співіснувати з зовнішнім ім'ям **Hill**. Правила, які регламентують оголошення і визначення в просторах імен, аналогічні таким же правилам для глобальних оголошень і визначень.

Простори імен ***відкриті,*** це означає, що в існуючий просторів імен можна включати нові імена. Наприклад, оператор

```
namespace Jill
{       char * goose (const char *);        }
```

додає ім'я **goose** до списку імен в просторі імен **Jill.**

Аналогічно простір імен **Jack** містить прототип функції **fetch**(). Ще раз скориставшись простором імен **Jack**, можна розмістить програмний код функції далі в файлі (або в іншому файлі):

```
namespace Jack
{       void fetch () { … … … }
}
```

Зрозуміло, що потрібен деякий спосіб для доступу до імен конкретного простору імен. Найпростіший шлях полягає у використанні оператора

визначення діапазону доступу '**::**' (кваліфікатор області бачення), який дозволяє *кваліфікувати* ім'я, вказавши його разом з найменуванням простору імен:

```
Jack::pail = 12,34;   // використання змінної
Jill::Hill mole;      // створюється структура Hill
Jack::fetch();        // використання функції
```

Ім'я без додавання оператора '**::**', називається ***некваліфікованим ім'ям***, тоді як ім'я з простору імен, таке як **Jack::pail**, називається ***кваліфікованим ім'ям***.

Using-оголошення і using-директиви

Необхідність кваліфікувати імена кожен раз, коли вони використовуються, – не дуже приваблива перспектива, тому для спрощення обробки імен з простору імен мова C++ забезпечує два механізми – ***using-оголошення*** і ***using-директиви***. *Using-оголошення* дозволяє включати окремі ідентифікатори, а *using-директиви* підключають цілу область імен.

Using-оголошення складається з ключового слова using, після якого записується ім'я змінної.

```
using Jill::fetch;          // using-оголошення
```

Using-оголошення додає в область оголошення окреме ім'я. Наприклад, using-оголошення **Jill::fetch** в **main()** додає **fetch** в область оголошення функції **main()**. Зробивши таке оголошення, можна в **main()** використовувати ім'я **fetch** замість **Jill::feteh**.

```
namespace Jill
{       double bucket(double n) {  …  }
        double fetch;
        struct Hill {  ...  };
}

char fetch;

int main()
{       using Jill::fetch;      // включити fetch в локальний простір імен
        double fetch;           // Помилка! Повторне оголошення локального імені fetch
        cin >> fetch;           // ввести значення в Jill::fetch
        cin >> ::fetch;         // ввести значення в глобальну змінну fetch
        ...
```

```
}
```

Using-оголошення додає ім’я в локальну область оголошень, тому в даному прикладі неможливо створення іншої змінної з ім’ям **fetch**. Крім того, як і будь-яка інша локальна змінна, **fetch** перекриває глобальну змінну з таким же ім’ям.

Розміщення using-оголошень на зовнішньому рівні призводить до того, що відповідне ім’я змінної з’являється в глобальному просторі імен.

```
void other ();
namespace Jill
{       double bucket(double n) { ... }
        double fetch;
        struct Hill { ... };
}
using Jill::fetch;              // змінна fetch розміщується в глобальному просторі імен

int main ()
{       cin >> fetch;           // ввід значення в змінну fetch
        other ();
        ...
}

void other()
{       cout << fetch;          // вивід змінної fetch
...
}
```

Таким чином, using-оголошення робить доступним конкретне ім’я. Using-директиви, навпаки, роблять доступними всі імена.

Using-директива складається з імені простору імен, перед яким записується ключові слова **using namespace**, завдяки чому всі імена з простору імен стають доступними без необхідності використання оператора визначення області бачення.

```
using namespace Jack;     // робить всі імена простору імен Jack доступними
```

При розміщенні директиви на глобальному рівні імена простору імен становляться глобально доступними. В наведених раніше прикладах вже неодноразово було продемонстровано, як працює цей механізм:

```
#include <iostream>         // розміщує імена в просторі імен std
using namespace std;        // робить імена з області std глобально доступними
```

Якщо розмістити ***using-директиву*** в окремій функції, то імена стануть доступними тільки даній функції.

```
int vorn(int m)
{
   using namespace jack; // імена доступні тільки в функції vorn()
   ...
}
```

Стандартний простір імен

Елементи стандартної бібліотеки C++ оголошено в просторі імен **std** (яке має назву «стандартний простір імен»). Щоб скористатися оголошеннями з стандартного простору імен, необхідно застосувати using-оголошення або using-директиву. Так, у багатьох наведених раніше прикладах використовується визначення класів і функцій потокового вводу-виводу C++:

```
#include <iostream>          // Засоби стандартного вводу-виводу бібліотеки мови C++
using namespace std;
//  ...
```

Одне з завдань, яке вирішувалось в ході проектування мови C++, полягало у підтримці користувальницької бази мови C. Необхідність забезпечення сумісності з кодом, який не використовує простори імен, вимагала деяких специфічних підходів.

Так, якщо в ході проектування програми виникає необхідність використання засобів бібліотек мови C (не C++), то можна скористатися файлами-заголовками, імена яких сконструйовано з імен файлів-заголовків бібліотек мови C додаванням літери '**c**' на початку імені. Наприклад, для функцій із стандартної бібліотеки вводу-виводу мови C це виглядає наступним чином:

```
#include <cstdio>    // Засоби стандартного вводу-виводу бібліотеки мови С
using namespace std;
 //  ...
```

Треба звернути увагу, що в директиві препроцесору використовується ім'я файлу без розширення (**.h**). Ім'я файлу без розширення є офіційним стандартом і призначене для використання спільно з концепцією просторів імен.

Для того щоб забезпечити сумісність коду, що використовує концепцію просторів імен, і коду на C/C++, який не використовує простори імен, розробники стандартної бібліотеки зробили наступне. Файли-заголовки з розширенням **.h** забезпечують неявне підключення стандартного простору імен:

```
// Файл stdio.h
namespace std              // Всі необхідні оголошення
{        …       }
using namespace std;       // Підключення простору імен std
```

або

```
// Файл stdio.h
#include <cstdio>          // БЕЗ РОЗШИРЕННЯ
using namespace std;       // Підключення простору імен std

// Файл cstdio
namespace std              // Всі необхідні оголошення
{        …       )
```

Аналогічно підготовлені і інші стандартні файли-заголовки.

Реалізація файлів-заголовків подібним чином дозволяє забезпечувати роботу з програмами, які не використовують концепцію просторів імен, без переробки.

Using-директиви у порівнянні з using-оголошеннями

Застосування using-директиви для імпорту всіх імен одразу ***не*** рівнозначне використанню декількох using-оголошень. ***Using-директива*** – це як би масове застосування оператора визначення діапазону доступу. Використання using-оголошення рівноцінно оголошенню імені в сфері дії using-оголошення. Якщо деяке ім'я вже було оголошено в функції, то це ім'я неможливо імпортувати без using-оголошення. Проте, якщо застосовується using-директива, то результат такого застосування буде аналогічним оголошенню відповідних імен в

найменший області оголошення, яка містить як using-оголошення, так і сам простір імен. В представленому нижче прикладі розглядається глобальний простір імен. Для імпорту конкретного імені, яке вже було оголошено в деякій функції, можна скористатися using-директивою, але при цьому локальне ім'я буде перекривати ім'я з простору імен, подібно тому, як воно перекривало б глобальну змінну з тим же ім'ям. Проте, все ж таки можна використовувати оператор визначення області бачення:

```
namespace Jill
{    double bucket(double n) { … }
     double   fetch;
     struct   Hill   { … };
}
char fetch;                          // глобальний простір імен

int main()
{    using namespace Jill;           // імпортувати імена з простору імен
     struct Hill Thrill;             // створити структуру Jill::Hill
     double water = bucket(2);       // використати ім'я Jill::bucket();
     double fetch;                   // це не помилка:
                                     // відміняється ім'я Jill::fetch
     cin >> fetch;                   // ввести значення в локальну змінну fetch
     cin >> ::fetch;                 // ввести значення в глобальну змінну fetch
     cin >> Jill::fetch;             // ввести значення в змінну Jill::fetch
          …
}

int foom() {
     Hill top;                       // неправильно!
     Jill::Hill crest;               // правильно
}
```

В даному прикладі в функції **main()** ім'я **Jill::fetch** розміщене в локальному просторі імен. Воно не має локальної області бачення, і тому не перекриває глобальне ім'я **fetch**. Тим не менш, обидві останні змінні **fetch** стають доступними, якщо скористатися оператором визначення області бачення.

Треба відмітити також, що хоча застосування using-директиви в функції дозволяє використовувати імена з області імен, як такі що вони оголошені поза функцією, вона не робить ці імена доступними іншим функціям файлу. Як наслідок, в розглянутому прикладі, функція **foom()** не може використовувати некваліфікований ідентифікатор **Hill**.

Підсумовуючи наведені приклади треба ще раз відмітити наступне. Одне і те ж ім’я може бути визначено як у просторі імен, так і в будь-якій іншій області оголошень. При спробі застосувати using-оголошення для розміщення імені з простору імен в області оголошень, обидва імені вступають у конфлікт з виводом відповідного повідомлення про помилку. Але якщо застосувати using-директиву для переносу імені з простору імен в область оголошень, то локальна версія цього імені перекриє версію з простору імен.

В цілому ж застосування using-оголошень вважається більш безпечним, тому що воно явно показує які імена стануть доступними. І якщо таки буде конфліктувати з локальним іменем, компілятор обов’язково повідомить про це. А using-директива додає всі імена без винятку, навіть ті, які не планується використовувати. Якщо локальне ім’я вступає в конфлікт, то воно перекриває версію імені з простору імен, але про це нічого не повідомляється. Крім того, внаслідок відкритого характеру простору повний перелік імен простору імен може розповсюджуватися на декілька областей, в результаті чого стає важко слідкувати за тим, які саме імена додаються.

Теж саме можна сказати про використання стандартного для C++ простору імен.

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

По-перше, файл-заголовок **iostream** розміщує всі імена в просторі імен **std**. Наступний рядок експортує все, що міститься в цьому просторі імен, в глобальний простір імен. Таким чином, даний підхід просто примушує згадати старий підхід, який існував до появи простору імен. Основне логічне обґрунтування такого підходу – доцільність. Це нескладно, а якщо система не підтримує простір імен, можна замінити наведені два рядки на наступний:

```
#include <iostream.h>
```

Прихильникам роботи із просторами імен можна порадити бути більш вимогливими у виборі засобів і застосовувати або оператор визначення області бачення, або using-оголошення. Іншими словами слід відмовитися від використання наступного оголошення:

```
using namespace std;   // уникати використання даної конструкції
                       // як є занадто невизначеною
```

і замість нього рекомендується використовувати наступні оператори:

```
int x;
std::cin >> x;
std::cout << x << std::endl;
```

або:

```
using std::cin;
using std::cout;
using std::endl;
int x;
cin >> x;
cout << x << endl;
```

В просторах імен передбачена можливість вкладених оголошень:

```
namespace elements
{
        namespace fire
        {
                int flame;
                ...
        }
float water;
}
```

В даному випадку звернення до змінної **flame** виконується так: **elements::fire::flame**. Аналогічно, внутрішні імена можна зробити доступними за допомогою наступної using-директиви:

```
using  namespace  elements::fire;
```

Крім того, всередині простору імен можна користуватися using-директивами і using-оголошеннями:

```
namespace  myth
{       using Jill::fetch;
        using namespace elements;
        using std::cout;
        using std::cin;         }
```

У зв’язку з тим, що **Jill::fetch** тепер є частиною простору імен **myth**, в якому її можна називати як **fetch**, доступ до неї можна отримати наступним чином:

```
std::cin >> myth::fetch;
```

Дана змінна залишається частиною простору імен **Jill**, тому до неї можна звернутися і по імені **Jill::fetch**:

```
std::cout << Jill::fetch;       // вивести значення, яке
                                // введене в змінну myth::fetch
```

Це також можна зробити зовсім безпечно, виключивши будь-який конфлікт локальних змінних:

```
using  namespace  myth;
cin >> fetch;           // насправді це
                        // std::cin  i  Jill::fetch
```

Неіменовані простори імен

Можна створити *неіменований простір імен* – для цього треба опустити ім’я після ключового слова **namespace**:

```
namespace  // неіменований простір імен
{
        int ice;
        int bandycoot;
}
```

Результат виконання наведених вище оголошень буде таким, нібито за ним записано using-директиву, тобто імена, які оголошено в цьому просторі імен, знаходяться в потенційній області бачення, яка сягає границі області оголошень, в якій розміщується неіменований простір імен. В цьому відношенні їх можна розглядати як глобальні змінні. Проте, внаслідок відсутності імені у простору імен, неможливо явно застосувати using-директиву або using-оголошення для того, щоб зробити доступ до цих імен можливим з будь-якої точки програми. Зокрема, неможливо використовувати імена з неіменованого простору імен в іншому файлі програми. Неіменовані простори можна розглядати у якості альтернативи використання зовнішніх статичних змінних з внутрішнім зв’язуванням. Тим більше, що і в стандарті C++ зазначено, що використання

ключового слова **static** в просторах імен і в глобальному діапазоні доступу вважається застарілим (**deprecate**) ("**deprecate**" – це термін, який використовується в стандарті для зазначення того, що дана практика ще застосовується, але у майбутньому може бути виключена з стандарту). Наприклад, наступний код:

```
static int counts;      // статичний клас пам'яті, внутрішнє зв'язування
int other;
int main ()
{       …       }
```

у відповідності до вимог стандарту слід замінити такими операторами:

```
namespace
{
        int counts;     // статичний клас пам'яті, внутрішнє зв'язування
}
int other ();
int main ()
{       …       }
int other ()
{       …       }
```

Приклад простору імен

Наступний приклад демонструє деякі властивості простору імен у багатофайловій програмі. Перший – файл-заголовок. Він містить деякі з елементів, які зазвичай знаходяться в такому файлі, – константи, визначення структур і прототипи функцій. В даному прикладі елементи розміщені в двох просторах імен. Перший простір імен с ім'ям **pers** містить визначення структури **Person**, а також прототипи двох функцій – перша розміщує в структурі ім'я особи, а друга відображає вміст структури. Другий простір імен, **debts**, визначає структуру для зберігання імені особи і суми грошей, якими володіє дана особа. В цій структурі використовується структура **Person**, таким чином, простір імен **debts** містить **using**, щоб зробити імена з **pers** доступними в просторі імен **debts**. Простір імен **debts** також містить деякі прототипи.

Приклад 2.40: Простори імен.

```
// namesp.h - файл-заголовок
namespace pers
{       const int LEN = 40;
        struct Person
```

```
        {       char fname[LEN];
                char lname[LEN];    };
        void  getPerson(Person &);
        void showPerson(const Person &);
}
namespace debts
{       using namespace pers;
        struct Debt
        {       Person name;
                double amount;              };
        void getDebt(Debt &);
        void showDebt(const Debt &);
        double sumDebts(const Debt ar[ ], int n);                }
```

Наступний файл (див. далі) складено за звичайною схемою, згідно якої необхідно мати файл с вихідним кодом, який містить визначення для прототипів функцій з файлу-заголовку. Імена функцій, які оголошено в просторі імен, мають діапазон доступу простору імен, тому визначення повинні бути в тому ж самому просторі імен, що і оголошення. Саме тут стає зрозумілою корисність відкритості просторів імен. Підключення вихідних просторів імен виконується за допомогою включення **namesp.h**. Далі в файлі визначення функцій додано в два простори імен.

Приклад 2.40: Простори імен (продовження)

```
// namesp.cpp  -  простори імен
#include  <iostream>
using namespace std;
#include "namesp.h"
namespace pers
{       void getPerson(Person & rp)
        {       cout << "Enter first name: ";
                cin >> rp.fname;
                cout << "Enter last name: ";
                cin >> rp.lname;
        }
        void showPerson(const Person & rp)
        {       cout << rp.lname << " , " << rp. fname ; }
}
namespace debts
{       void getDebt(Debt & rd)
        {       getPerson(rd.name);
                cout << "Enter debt: ";
                cin >> rd.amount;
        }
        void showDebt(const Debt & rd)
```

```
        {       showPerson(rd.name);
                cout << "; $" << rd. amount << endl;
        }
        double sumDebts(const Debt ar[ ], int n)
        {       double total = 0;
                for (int i = 0; i < n; i++)
                        total += ar[i].amount;
                return total;
        }
}
```

Третій файл програми – це файл з вихідним кодом, в якому використовуються визначені в просторі імен структури і функції. В програмі показано декілька способів зробити доступними ідентифікатори простору імен.

Приклад 2.40: Простори імен (продовження)

```
// usenmsp.cpp – використання просторів імен
#include <iostream>
using namespace std;
#include "namesp.h"
void other() ;
void another ();
int main()
{       using debts::Debt;
        using debts::showDebt;
        Debt golf = { {"Benny", "Goatsniff"}, 120.0 };
        showDebt(golf);
        other();
        another();
        return 0;
}
void other()
{       using namespace debts;
        Person dg = {"Doodles", "Glister"};
        showPerson(dg);
        cout << endl;
        Debt zippy[3];
        int i;
        for (i = 0; i < 3; i++) getDebt(zippy[i]);
        for (i = 0; i < 3; i++) showDebt(zippy[i]);
        cout << "Total debt: $" << sumDebts (zippy, 3) << endl;
        return;
}
void another()
{       using pers::Person;
        Person collector = { "Milo", "Rightshift" };
        pers::showPerson(collector);
```

```
    cout << endl ;                }
```

По-перше, в функції **main()** використовуються два using-оголошення:

```
using debts::Debt;                // робить доступним визначення структури Debt
using debts::showDebt;            // робить доступною функцію showDebt
```

Треба звернути увагу на те, що в using-оголошенні використовується тільки ім'я; наприклад, в другому оголошенні не визначено тип, що повертається функцією **showDebt**, або її сигнатура, а є тільки ім'я цієї функції (тобто, якщо функцію перевантажено, одне using-оголошення буде імпортувати всі її версії). Крім того, хоча в обох функціях, **Debt** і **showDebt**, використовується тип **Person,** немає необхідності імпортувати будь-яке імя **Person**, тому що в просторі імен **debt** вже міститься using-директива, яка включає простір імен **pers**.

В функції **other()** використовується менш привабливий спосіб імпортування всього простору імен за допомогою директиви **using**:

```
using namespace debts;   // робить доступними для функції other()
                         // всі імена з просторів debts і pers
```

Директива **using** в **debts** імпортує простір імен **pers**, тому в функції **other()** можуть використовуватися тип **Person** і функція **showPerson()**.

Нарешті, в функції **another()** використовується using-оголошення і оператор визначення діапазону доступу для доступу до окремих імен:

```
using pers::Person;
pers::showPerson(collector);
```

Результати виконання програми:

```
Goatsniff, Benny: $120
Glister, Doodles
Enter first name: Arabella
Enter last name: Binx
Enter debt: 100
Enter first name: Cleve
Enter last name: Delaproux
Enter debt; 120
Enter first name; Eddie
Enter last name: Fiotox
Enter debt; 200
Binx, Arabellai $100
Delaproux, Cleve: $120
```

```
Fiotox, Eddie: $200
Total debt: $420
Rightshift, Milo
```

Висновки

C++ зручна мова для розробки програм, в яких використовується декілька файлів одночасно. Ефективною організаційною стратегією в цьому відношенні може бути використання файлу-заголовку для визначення типів користувача і прототипів для функцій, які обробляють типи користувача. Визначення функцій слід виносити в окремий файл с вихідним кодом. В обох файлах – в заголовку і в файлі з вихідним кодом визначається і реалізується тип даних, визначений користувачем, а також те, як його можна використовувати. Функція **main()** і інші функції, які використовують ці функції, можуть бути розміщені в третьому файлі.

Схеми розподілення пам'яті C++ визначають, як довго змінні залишаються в пам'яті, і які частини програми мають до них доступ (область бачення і зв'язування). Автоматичними являються ті змінні, які визначені в межах блоку, такого як, наприклад, тіло функції або блок всередині блоку. Вони існують і доступні тільки тоді; коли програма виконує оператори в блоці, який містить їх визначення. Автоматичні зміні можуть бути оголошені за допомогою специфікаторів класу пам'яті **auto** і **register** або якщо змінна оголошується без специфікаторів, що дорівнює використанню **auto**. Специфікатор **register** вказує компілятору, що змінна буде часто використовуватися.

Статичні змінні існують протягом всього періоду виконання програми. Змінна, яку визначено за межами будь-якої функції, належить зовнішньому класу пам'яті. Вона відома всім функціям у файлі, які з'являються в цьому файлі після її визначення (діапазон доступу файлу). Для того, щоб інший файл міг користуватися такою змінною він повинен оголосити її із використанням ключового слова **extern**. Змінна, яку оголошено поза межами будь-якої функції, або та в її оголошенні є ключове слово **static**, має діапазон доступу файлу, але вона недоступна іншим файлам (внутрішнє зв'язування). Змінна, яку оголошено всередині блоку із ключовим словом **static**, є локальною по відношенню до цього блоку, проте вона зберігає своє значення протягом всього часу виконання програми.

За умовчанням всі функції в C++ мають зовнішній клас пам'яті, тому їх можуть спільно використовувати відразу декілька файлів. Проте для вбудованих

функцій і функцій, оголошених з ключовим словом **static**, характерне внутрішнє зв'язування і їх використання обмежене файлом, в якому міститься їх визначення.

Простір імен надає можливість вказувати іменовані області пам'яті, в яких можна оголошувати ідентифікатори. Вони призначені для зниження ймовірності конфліктів імен, що особливо важливо для великих програм, які використовують програмні коди, розроблені різними постачальниками програмних продуктів. Ідентифікатори в просторах імен можуть ставати доступними при використанням оператора визначення діапазону доступу, а також завдяки застосуванню ***using-оголошення*** або *директиви* ***using***.

Стандартні бібліотечні функції C++ реалізовано в просторі імен **std**. Для файлів-заголовків старих зразків, таких як **iostream.h**, непотрібно використовувати простори імен, проте новий файл **iostream** потребує підключення простору імен **std**.

Контрольні запитання та завдання

1. Пояснити, що треба включити у визначення класу мовою C++. Навести власні приклади визначень, наприклад, класу «Комп'ютер».
2. Дати визначення поняттям «загальний» і «закритий» елемент класу. Як такі елементи визначаються у класі мовою C++?
3. Пояснити, як атрибути та операції класів мови UML записати мовою C++.
4. Пояснити сутність інкапсуляції і її реалізація в програмі на C++.
5. Навести приклади використання визначення класу для створення об'єктів даного класу та покажчиків на об'єкти.
6. Які операції мови C++ використовуються для доступу до елементів класу? Чим відрізняється доступ з використанням змінної типу «об'єкт класу» від доступу через покажчик на об'єкт.
7. Пояснити фізичний сенс конструктора та деструктора класу.
8. В програмах на C++ можна визначити такі конструктори: без параметрів, ініціалізації, перетворення і копіювання. Пояснити їх призначення.
9. В чому полягає сенс статичних методів та властивостей класу?
10. Які перетворення типів можна задавати у визначенні класу мовою C++?
11. Пояснити призначення дружніх функцій і їх оголошення в програмах.
12. Пояснити сутність перевантаження операцій в класі дружніми функціями та методами класу.
13. Показати на прикладах, як перевантажуються префіксні і постфіксні операції інкремента та декремента в C++.
14. Пояснити сутність успадкування та показати, як це позначається на діаграмі класів (UML) та в коді мовою C++.
15. Пояснити сутність множинного успадкування та проблем, що можуть виникати при його використанні.
16. Яким чином здійснюється ініціалізація елементів базового класу у похідному класі при успадкуванні?
17. Пояснити сутність поліморфізму і його реалізації в C++.
18. В чому полягає зміст статичного та динамічного зв'язування методів класу?
19. Віртуальні функції і поліморфічні кластери: пояснити, як вони створюються і як викликаються віртуальні функції.

20. Пояснити призначення і формат оголошення абстрактних базових класів та яким стереотипом це позначається на діаграмах класів UML.

21. Дати характеристику засобам обробки виключень в C++. Навести приклади обробки виключень в класах та класів виключень.

22. Пояснити, від яких чинників залежить час життя, діапазон доступу та зв'язування змінних в програмах на C++.

23. Що таке простори імен та як вони використовуються?

24. Що відбудеться, якщо спробувати в програмі (Приклад 2.15) виконати доступ до '**i**' через інтерфейс класу **C** або **B** ?

25. Що відбудеться, якщо вилучити ім'я **A** перед оператором "**::**" в методі **geti** класу **C** (Приклад 2.15)?

26. Як вплине на програму (Приклад 2.16), якщо в похідних класах **Child** і **GrandChild** видалити ключове слово **virtual** перед функцією **answerName**?

27. Як працюватиме програма (Приклад 2.16), якщо видалити ключове слово **virtual** перед усіма функціями-методами **answerName**?

28. Що буде, якщо в програмі (Приклад 2.21) переписати функцію **show(Derived& d)** у вигляді:

```
void show(Derived& d)
    { cout << " id " << d.id << " name " <<
    d.name << " ch " << d.ch << endl; }
```

29. Що зміниться в роботі програми (Приклад 2.26), якщо з оголошення класів **Cow** і **Buffalo** видалити ключове слово **virtual**.

30. Розробити мовою C++ клас комплексних чисел; визначити в даному класі перевантаження арифметичних операцій над комплексними числами.

31. Розробити програму мовою C++ для моделювання черги на базі масиву. В програмі визначити два класи: «**Черга**» та «**Елемент_черги**». Виключення типу «порожня черга» та «переповнення масиву» обробити засобами **try**, **throw**, **catch**.

3. ВВЕДЕННЯ В УЗАГАЛЬНЕНЕ ПРОГРАМУВАННЯ

3.1. Параметричний поліморфізм. Шаблони функцій

3.1.1. Види поліморфізму та поняття узагальненого програмування

Як відмічалося вище, *поліморфізм*, у загальному сенсі – здатність об’єкта змінювати форму.

Стосовно ООП, поліморфізм розглядається як властивість, що дозволяє деяке повідомлення передавати різними шляхами. Отже, повідомлення може мати безліч різних реалізацій.

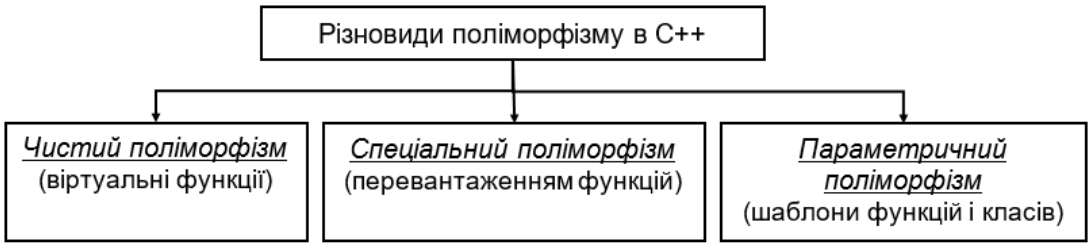


Рисунок 3.1 – Типи поліморфізму в C++

Чистий поліморфізм реалізується за допомогою механізму віртуальних функцій.

Спеціальний поліморфізм підтримується перевантаженням функцій.

Параметричний поліморфізм забезпечується шаблонами функцій і класів.

Мова C++ підтримує так зване *узагальнене* або *параметричне* програмування.

Програмування без прив'язки до конкретного типу даних, зазвичай називають ***узагальненим програмуванням***. Якщо типізація управляється параметрами, то говорять про ***параметричне узагальнене програмування***.

Основна область застосування узагальненого програмування – реалізація алгоритмів, які не прив'язані до конкретних типів оброблюваних даних.

Основним механізмом, що забезпечує узагальнене програмування в C++, є *шаблонні функції*, *класи* і їх спеціалізації.

Параметризовані компоненти мають властивість адаптивності до конкретної ситуації, в якій такий компонент використовується, що дозволяє розробляти досить універсальні і в той же час високоефективні компоненти програм (зокрема, об'єкти).

Параметричне програмування в мові C++ реалізовано за допомогою *шаблонів* (template). У C++ визначено два види шаблонів: *шаблони-класи* та *шаблони-функції*.

Шаблони-класи можуть використовуватися багатьма способами, але найбільш очевидним є їх використання в якості адаптивних об'єктів пам'яті.

Шаблони-функції можуть використовуватися для визначення параметризованих алгоритмів. Основна відмінність шаблону-функції від звичайного визначення функції в тому, що не потрібно повідомляти компілятору, до яких типів параметрів застосовується функція, він сам може визначити це за типами її фактичних параметрів.

3.1.2. Шаблони функцій та шаблонні функції

Розглянемо просту функцію, що реалізує алгоритм порівняння двох цілих величин:

```
/* Якщо значення iVal_1 менше iVal_2, то повертається значення iVal_1,
   в іншому випадку повертається значення iVal_2  */
int min (int  iVal_1, int  iVal_2)
        { return iVal_1 < iVal_2 ? iVal_1 : iVal_2; }
```

Для кожного типу порівнюваних величин необхідно визначати власний варіант функції **min**(). Ось як ця функція виглядає для **float**:

```
float min (float fVal_1, float fVal_2)
        { return fVal_1 < fVal_2 ? fVal_1 : fVal_2; }
```

А для **double** ...? А для … ? ... і так для всіх використовуваних в програмі типів порівнюваних величин.

Можна також скористатися засобами **препроцесору**:

```
#define min(a,b)   ((a)<(b)?(a):(b))
```

Це визначення правильно працює в простих випадках:

```
min(10, 20);          // для int
min(10.0, 20.0);      // для float
```

У більш складних випадках можна отримати несподівані результати. Це відбувається через те, що препроцесор діє до компілятора і незалежно від компілятора. Препроцесор взагалі виконує лише просту текстову обробку вихідного модуля.

C++ надає ще один засіб для вирішення цієї задачі. При цьому зберігається властива макровизначеннями стислість і строгість контролю типів мови. Цим засобом є *шаблон функції*.

Шаблон функції дозволяє визначати ціле сімейство функцій. Це сімейство характеризується загальним алгоритмом, який може застосовуватися до даних різних типів. При цьому визначення конкретного типу даних для чергового варіанту функції забезпечується спеціальною синтаксичною конструкцією, званою ***списком параметрів шаблону функції***.

Оголошення функції, якому передує список параметрів шаблону, називається ***шаблоном функції***.

Оголошення і визначення шаблону функції починається ключовим словом **template**, за яким записується в кутових дужках розділений комами *непорожній* список параметрів шаблону. Ця частина оголошення або визначення зазвичай називається ***заголовком шаблону***.

Кожен параметр шаблону складається з службового слова **class**, за яким йде ідентифікатор. У контексті оголошення шаблону функції службове слово **class** не несе ніякої особливого змістового навантаження. Аналогічна конструкція використовується також і для оголошення шаблону класу, де ключове слово **class** грає свою особливу роль. У заголовку шаблону імена параметрів шаблону повинні бути унікальні.

Безпосередньо за заголовком шаблону розташовується прототип або визначення функції – все залежить від контексту програми. Як відомо, у прототипа і визначення функції також є власний заголовок. Цей заголовок складається з специфікатора значення, що повертається, (цілком можливо, що специфікатором значення, що повертається, може бути ідентифікатор зі списку параметрів шаблону), ім'я функції і список параметрів. Ідентифікатори з заголовка шаблону також можуть входити в список параметрів функції. У цьому списку вони грають роль специфікаторів типу.

Параметри, у яких в якості специфікаторів типу використовуються ідентифікатори зі списку параметрів шаблону, називаються ***шаблонними параметрами***.

Поряд з шаблонними параметрами в список параметрів функції може також входити параметри основних і похідних типів.

Шаблон функції служить інструкцією для транслятора. За цією інструкцією транслятор може самостійно побудувати визначення нової функції.

Параметри шаблону в шаблонних параметрах функції позначають місця майбутньої підстановки, яку здійснює транслятор в процесі побудови функції. Область дії параметрів шаблону обмежується шаблоном. Тому в різних шаблонах дозволено використання одних і тих же ідентифікаторів-імен параметрів шаблону.

Як приклад розглянемо програму, в якій для визначення мінімального значення використовується шаблон функції **min**().

```
template <class Type> Type _min (Type a, Type b);   // Прототип шаблону функції
int main ()                    // Функція main()
{ _min(10, 20);                // Виклик шаблонної функції int _min (int, int);
  _min(10.0, 20.0);            // Виклик шаблонної функції float _min (float, float);
  return 1;  }
template <class Type> Type _min (Type a, Type b) // Визначення шаблону функції
        {  return a < b ? a : b; }
```

Тут ключове слово **template** позначає початок списку параметрів шаблону. Цей список містить єдиний ідентифікатор **Type**. Сама функція містить два оголошення **шаблонних параметрів**, специфікованих параметром шаблону **Type**. Специфікація поверненого значення також представлена параметром шаблону **Type**.

В функції **main()** виконується виклик **шаблонних функцій.** Тип фактичних параметрів визначається компілятором на основі виразу виклику (компілятор повинен розпізнати тип параметрів) і визначення шаблону. Компілятор самостійно будує різні визначення шаблонних функцій.

Визначення шаблону функції змушує транслятор самостійно добудовувати визначення нових шаблонних функцій, а точніше, створювати множину спільно використовуваних функцій, у яких типи параметрів і, можливо, тип значення, що повертається, залежить від типу параметрів і типу поверненого значення у викликах шаблонної функції. Цей процес визначення називають **конкретизацією** (або **інстанціюванням**) шаблону функції.

В результаті конкретизації шаблону функції **min**() транслятор вибудовує варіант програми *з двома* шаблонними функціями. По виразу виклику на основі шаблону будується шаблонна функція.

Треба звернути увагу на відмінність однокореневих слів: ***шаблон функції*** *і* ***шаблонна функція*** *– два різних поняття*.

В результаті конкретизації програма буде наступною:

```
int  _min (int a, int b);              // Прототип шаблонної функції
float  _min (float a, float b);              // Прототип шаблонної функції
int main ()
        { Bmin(10, 20);              // Виклик шаблонної функції
          _min(10.0, 20.0);          // Виклик шаблонної функції
          return 1;     }
int  _min (int a, int b)                     // Шаблонна функція
        { return a < b ? a : b; }
float  _min (float a, float b)               // Шаблонна функція
        { return a < b ? a : b; }
```

Побудова шаблонної функції здійснюється на основі виразів виклику. При цьому в якості значень параметрів у виразі виклику може бути використано значення будь-яких типів, для яких визначено використовувані в тілі функції операції. Так, для функції **_min**() тип параметрів залежить від області визначення операції порівняння '<'.

Типи формального параметру шаблону і значення параметру виразу виклику співставляються без урахування будь-яких модифікаторів типу. Наприклад, якщо параметр шаблону у визначенні функції оголошено як

```
template <class Type> Type*  _min (Type *a, Type *b)
        { return a < b ? a : b; }
```

і при цьому виклик функцій має вигляд:

```
int a = 10, b = 20;
int *pa = &a, *pb = &b;
_min(pa, pb);          // порівнюються адреси

_min(a, b);            // порівнюються числа
```

то в процесі конкретизації ідентифікатор типу **Type** буде замінений ім'ям похідного типу **int**.

У процесі конкретизації неприпустимі розширення типів та інші перетворення типів параметрів:

```
template <class Type> Type _min (Type a, Type b)
        { return a < b ? a : b; }
unsigned int a = 10;
… … …
_min(1024, a);                  // типи параметрів - int та unsigned int (помилка)
```

Тут транслятор повідомить про помилку. У виклику функції тип другого фактичного параметра модифіковано в порівнянні з типом першого параметру – **int** та **unsigned int**. Це неприпустимо. В процесі побудови нової функції транслятор не розпізнає модифікацію типів. У виклику функції типи параметрів повинні збігатися. Виправити помилку можна за допомогою явного приведення типу першого параметра.

```
_min((unsigned int)1024, a);
```

Ім'я параметра шаблону використовується в якості імені типу. З його допомогою спеціалізуються формальні параметри, визначається тип значення, визначається тип об'єктів, локалізованих в тілі функції. Ім'я параметра шаблону приховує об'єкти з аналогічним ім'ям в глобальній по відношенню до визначення шаблонної функції області видимості. Якщо в тілі шаблонної функції необхідний доступ до зовнішніх об'єктів з тим же ім'ям, слід використовувати операцію зміни області видимості '**::**'.

Розглянемо приклад класу **ComplexType**. На множині комплексних чисел визначено лише два відношення: ***рівності*** (передбачає одночасну рівність дійсної та уявної частин) і ***нерівності*** (передбачає всі інші випадки). У програмі оголошено і визначимо шаблон функції **neq**(), яка буде перевіряти на нерівність значення різних типів.

Для того, щоб побудувати шаблонну функцію **neq**() для комплексних чисел, необхідно додатково визначити операторну функцію перевизначення операції '!=' для об'єктів-представників множини комплексних чисел. Це важливо, оскільки операція '!=' явно задіяна в шаблоні **neq()**. Транслятор не зрозуміє, як трактувати символ '!=', а, отже, і як будувати шаблонну функцію **neq(ComplexType, ComplexType)**, якщо ця операторна функція не визначена для класу **ComplexType**.

```
template <class Type> int neq (Type, Type); // Прототип шаблону функції
class ComplexType
{ public:
```

```
        double real;
        double imag;
// Конструктор:
        ComplexType(double re = 0.0, double im = 0.0)
                        {real = re; imag = im;}

        int operator != (ComplexType  &KeyVal)  // Перевантаження операції "!="
                { if (real == KeyVal.real && imag == KeyVal.imag) return 0;
                  else  return 1; }
};
void main ()
{ // Визначені і проініціалізовані змінні трьох типів.
        int i = 1, j = 2;
        float k = 1.0, l = 2.0;
        ComplexType CTw1(1.0,1.0), CTw2(2.0,2.0);
  // На основі виразів виклику транслятор будує три шаблонні функції.
        cout << "neq() for int:" << neq(i,j) << endl;
        cout << "neq() for float:" << neq(k,l) << endl;
        cout << "neq() for ComplexType:" << neq(CTw1, CTw2) << endl;
}
// Визначення шаблону функції:
template <class Type> int neq (Type a, Type b)
        { return a != b; }
```

Транслятори старих версій C++ вимагали обов'язкове включення всіх параметрів шаблону в список параметрів функції. В нових стандартах цю вимогу скасовано.

Такий шаблон:

```
template <class TYPE>
void convert(int i) { TYPE temp(i); return temp; )
```

раніше вважався неприпустимим, але тепер це також вірно, у відповідності із стандартом, запропонованим ANSI. Згідно цього стандарту виклик функції виконується так:

```
convert<double> ( i + j );
```

Оскільки такий виклик функції раніше вважався помилковим, то подібний код може не працювати у деяких існуючих системах.

3.1.3. Збіг сигнатури і перевантаження шаблонних функцій

Часто підпрограма не може працювати для якогось «*спеціального*» випадку. Наприклад, наступна форма шаблону функції обміну **swap** працює для базових типів.

```
template <class T>
void swap(T &x, T &y)
{       T temp;
temp=x;
x=y;
y=temp;       }
```

Шаблон функції використовується для того щоб створювати для будь-якого виклику відповідну функцію, що недвозначно відповідає параметрам.

```
int i, j;
char   str1[10]={'1','1','1','1','1','1'},
       str2[10]={'2','2','2','2','2','2'};
complex c1, c2;
swap(i, j);                    // i, j -int допустимо
swap(c1, c2);                  // c1, c2 -complex допустимо
swap(str1[50], str2[33]);      // обидві -char допустимо
swap(str1, str2);              // помилка, ім'я масиву - константа
//swap(i, chj);                // i - int, ch - char помилка
```

Зробимо так, щоб **swap** працювала для рядків, поданих як символьні масиви. Для цього опишемо наступний **спеціальний** випадок, який **перекриває** шаблонну функцію для випадку обміну символьних масивів.

```
#include <string.h>
void swap (char *s1, char *s2)
{ int len = (strlen(s1) >= strlen(s2)) ? strlen(s1) : strlen(s2);
  char *temp=new char [len+1];
  strcpy(temp,s1);
  strcpy(s1,s2);
  strcpy(s2,temp);
  delete temp; }
```

З додаванням такого явного випадку, при виклику з точною відповідністю сигнатурі **swap**(), має**ться** перевага над точною відповідністю, знайденою за допомогою підстановки шаблону.

Алгоритм вибору функції перевантаження

1. Знайти точну відповідність функції не-шаблону.

2. Знайти точну відповідність, що використовує шаблон функції.

3. Забезпечити звичайну обробку параметра для функцій не-шаблонів,

Таким чином, при визначені шаблонів функцій треба спиратися на наступні основні положення.

1) ***Шаблон функції*** – це оголошення функції, перед яким визначено специфікація шаблону. Специфікація шаблону складається з ключового слова **template**, за яким слідує список параметрів, укладений в кутові дужки '< >'.

2) Шаблони функцій мають параметри типу, які позначаються ключовим словом **class**, за яким йде ідентифікатор. Ключове слово **class** в контексті шаблонів означає будь-який тип, а не тільки клас. Ідентифікатор служить для заміщення імені типу. Може бути більш одного параметра типу.

3) Шаблони функцій автоматично розширюються транслятором до повного визначення функції.

4) Якщо функція-шаблон повинна працювати з будь-яким класом, то в цьому класі операції функції-шаблону необхідно перевантажити. В іншому випадку при зв'язуванні програми відбудеться помилка.

5) Шаблони функцій можуть бути перевантажені іншими функціями.

6) Для певних типів шаблони функцій можуть бути перекриті для того, щоб виконувати (або не виконувати) дії, які функції-шаблони не виконують (або виконують).

3.1.4 Програмування з використанням шаблонів функцій

Приклад 3.1: Копіювання масиву

Більшість функцій мають одне і те саме тіло коду, незалежно від типу. Наприклад, ініціалізація одного масиву від іншого масиву того ж самого типу. Зазвичай цей код виглядає так:

```
for(i=0; i<n; i++) a[i]=b[i];
```

Даний код можно на C/C++ автоматизувати простою макрокомандою.

```
#define COPY(A,B,N) { int i; for(i=0; i<N; i++) A[i]=B[i]; }
```

Вона працює, але не небезпечно по відношенню до типу. Користувач легко змішує типи, які вимагають відповідних перетворень. Однак, за відсутності відповідних перетворень і сигнатур, не буде робитися ніяких дій. Шаблони забезпечують для цього наступний поліморфний мовний механізм:

```
template <class TYPE> void copy(TYPE a[ ], TYPE b[ ], int n)  // однакові типи масивів
{ for(int i=0; i<n; i++)        a[i] = b[i]; }
```

Виклик **copy()** зі специфічними параметрами змушує компілятор на підставі цих параметрів, генерувати реальну (шаблонну) функцію. Якщо це неможливо, то виникає помилка під час компіляції.

```
double f1[50], f2[50];        copy(f1, f2, 50);
char c1[25], c2[50]; copy(c1, c2, 10);
int i1[75], i2[75];      copy(i1, i2, 40);
char *ptr1, *ptr2;     copy(ptr1, ptr2, 100);
copy(i1, f2, 50);              // помилка
copy(ptr1, f2, 50);            // помилка
```

Останні два виклики призведуть до помилки компіляції. Типи фактичних параметрів не відповідають шаблону.

Запис наступного вигляду не викличе помилки при компіляції

```
copy(i1, (int *)f2, 50);
```

Однак при цьому результат буде непередбачуваним. Справа в тому, що в цьому випадку процедура копіювання отримує у вигляді параметрів два класи різного типу.

У наступній формі виконується по-елементне перетворення. Воно є найбільш безпечним перетворенням при даному копіюванні:

```
template <class T1, class T2>  // типи можуть бути різними
void copy(T1 a[ ], T2 b[ ], int n)
        { for(int i=0; i<n; i++) a[i]=b[i]; }
```

Приклад 3.2: Реалізація сортування простими обмінами («метод пухирця»).

```
// Узагальнена реалізація функції сортування простими обмінами
template <class T>  // шаблон функції
void BubbleSort( T *array, int n )
{    for( int i = 1; i < n; i++ )
```

```
      for( int j = n-1; j>=i; j-- )
         if( array[ j ] < array[ j-1 ] )
          { T tmp = array[ j ];    array[ j ] = array[ j-1 ];   array[ j-1 ] = tmp;      }        }
void main()  // перевірка роботи
{   int a[ 8 ] = { 9, 2, 7, 7, 8, 3, 8, 6 };
   BubbleSort( a, 8);   }
```

Програма налаштована на обробку масивів цілих чисел. Перевірка для інших типів та вивід результатів на консоль не вимагає додаткових пояснень і може бути виконана самостійно.

3.1.5. Шаблони класів

Як вже показано раніше, C++ використовує ключове слово **template** для того, щоб забезпечити *параметричний поліморфізм*. Параметричний поліморфізм дозволяє одному і тому ж програмному коду використовуватися щодо різних типів, де *тип – параметр коду*. Параметричний поліморфізм особливо корисний при визначенні контейнерних класів. Обробка даних в контейнерному класі має одну і ту ж форму, незалежно від типу. Шаблони визначення класу і шаблони визначення функції дають можливість багаторазово використовувати код простим способом, безпечним по відношенню до типу, який дозволяє компілятору автоматизувати процес реалізації типу. Поліморфізм забезпечує багаторазове використання коду.

Прийнятий в C++ спосіб визначення множини класів з параметризованим внутрішнім типом даних називається ***шаблоном*** (template). Синтаксис *шаблону класу* розглянемо на прикладі шаблону класу векторів, що містять динамічний масив покажчиків на змінні заданого типу.

```
template <class T>  class vector // "T" – параметр, vector – ім'я шаблону
{  int  tsize;            // загальна кількість елементів
   int  csize;            // поточна кількість елементів
  T  **obj;               // масив покажчиків на параметризовані
                          // об'єкти типу "T"
  public:
   T  *operator[ ](int);        // operator[ ](int) повертає покажчик
                                // на параметризований об'єкт класу "T"
   void  insert(T*);            // функція вставки в вектор об'єкта типу "T"
   int  index(T*);              // функція визначення номеру об'єкта у векторі
};
```

Даний шаблон може використовуватися для породження об'єктів-векторів, кожен з яких зберігає об'єкти певного типу. Ім'я класу при цьому складається з імені шаблону **vector** і типу даних (класу) в кутових дужках '**< тип >**', який підставляється замість параметра '**T**':

```
vector<int>  a;              // a – об’єкт класу  vector<int>
vector<double>  b;           // b – об’єкт класу  vector<double>
extern class time;
vector<time>  c;             // c – об’єкт класу  vector<time>
```

Зауважимо, що транслятором при визначенні кожного вектора з новим типом об'єктів генерується опис нового класу за заданим шаблоном. Так при оголошенні об'єкта '**a**' з типом **vector<int>** транслятором буде згенеровано наступне визначення *шаблонного класу*:

```
class vector<int>
{       int     tsize;
        int     csize;
        int     **obj;
  public:
        int     *operator[ ](int);
        void    insert(int*);
        int     index(int*);          };
```

Очевидно, що елементи-функції шаблону також повинні бути параметризовані, тобто генеруватися для кожного нового типу даних. Тому функції-елементи шаблону класів в свою чергу також є *шаблонами функцій* з тим же самим параметром. Те ж саме стосується перевизначених операцій:

```
// Шаблон функції перевантаження операції "[ ]"
template <class T>  T* vector<T>::operator[ ](int n)
{ if (n >= tsize) return(NULL);
  return (obj[n]);  }

// Шаблон функції index(T*);
template <class T> int vector<T>::index(T *pobj)
{ int  n;
  for (n=0; n<tsize; n++)
    if (pobj == obj[n]) return(n);
  return(-1);
}
```

При цьому самі шаблони функцій повинні розміщуватися в тому ж заголовочному файлі, де розміщується визначення шаблону самого класу.

Транслятором при визначенні кожного вектора з новим типом об'єктів генерується набір шаблонних функцій за заданими шаблонами (в процесі трансляції).

```
// Шаблонна функція перевантаження операції "[ ]"
// для класу vector<int>
int* vector<int>::operator[ ](int n)
{ if (n >=tsize) return(NULL);
    return (obj[n]);  }

// Шаблонна функція index(T*)
// для класу vector<int>
int vector<int>::index(int *pobj)
{ int  n;
  for (n=0; n<tsize; n++)
     if (pobj == obj[n]) return(n);
  return(-1);  }
```

Шаблони можуть мати також і *параметри-константи*, які використовуються для статичного визначення розмірностей внутрішніх структур даних. Крім того, шаблон може використовуватися для розміщення не тільки посилань на параметризовані об'єкти, але і самі об'єкти. Як приклад розглянемо шаблон для побудови циклічної черги обмеженого розміру для параметризованих об'єктів.

```
template <class T, int size> class FIFO
{  int  fst, lst;  // Покажчики на початок-кінець черги
   T  area[size];       // Масив об'єктів класу "T"
                        // розмірності "size"
 public:
   T  from();           // Функція виключення
   void  into(T);       // Функція включення
   FIFO();              // Конструктор
};

template <class T, int size> FIFO<T, int>::FIFO()
{  fst = lst = 0;  }

template <class T, int size> T FIFO<T, int>::from()
{  T work;
   if (fst != lst)
    { work = area[lst++];
```

```
      lst = lst % size; }
   return(work);  }

template <class T, int size> void FIFO<T, int>::into(T obj)
{ area[fst++] = obj;
  fst = fst % size;  }
```

Приклад використання:

```
FIFO<double,100>  a;
FIFO<int,20>      b;
struct  x {};
FIFO<x,50>        c;
```

В цьому прикладі для об'єкта '**a**' компілятор генерує шаблонний клас:

```
class FIFO<double,100>
{  int    fst,lst;
   double  area[100];
  public:
   double  from();
   void  into(double);
   FIFO(); };

FIFO<double,100>::FIFO()
{  fst = lst = 0;  }

double FIFO<double,100>::from()
{ double work;
  if (fst !=lst)
   { work = area[lst++];
     lst = lst % 100;  }
  return(work); }

void FIFO<double,100>::into(double obj)
{  area[fst++] = obj;
   fst = fst % 100;  }
```

Дружність

Шаблони класів можуть містити друзів. **friend**-функція, що не використовує специфікацію шаблону, буде універсальної **friend** для всіх примірників шаблону класу. **friend**-функція, яка включає шаблони параметрів – **friend** тільки для того класу, екземпляр якого створюється.

```
template <class T> class matrix
{  friend void foo_bar();                    // універсальна
```

```
  friend vect <T> product(vect <T> v);    // створюється екземпляр
};
```

Дружні функції можуть бути оголошені для всього класу.

```
template<class T>class Person {
friend class Family;
freind class Acquaintance;
template<class U> freind class Neighbor;
};
```

Тут для кожного типу **T** всі функції-члени класу **Family** – це функції, дружні **Person<T>**. Для будь-якого типу, скажімо для **int**, всі функції-члени класу **Acquaintance<int>** є друзями **Person<int>**, але не **Person<char>**, або будь-якого іншого типу. Для кожного типу **T** і кожного типу **U** всі фунции-члени **Nieghbor<U>** є друзями **Person<T>**.

Дружні функції можуть також бути друзями нешаблонних класів.

Статичні члени

Статичні члени не є універсальними, а є специфічними для кожної реалізації.

```
template <class T> class foo
{ public:
  static int count;  };
…  …  …

// Визначити, як шаблон для foo< T >
template <class T> int foo< T >::count=0;
… …
foo<int> a;
foo<double> b;
```

Визначено дві різні статичні змінні цілого типу **foo<int>::count** і **foo<double>:: count**.

Успадкування

Параметризрвані класи можуть повторно використовуватися, завдяки успадкуванню. Як шаблони, так і успадкування є механізмом для повторного

використання коду і можуть включати поліморфізм. Вони – відмінні риси C++ і можуть комбінуватися в різних формах.

Шаблони класів можуть бути породжені як від нешаблонних класів, так і від класів шаблонів. А також можуть породжувати як нешаблонні класи, так і класи-шаблони. Коли від класу-шаблону породжується нешаблонний клас, всім параметрам класу-шаблону повинні бути присвоєні деякі «реальні» значення. У прикладі це **int**.

```
class A { /*...*/ }
template <class T> class B: public A { /*...*/ }   // від не шаблону
template <class T> class C: public B { /*...*/ }   // від шаблону
class D: public C <int> { /*...*/ }                // не шаблону від шаблону
```

У деяких ситуаціях використання шаблону веде до неприпустимих витрат, що виражається в розмірі об'єктного модуля. Кожен реалізований клас-шаблон вимагає власного відкомпільованого об'єктного модуля.

3.1.6. Параметризація класів. Приклад програми

Задача

Розробити шаблон класу **vect** для зберігання одновимірного масиву. До розробленого класу включити методи (функції-методи класу) сортування масиву методом обмінного сортування та методом вибору. Функції сортування мають бути визначеними поза межами визначення класу.

```
#include <iostream>
#include <string.h>
using namespace std;
//      Класс об’єктів користувача (будь-який клас)
class my_Type
{       int len;
        char *name;
public:
        my_Type() { len=0; name=NULL;}
        my_Type(char *id) { len=strlen(id); name=new char [len+1]; strcpy(name, id);}
        // Конструктор копіювання може бути необхідним для інших класів
        char *get() { return name;}
        int operator < (const my_Type &El)
        { if (name[0] < El.name[0]) return 1;
          else                      return 0; }
};
```

```
template <class T > class vect
{       T *p;                   // базовий покажчик
        int size;               // кількість елементів
  public:
        vect();                         // пустий вектор  розміром 10
        vect(const T a[ ], int n);      // ініціалізація від масиву
        ~vect() {  delete [size] p;  };  // деструктор

        int maxN(int n);                // пошук номеру максимального елемента
        void swap(T &x, T &y);          // обмін
        void ChoiceSort( );             // сортування вибором
        void BubbleSort( );             // сортування простими обмінами
        void print();
        };

// Визначення функцій-методів
template <class T> // створює пустий вестор розміром 10
vect<T>::vect() { size=10; p=new T[size]; }

template <class T> // ініціалізація від масиву
vect< T >::vect( const T a[ ], int n)
{ size=n; p=new T [size]; for( int i=0; i<size; i++) p[ i ]=a[ i ]; }

template <class T> // print
void vect<T>::print()
{ for( int i=0; i<size; i++) cout << p[i] << " "; cout << endl; }

// print - перевантаження для my_Type
void vect<my_Type>::print( )
{ for(int i=0; i < size; i++) cout << p[i].get()<< "   "; cout << endl;}

template <class T>
int vect<T>::maxN(int n)
{   int m = 0;
        for(int i=1; i<=n; i++) m = p[i] < p[m] ? m : i;
        return m;       }

template <class T>
void vect<T>::swap(T &x, T &y)
{       T temp;         temp=x;         x=y;    y=temp;         }

template <class T>
void vect<T>::ChoiceSort( )
{    for( int i = size-1; i > 0; i-- )  swap(p[i], p[maxN(i)]);  }

template <class T>
void vect<T>::BubbleSort( )
{       for( int i = 1; i < size; i++ )
```

```
                for( int j = size-1; j>=i; j-- )
                {       if( p[ j ] < p[ j-1 ] )    swap(p[ j ], p[ j-1 ]);
                }       }

void main()
{       my_Type mas[] = { "David", "Sergo", "Bill", "Artur"};
        int     mas_int[] = {2, 4, 3, 10, 8, -5};
        float mas_float[] = {2.0, 4.0, 3.0, 10.0, -8.0, 5.0};
        double mas_double[] = {2.0, 4.0, -3.0, 10.0, 8.0, 5.0};

        cout << "Sorting object my_Type  \n";
        vect <my_Type> v1( mas, 4);
//      v1.print();
        v1.ChoiceSort( );    // v1.BubbleSort(  );
        v1.print();

        cout << "Sorting int\n";
        vect<int> v2(mas_int, 6);
        v2.ChoiceSort( );    // v2.BubbleSort( );
        v2.print();

        cout << "Sorting float\n";
        vect<float> v3(mas_float, 6);
        v3.ChoiceSort( );    // v3.BubbleSort(  );
        v3.print();

        cout << "Sorting double\n";
        vect<double> v4(mas_double, 6);
        v4.ChoiceSort( );    // v4.BubbleSort(  );
        v4.print();     }
```

Результа роботи програми:

```
Sorting object my_Type
Artur   Bill   David   Sergo
Sorting int
-5 2 3 4 8 10

Sorting float
-8 2 3 4 5 10

Sorting double
-3 2 4 5 8 10
```

3.2. Стандартні бібліотеки мови С++

3.2.1. Бібліотека STL

Стандартна бібліотека мови С++ – це бібліотека, визначена стандартом мови. Реалізація мови С++, що відповідає стандарту, повинна реалізувати цю бібліотеку. Насправді стандартна бібліотека С++ – це набір, що складається з наступних частин:

1. Адаптована стандартна бібліотека шаблонів (**STL**).
2. Бібліотека введення-виведення потоків **iostream**.
3. Засоби визначення національних особливостей реалізації мови С++ (позначення грошової одиниці, представлення чисел: десяткова крапка або кома, запис великих чисел, спосіб запису часу і дати тощо).
4. Клас-шаблон для представлення рядків символів.
5. Клас-шаблон для комплексних чисел.
6. Можливості опису конкретної обчислювального середовища, зокрема діапазонів чисел.
7. Засоби управління пам'яттю.
8. Засоби підтримки різних мов, зокрема повідомлень на різних мовах.
9. Попередньо визначені класи виняткових ситуацій.
10. Стандартна бібліотека С.

Слід зазначити, що всі частини, незважаючи на те, що багато хто з них вже існували задовго до появи стандартної бібліотеки С++, при включенні в бібліотеку зазнали суттєвої переробки. Була перероблена як система файлів-заголовків (навіть імена стали іншими – зник суфікс '**.h**'), так і самі функції і класи.

Розробники домагались, щоб бібліотека виглядала не еклектичним зібранням всього що було доступно, а дійсно єдиною-бібліотекою. Зокрема цим пояснюється широке використання шаблонів, повідомлень про помилки за допомогою виняткових ситуацій, використання іменованого контексту **std**.

Стандартна бібліотека шаблонів С++

Значну частину бібліотеки класів С++ складає стандартна бібліотека шаблонів (**STL** – Standard Template Library). Бібліотека **STL** надає шаблонні класи та функції загального призначення, які реалізують багато популярних і часто використовуваних структур даних. Наприклад, вона включає підтримку

векторів, *списків*, *черг* і *стеків*, а також визначає різні процедури, що забезпечують доступ до цих об'єктів. Оскільки бібліотека **STL** складається з шаблонних класів, алгоритми і структури даних можуть бути застосовані до даних практично будь-якого типу.

STL – це велика бібліотека, і тому не всі її можливості повністю розглядаються в даному виданні. Крім того, розглядається версія бібліотеки **STL** , яка визначається стандартом ANSI/ISO C++. Різні версії компіляторів можуть представляти версію **STL** , яка дещо відрізняється від стандартної.

Принципи побудови бібліотеки полягають у тому, що бібліотека включає ортогональні реалізації контейнерів (колекцій), ітераторів і алгоритмів. Ортогональность або незалежність означає, що контейнери, ітератори і алгоритми можуть поєднуватися практично в будь-якій комбінації, наприклад, алгоритм не залежить від того, який саме тип колекції використовується. Крім того, бібліотека **STL** практично не використовує спадкування.

Ядро стандартної бібліотеки шаблонів складають три основні елементи: *контейнери*, *алгоритми* і *ітератори*.

Контейнери – це об'єкти, що містять інші об'єкти. Існує кілька різних типів контейнерів (табл. 3.1). Наприклад, клас **vector** визначає динамічний масив, клас **queue** створює двосторонню чергу, а клас **list** забезпечує роботу з лінійним списком. Ці контейнери називаються ***послідовними*** контейнерами, оскільки в термінології **STL** послідовність являє собою послідовне розташування і доступ. Крім базових контейнерів, бібліотека **STL** визначає ***асоціативні*** контейнери, які дозволяють ефективно знаходити потрібні значення на основі заданих ключових значень (ключів). Наприклад, клас **map** забезпечує доступ до значень з унікальними ключами, тобто зберігати пару «ключ/значення» і надає можливість знаходити значення по заданому ключу.

Таблиця 3.1 – Контейнерні класи бібліотеки **STL**

Контейнер	**Опис**	**Файл-заголовок**
bitset	Бітова множина	**<bitset>**
deque	Дек (двостороння черга, або черга з двостороннім доступом)	**<deque>**
list	Лінійний список	**<list>**
map	Відображення. Зберігає пари «ключ / значення», в яких кожен ключ пов'язаний лише з одним значенням	**<map>**

Контейнер	Опис	Файл-заголовок
multimap	Мультивідображення. Зберігає пари «ключ / значення», в яких кожен ключ може бути пов'язаний з двома або більше значеннями	**<map>**
multiset	Множина, в якій кожен елемент необов'язково унікальний (мультимножина)	**<set>**
priority_queue	Пріоритетна черга	**<queue>**
queue	Черга	**<queue>**
set	Множина, в якій кожен елемент унікальний	**<set>**
stack	Стек	**<stack>**
vector	Динамічний масив	**<vector>**

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

Алгоритми працюють з даними у контейнерах. Вони забезпечують можливості ініціалізації, сортування, пошуку і перетворення вмісту контейнерів. Багато алгоритмів працюють з групою (або діапазоном) елементів всередині контейнера.

Ітератори – це об'єкти, які в тій чи іншій мірі є покажчиками. Вони дозволяють циклічно опитувати вміст контейнера практично так само, як це робить покажчик для циклічного опитування елементів масиву. Існує п'ять типів ітераторів (табл. 3.2).

Таблиця 3.2 – Типи ітераторів бібліотеки **STL**

Ітератори	Дозволений доступ
Довільного доступу (random access)	Зберігають і зчитують значення; дозволяють організувати довільний доступ до елементів
Двонаправлені (bidirectional)	Зберігають і зчитують значення; забезпечують інкрементно-декрементне переміщення
Однонаправлені (forward)	Зберігають і зчитують значення; забезпечують тільки інкрементне переміщення
Вхідні (input)	Зчитують, але не записують значення; забезпечують тільки інкрементне переміщення
Вихідні (output)	Записують, але не зчитують значення; забезпечують тільки інкрементне переміщення

3.2.2. Вбудовані потоки C++. Перевантаження операцій вводу-виводу

Механізм введення-виведення в C++ називається ***потоком***. Назва походить від того, що інформація вводиться і виводиться у вигляді потоку байтів – символ за символом.

Вбудовані потоки C++

Бібліотека **iostream** стандарту C++ автоматично відкриває такі потоки (табл. 3.3).

Таблиця 3.3 – Стандартні потоки бібліотеки **iostream**

Потік	Значення
cin	Стандартний ввід
cout	Стандартний вивід
cerr	Стандартний вивід помилок
clog	Буферизована версія потоку cerr
wcin	Версія потоку cin для двобайтових символів
wcout	Версія потоку cout для двобайтових символів
wcerr	Версія потоку cerr для двобайтових символів
wclog	Версія потоку clog для двобайтових символів

За замовчуванням стандартні потоки використовуються для зв'язку з консоллю. Однак в середовищах, які підтримують можливість перенаправлення вводу-виводу (наприклад, в DOS, UNIX і Windows), стандартні потоки можуть бути перенаправлені на інші пристрої або в файли.

Клас **istream** реалізує потік введення, клас **ostream** – потік виведення. Ці класи визначені в файлі заголовків **iostream**. Бібліотека потоків введення-виведення визначає три глобальних об'єкта: **cout**, **cin** і **cerr**. **cout** називається стандартним виводом, **cin** – стандартним вводом, **cerr** – стандартним потоком повідомлень про помилки, **cout** і **cerr** виводять на термінал і належать до класу **ostream**, **cin** має тип **istream** і вводить дані з терміналу. Різниця між **cout** і **cerr** істотна в **Unix** – вони використовують різні дескриптори для виведення. В інших системах вони існують більше для сумісності.

Заголовки вводу-виводу

Система введення-виведення, що відповідає стандарту C++, спирається на ряд файлів-заголовків, перерахованих в таблиці нижче (табл. 3.4).

Таблиця 3.4 – Основні заголовки бібліотеки **iostream**

Заголовок	Призначення
<fstream>	Файли вводу-виводу
<iomanip>	Параметризовані маніпулятори вводу-виводу
<ios>	Базова підтримка вводу-виводу
<iosfwd>	Попередні оголошення, що використовуються системою вводу-виводу
<iostream>	Загальні операції вводу-виводу
<istream>	Базова підтримка вводу
<ostream>	Базова підтримка виводу
<sstream>	Потоки, орієнтовані на роботу з рядками
<streambuf>	Підтримка операцій вводу-виводу нижнього рівня

Деякі з цих файлів використовуються для внутрішніх потреб системи введення-виведення. Загалом, програма включає лише заголовочні файли <**iostream**>, <**fstream**>, <**sstream**> і <**iomanip**>.

Вивід здійснюється за допомогою операції '**<<**', введення за допомогою операції '**>>**'. Вираз

```
cout << "Приклад виводу: " << 34;
```

надрукує на терміналі рядок "**Приклад виводу:**", за яким буде виведено число **34**. Оператори

```
int x;
cin >> x;
```

вводять ціле число з терміналу в змінну **x**.

У класі **iostream** операції '**>>**' та '**<<**' визначені для всіх вбудованих типів мови C++ і для рядків (тип **char** *). Якщо необхідно використовувати такий же запис для введення і виведення інших класів, визначених у програмі, для них потрібно перевантажити ці операції, як показано в наступному прикладі.

```
class String
{ public:
        friend ostream& operator<<(ostream & os, const String & s);
        friend istream& operator>>(istream & is, String & s);
 private:
        char* str;
        int length;
};
```

```
ostream & operator<<(ostream & os, const String & s)
{       os << s.str;
return os;      }

istream & operator>>(istream & is, String & s)
{       char tmp[1024];       // максимальна довжина рядка
is >> tmp;
if (s.str != 0)
{       delete [] s.str;        }
length = strlen(tmp);
s.str = new char[length + 1];
if (s.str == 0)
{       // обробка помилок
length = 0;
return is;      }
strcpy(s.str, tmp);
return is;      }
```

Як показано в прикладі класу **String**, операція '<<', *по-перше*, не метод класу **String**, а окрема *дружня* класу функція. Вона і не може бути методом класу **String**, оскільки її перший операнд – об'єкт класу **ostream**. З погляду запису, вона могла б бути методом класу **ostream**, але тоді з додаванням нового класу доводилося б модифікувати клас **ostream**. Коли ж операція '<<' реалізована як окрема функція, достатньо в кожному новому класі її визначити і можна використовувати запис:

```
String x;
cout << "this is a String: " << x;
```

По-друге,_операція '<<' повертає в якості результату посилання на потік виведення. Це дозволяє використовувати її у виразах типу наведеного вище, що з'єднують кілька операцій виведення в один вираз.

Аналогічно реалізована операція введення. Для класу **istream** вона визначена для всіх вбудованих типів мови C++ і покажчиків на рядок символів. Якщо ж необхідно, щоб клас, визначений у програмі, дозволяє введення з потоку, для нього потрібно визначити операцію '>>' в якості функции **friend**.

Не слід визначати функції перевантаження операцій '>>' та '<<' у вигляді методів класу (незважаючи на формальну легальність таких визначень), наприклад:

```
class MyComplex {
// ... public:
```

```
ostream& operator<<( ostream & ); // Поганий стиль!
istream& operator>>( istream & ); // Поганий стиль!
};
```

В цьому випадку при введенні-виведенні об'єктів класу **MyComplex** доведеться записувати операції введення-виведення в незвичній формі (об'єкт – зліва від операції, а потоковий об'єкт – праворуч):

```
MyComplex c;
c << cout; // Поганий стиль!
c >> cin;  // Поганий стиль!
```

Це працює, але суперечить стандартним угодами про записи операцій введення-виведення, і, безумовно, плутає читача програми.

Підсумки

1. Операції '>>' та '<<' визначені для всіх вбудованих типів мови C++ і для рядків (тип **char** *). Якщо необхідно використовувати такий же запис для введення і виведення інших класів, визначених у програмі, для них потрібно перевантажити ці операції.
2. Потік **cout** – об'єкт класу **ostream**, потік **cin** – об'єкт класу **istream**, які визначені у відповідних файлах-заголовках.
3. Щоб можна було з'єднати кілька операцій введення-виведення в один вираз, операції '>>' та '<<' повертають посилання на відповідні потоки введення і виведення, тому ці потоки необхідно передавати через список параметрів.
4. Доступ до закритих членів користувальницького класу в операторі перевантаження можна забезпечити, або за допомогою дружніх функцій, або зробивши функцію **operator** членом класу. Перший випадок продемонстрований в прикладі для класу користувача **String**.
5. У другому випадку при введенні-виведенні об'єктів класу **String** доведеться записувати операції введення-виведення в незвичній формі (об'єкт – ліворуч від операції, а потоковий об'єкт – праворуч).

Маніпулятори та форматування вводу-виводу

Часто необхідно вивести рядок або число в певному форматі. Для цього використовуються так звані *маніпулятори*.

Маніпулятори **–** це об'єкти особливих типів, які керують тим, як **ostream** або **istream** обробляють подальші аргументи.

Деякі маніпулятори можуть також виводити або вводити спеціальні символи.

Для використання маніпуляторів з параметрами необхідно підключити відповідний заголовочний файл:

```
#include <iomanip>
```

Маніпулятор **endl** спричиняє вивід символу нового рядка. Інші маніпулятори дозволяють задавати формат виведення чисел (табл. 3.5).

Таблиця 3.5 – Дії маніпуляторів бібліотеки **iostream**

Маніпулятор	Дія
endl	при виведенні перейти на новий рядок
ends	вивести нульовий байт (ознака кінця рядка символів)
flush	негайно вивести, спустошити все проміжні буфери
dec	виводити числа в десятковій системі (діє за замовчуванням)
oct	виводити числа в вісімковій системі
hex	виводити числа в шістнадцятковій системі числення
setw(int n)	встановити ширину поля виведення в n символів (n – ціле число)
setfill(int n)	встановити символ-заповнювач; цим символом виведене значення буде доповнюватися до необхідної ширини
setprecision(int n)	встановити кількість цифр після коми при виведенні дійсних чисел
setbase(int n)	встановити систему числення для виведення чисел; n може приймати значення 0, 8, 10, 16, причому 0 означає систему числення за замовчуванням, тобто 10

Використання маніпуляторів просте – їх треба вивести в вихідний потік. Припустимо, необхідно вивести одне і те ж число в різних системах числення:

```
int x = 53;
cout   << "Десятковий вигляд: " <<  dec  << x << endl
       << "Вісімковий вигляд: " << oct << x << endl
       << "Шістнадцятковий вигляд: " << hex << x << endl;
```

Аналогічно використовуються маніпулятори з параметрами. Вивід числа з різною кількістю цифр після коми:

```
double x;
// вивести число в поле загальною шириною 6 символів
// (3 цифри до коми, десяткова крапка
```

```
// і 2 цифри після коми)
cout << setw(6) << setprecision(2) << x << endl;
```

Ті ж самі маніпулятори (за винятком **endl** і **ends**) можуть використовуватися і при введенні. У цьому випадку вони описують, як представлені числа, якім вводяться.

```
int x;
// ввести шістнадцяткове число
cin >> hex >> x;
```

Крім того є маніпулятор, який працює тільки при введенні: **ws**. Цей маніпулятор перемикає потік введення в такий режим, при якому всі пробіли (включаючи табуляцію, переведення рядка, переведення каретки і переводи сторінки) будуть вводитися. За замовчуванням ці символи сприймаються як роздільники між атрибутами введення.

Існує дві фундаментальні відмінності між бібліотеками **iostream** старого стилю і бібліотекою, яка відповідає стандарту ANSI/ISO для мови C++. По-перше, бібліотека старого стилю була визначена в глобальному просторі імен, а бібліотека **iostream** стандарту ANSI/ISO для мови C++ міститься в просторі імен **std**. По-друге, бібліотека старого стилю використовує C-подібні заголовки з розширенням **.h**, а бібліотека стандарту C++ – заголовки в стилі C++ (без розширення **.h**).

Щоб використовувати бібліотеку **iostream** стандарту C++, включіть в свою програму заголовок <**iostream**>. Після цього, швидше за все, доведеться внести цю бібліотеку в поточний простір імен за допомогою наступного оператора.

```
using namespace std;
```

Після використання цього оператора робота як старої, так і нової бібліотек багато в чому збігається.

По суті, немає гострої необхідності в використанні наведеного вище оператора **using**. Замість нього при кожному посиланні на який-небудь член класів введення-виведення можна в явному вигляді використовувати кваліфікатор простору імен. Наприклад, наступний оператор явно посилається на потік **cout**.

```
std::cout << "Це тест";
```

3.2.3. Ввід-вивід файлів

Введення-виведення файлів може здійснюватися як за допомогою стандартних функцій бібліотеки С, так і за допомогою потоків введення-виведення С++. Проте, функції бібліотеки С не забезпечують будь-якого контролю типів.

Файл розглядається як послідовність байтів. Читання або запис відбувається послідовно. Наприклад, якщо при читанні з початку файлу перша операція читання ввела 4 байти, інтерпретовані як ціле число, то наступна операція читання почне введення з п'ятого байта, і так далі до кінця файлу.

Аналогічно відбувається запис в файл – за замовчуванням перший запис проводиться в кінець наявного файлу, а всі наступні операції запису послідовно пишуть дані один за одним. При операціях читання-запису кажуть, що існує *поточна позиція*, починаючи з якої буде проводитися наступна операція.

Більшість файлів мають можливість прямого доступу. Це означає, що можна робити операції введення-виведення не послідовно, а в довільному порядку: після читання перших 4-х байтів прочитати з 20 по 30, потім два останніх і т.п. При написанні програм на мові С++ можливість прямого доступу забезпечується тим, що поточну позицію читання або запису можна явно встановити.

У бібліотеці С++ для введення-виведення файлів існують класи **ofstream** або **basic_ofstream** (вивід) і **ifstream** або **basic_ifstream** (ввід). Самі операції введення-виведення виконуються так само, як і для інших потоків. Операції '>>' та '<<', визначені для класів введення-виведення файлів (**fstream** і **basic_fstream)**.

При ***виведенні*** інформації в файл, перш за все, потрібно визначити в який файл буде проводитися вивід. Для цього можна використовувати конструктор класу **basic_ofstream** у вигляді:

```
explicit basic_ofstream(const char *_S, ios_base::openmode _M = out | trunc);
```

Перший аргумент – ім'я вихідного файлу і це єдиний обов'язковий аргумент. *Другий аргумент* задає режим, в якому відкривається потік. Цей аргумент – бітове АБО наступних величин (табл. 3.6).

Примітка. Ключове слово **explicit** у визначенні конструктора означає, що даний конструктор не буде використовуватися компілятором для неявного перетворення типів даних.

Таблиця 3.6 – Режими відкриття потоку виводу/виводу

Аргумент	Режим потоку
ios_base::app	при запису дані додаються в кінець файлу, навіть якщо поточна позиція була перед цим переміщена
ios_base::ate	при створенні потоку поточна позиція встановлюється в кінець файлу; проте на відміну від режиму **app** запис ведеться в поточну позицію
ios_base::in	потік створюється для введення; якщо файл вже існує, він зберігається
ios_base::out	потік створюється для виведення (режим за замовчуванням)
ios_base::trunc	якщо файл вже існує, його попередній вміст знищується і довжина файлу ставати рівною нулю; режим діє за замовчуванням, якщо не задані **ios_base :: ate**, **ios_base :: app** або **ios_base :: in**
ios_base::binary	введення-виведення буде відбуватися в двійковому вигляді, за замовчуванням використовується текстове представлення даних

Можна створити потік виводу за допомогою стандартного конструктора без аргументів, а пізніше виконати метод **open()** з такими ж аргументами, як і у конструктора:

```
void open(const char* szName, int nMode = ios::out, int nProt = filebuf::openprot);
```

Тільки після того, як потік створений і з'єднаний з певним файлом (або за допомогою конструктора з аргументами, або за допомогою методу **open**) можна виконувати вивід. Виводяться дані операцією '<<'. Крім того, дані можна вивести за допомогою методів **write** або **put**:

```
basic_ofstream & write(const _E *_S, streamsize _N); // Вивід _N одиниць даних з адреси _S
basic_ofstream & put(_E _X); // Вивід однієї одиниці даних типу _E
```

Метод **write** виводить вказану кількість байтів (**_N**), розташованих в пам'яті, починаючи з адреси **_S**. Метод **put** виводить одну одиницю даних типу **_E**.

Для того щоб перемістити поточну позицію, використовується метод **seekp**:

```
basic_ofstream & seekp(off_type  _O, ios_base::seekdir _W);
```

Перший аргумент – ціле число, зміщення позиції в байтах. Другий аргумент визначає, звідки відраховується зміщення, і він може приймати одне з трьох значень:

– **ios_base :: beg** – зміщення від початку файлу;

– **ios_base :: cur** – зміщення від поточної позиції;

– **ios_base :: end** – зміщення від кінця файлу.

Змістивши поточну позицію, операції виведення тривають з нового місця у файлі.

Після завершення виведення можна виконати метод **close**, який виводить внутрішні буфери в файл і від'єднує потік від файлу. Те ж саме відбувається і при знищенні об'єкта (потоку).

Приклад виведення в новий файл:

```
#include <fstream>
…
//  Відкрити файл output.txt для запису
  basic_ofstream<char>   fOutput( "output.txt" );
   if( !fOutput ) { /* Помилка відкриття */ }
….
// Варіанти виводу в файл:
int x[ ] = {1, 2, 3, 4};
fOutput  <<  x[ 1 ];
fOutput.put( x[ 2 ] );
fOutput.write( x, 4 );          // вивід x[ 0 ]
fOutput.seekp( -4, ios_base::cur );  // встановлення поточної позиції у файлі
fOutput.close();                // закрити файл
```

Клас **basic_ifstream**, який здійснює ***введення*** з файлів, працює аналогічно. При створенні об'єкта типу **ifstream**, як аргумент конструктора потрібно задати ім'я існуючого файла

```
explicit basic_ifstream(const char *_S, ios_base::openmode _M = in);

// Приклад відкриття файлу output.txt для зчитування
  basic_ ifstream <char>   fInput( "output.txt" );
   if( ! fInput) { /* Помилка відкриття */ }
```

Можна скористатися стандартним конструктором, а підключитися до файлу за допомогою методу **open**.

Читання з файлу здійснюється операцією '>>' або методами **read** або **get**:

```
basic_ifstream & read(_E *_S, streamsize _N); // Введення _N одиниць даних на адресу _S
basic_ifstream & get(_E& _X); // Введення однієї одиниці даних типу _E
```

Метод **read** вводить вказану кількість байтів (**_N**) в пам'ять, починаючи з адреси **_S**. Метод **get** вводить одну одиницю даних типу **_E**.

Так само, як і для виведення, поточну позицію введення можна змінити с допомогою методу **seekg**, а після закінчення операцій закрити файл за допомогою **close** або просто знищити об'єкт.

```
basic_ifstream  seekg(off_type _O, ios_base::seekdir _W)
```

У висновку треба відзначити, що наведений опис операцій введення-виведення файлів не є повним. Деякі операції вводу-виводу файлів мають специфіку, яка залежить від конкретної операційної системи і компілятора, тому за деталями краще звертатися до технічної документації компілятора.

Крім того, сумісність зі старою бібліотекою класів забезпечується призначенням псевдонімів:

```
typedef basic_istream<char, char_traits<char> > istream;
typedef basic_ostream<char, char_traits<char> > ostream;
typedef basic_iostream<char, char_traits<char> > iostream;
typedef basic_ifstream<char, char_traits<char> > ifstream;
typedef basic_ofstream<char, char_traits<char> > ofstream;
typedef basic_fstream<char, char_traits<char> > fstream;
```

3.2.4. Алгоритми STL

Алгоритм – це функція, яка виконує деякі дії над елементами контейнера (контейнерів). Алгоритми в **STL** не є методами класів і навіть не є дружніми функціями по відношенню до контейнерів. У нинішньому стандарті мови алгоритми – це незалежні шаблонні функції (табл. 3.7). Їх можна використовувати при роботі як зі звичайними масивами C++, так і з будь-якими власними класами-контейнерами (передбачається, що в клас вже включені деякі базові функції).

Таблиця 3.7 – Деякі типові алгоритми **STL**

Алгоритм	Призначення
find	Повертає перший елемент із зазначеним значенням
count	Вважає кількість елементів, що мають вказане значення
equal	Порівнює вміст двох контейнерів і повертає true, якщо всі відповідні елементи еквівалентні
search	Шукає послідовність значень в одному контейнері, яка відповідає такій же послідовності в іншому
copy	Копіює послідовність значень з одного контейнера в інший (Або в інше місце того ж контейнера)
swap	Обмінює значення, що зберігаються в різних місцях
iter_swap	Обмінює послідовності значень, що зберігаються в різних місцях
fill	Копіює значення в послідовність комірок
sort	Сортує значення в зазначеному порядку
merge	Комбінує два відсортованих діапазони значень для отримання більшого діапазону (злиття)
accumulate	Повертає суму елементів в заданому діапазоні
for_each	Виконує зазначену функцію для кожного елемента контейнера

Приклад застосування алгоритму sort()

Нехай в програмі створено масив типу **int** з наступними даними:

{42, 31, 7, 80, 2, 26, 19, 75};

Для сортування масиву можна застосувати алгоритм **sort**().

```
#include <iostream>
#include <algorithm>                    // для sort ()
using namespace std;
int arr[8] = {42,  31,  7, 80, 2,  26,   19,  75};

int main()
{       sort(arr,  arr+8);  //сортування
        for(int i=0; i < 8; i++)
        cout << "  " << arr[i]; cout << endl;
return 0;
}
```

де **arr** – це адреса початку масиву, **arr+8** – адреса кінця (елемент, розташований першим за останнім у масиві).

Результат:

2 7 19 26 31 42 75 80

Приклад застосування алгоритму find ()

Цей алгоритм шукає перший елемент в контейнері, значення якого дорівнює вказаному. У прикладі показано, як потрібно діяти, якщо необхідно знайти значення в масиві цілих чисел.

```
// знайти перший об'єкт, значення якого дорівнює даному
#include <iostream>
#include <algorithm>                    // для find()
using namespace std;
int arr[] = { 11, 22, 33, 44, 55, 66, 77, 88 };
int main()
{       int* ptr;
        ptr = find(arr, arr+8, 33);    //знайти перше входження 33
        cout << "Перший об’єкт зі значенням 33 знайдено в позиції "
                << (ptr-arr) << endl;
return 0;
}
```

Результат

```
Перший об’єкт зі значенням 33 знайдено в позиції  2
```

:

Перший елемент в масиві має індекс 0, а не 1, тому число 33 було знайдено в позиції 2, а не 3.

Діапазони

Перші два аргументи алгоритмів **find**() та **sort()** визначають діапазон елементів, які переглядаються. Значення задаються ітераторами. В даному прикладі використано значення звичайних покажчиків C++, які, загалом, є окремим випадком ітераторів.

Перший параметр – це ітератор (тобто в даному випадку – покажчик) першого значення, яке потрібно перевіряти. Другий параметр – итератор останньої позиції (він вказує на наступну позицію за останнім потрібним значенням). Так як всього в масиві 8 елементів, це значення дорівнює першій позиції, збільшеної на 8. Це називається «перше значення після останнього». Заданий у такий спосіб інтервал застосування алгоритму називають ***діапазоном***.

3.2.5. Послідовні контейнери STL

Як вже зазначалося, в **STL** реалізовано дві основні категорії контейнерів: *послідовні* і *асоціативні* (є й інші але тут вони не розглядаються). У цьому розділі розглядаються три послідовних контейнери: *вектори*, *списки* та *черги з двостороннім доступом*. Ще майже нічого не сказано про ітератори, тому будемо уникати деяких операцій, які без цих знань не будуть зрозумілі. Про ітератори буде розказано в наступному розділі. Слід також знати, що різні види контейнерів використовують методи з однаковими іменами і характеристиками, тому, наприклад, метод **push_back()** для векторів буде нітрохи не гірше працювати зі списками та чергами.

У *послідовних контейнерах* дані зберігаються подібно до того, як будинки стоять на вулиці – в ряд. Кожен елемент зв'язується з іншими за допомогою номера своєї позиції у ряді. Всі елементи, крім кінцевих, мають по одному сусіду з кожного боку. Прикладом послідовного контейнера є звичайний масив.

Проблема з масивами у тому, що їх розміри потрібно вказувати при компіляції, тобто у вихідному коді. Можна використовувати масиви в динамічній пам’яті (за допомогою **new**), але їх розміри також неможливо змінювати. Деякі компілятори дозволяють не вказувати явно розмір масиву при компіляції, але це, по суті, тіж самі масиви у динамічній пам’яті.

На жаль, дуже часто при написанні програми буває невідомо скільки елементів буде потрібно записати в масив. Тому, зазвичай, розраховують на найгірший варіант, і розмір масиву вказують «із запасом». В результаті під час роботи програми з'ясовується, що у масиві залишається дуже багато вільного місця, або бракує елементів. Це може призвести до чого завгодно, навіть «зависання» програми. У **STL** для боротьби з такими труднощами існує контейнер вектор (**vector**).

Але із масивами є ще одна проблема. Припустимо, в масиві зберігаються дані про працівників, і ви відсортували їх за абеткою відповідно до прізвищ. Тепер, якщо потрібно буде додати співробітника, прізвище якого починається з літери **Л**, то доведеться зсунути всіх працівників із прізвищами на **М-Я**, щоб звільнити місце для вставки нового значення. Все це може тривати досить багато часу. У **STL** є контейнер *список* (**list**), який ґрунтується на ідеї зв'язаного списку і який вирішує це питання.

Третім представником послідовних контейнерів є *черга із двостороннім доступом* (**deq**). Це комбінація стека та звичайної черги. Стек, як відомо, працює за принципом **LIFO** (останній увійшов перший вийшов). Тобто і введення, і виведення даних у стеку виконується зверху. У черзі, навпаки, використовується принцип **FIFO** (перший увійшов – перший вийшов): дані надходять «з хвоста», а виходять «з голови». Черга з двостороннім доступом використовує комбінований порядок обміну даних. І вносити значення в чергу з двостороннім доступом і виводити з неї можна з обох сторін. Це досить гнучкий інструмент, він цінний не тільки сам по собі, але може бути базою для інших контейнерів.

В наступній таблиці надано зведену характеристику послідовних контейнерів **STL**. Для порівняння згадано і звичайний масив (табл. 3.8).

Таблиця 3.8 – Основні послідовні контейнери

Контейнер	**Характеристика**	**Плюси/мінуси**
Звичайний масив	Фіксований розмір	Швидкісний випадковий доступ (за індексом). Повільна вставка або вилучення даних із середини. Розмір не може бути змінено під час роботи програми.
Вектор	Розширюваний масив	Швидкісний випадковий доступ (за індексом) Повільна вставка або вилучення даних із середини. Швидкісна вставка або вилучення даних з хвоста.
Список	Аналогічний зв'язаного списку	Швидкісна вставка або вилучення даних із будь-якого місця. Швидкий доступ до обох кінців. Повільний випадковий доступ.
Черга з двостороннім доступом	Як вектор, але доступ з обох кінців	Швидкісний випадковий доступ. Повільна вставка або вилучення даних із середини. Швидкісна вставка та вилучення даних з хвоста та голови.

Використання в програмах контейнерів вимагає наступних дій:

– підключити до програми відповідний файл-заголовок (директивою **include**);

– оголосити об’єкт (змінну) шаблонного класу відповідного типу; в оголошенні має бути зазначено, об’єкти яких типів будуть зберігатися в контейнері.

```
#include <vector>          // для вектора
#include <list>            // для списку
#include <deque>           // для черги з двостороннім доступом
```

```
vector<int>   aVect;                // оголосити вектор цілих чисел (тип int)
list<airtime>   departure_list;     // оголосити список об'єктів типу airtime
deque<double>   my_deq;             // оголосити чергу  дійсних чисел
```

В **STL** немає обов'язкової необхідності задавати розміри контейнерів, але така можливість існує. Розміри можна змінювати після створення.

Вектори

Вектори – це, так би мовити, розумні масиви. Вони виконують автоматичне розміщенням себе в пам'яті, розширення і звуження свого розміру в міру вставки або видалення даних. Вектори можна використовувати в якійсь мірі як масиви, звертаючись до елементів за допомогою звичайного оператора '**[ ]**'. Випадковий доступ в векторах виконується дуже швидко.

Також досить швидко здійснюється додавання (або проштовхування) нових даних в кінець вектора. Коли це відбувається, розмір вектора автоматично збільшується для того, щоб було, куди покласти нове значення.

Перший приклад стосується найзагальніших векторних операцій.

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

int main()
        {
        vector<int> v;              // створення вектора цілих чисел

        v.push_back(10);            // вставка значень в кінець вектора
        v.push_back(11);
        v.push_back(12);
        v.push_back(13);

        v[0] = 20;                  // заміна
        v[3] = 23;

        for(unsigned int  j=0; j<v.size(); j++)       // вивід вектора
                cout << v[ j ] << ' ';   // 20 11 12 23
  cout << endl;
  return 0;
        }
```

Для створення вектора **v** використовується штатний конструктор вектора без параметрів. Як з будь-якими контейнерами **STL**, для завдання типу змінних,

які будуть зберігатися в векторі, використовується шаблонний формат (в даному випадку це тип **int**). Тут не визначено розмір контейнера, тому спочатку він дорівнює 0.

Метод **push_back()** вставляє значення свого аргументу в кінець/хвіст вектора (кінець/хвіст розташовується там, де знаходиться найбільший індекс). Початок вектора (елемент з індексом 0), на відміну від списків і черг, не може використовуватися для вставки нових елементів. Програма проштовхує значення **10**, **11**, **12** і **13** таким чином, що **v[0]** містить **10**, **v[1]** містить **11**, **v[2]** містить **12**, і **v[3]** містить **13**.

Як тільки в векторі з'являються будь-які дані, до них відразу можна отримати доступ за допомогою перевантаженого оператора '**[ ]**'. Так же, як і при роботі з масивами. Дуже зручно. Цей оператор використовується у прикладі щоб замінити перший елемент з 10 на 20, а останній – з 13 на 23. Таким чином, результат програми буде таким:

```
20 11 12 23
```

Метод **size()** повертає поточне число елементів, що містяться в контейнері. У прикладі – це 4. Цей параметр використовується в циклі **for** для виведення значень вектора на екран.

Є ще один метод, **max_size()**, який не демонструється тут. Він повертає максимальний розмір, до якого може розширюватися контейнер. Це число залежить від типу збережених в ньому даних, типу контейнера та операційної системи. Наприклад, в деяких системах **max_size()** повертає **1 073 741 823** для цілочисельного вектора.

Наступний приклад демонструє ще кілька методів і векторів.

```
// демонстрація конструкторів, методів, swap(), empty(), back(), pop_back()
#include <iostream>
#include <vector>
using namespace std;

int main()
	{                                       // масив чисел типу double
	double arr[] = { 1.1, 2.2, 3.3, 4.4 };

	vector<double> v1(arr, arr+4);      // ініціалізація вектора масивом
	vector<double> v2(4);               // порожній вектор розміром 4 елементи
```

```
v1.swap(v2);                            // дані v1 і v2 міняються місцями

        while( !v2.empty() )            // цикл доки вектор не порожній
                {       cout << v2.back() << ' ';     // вивід останнього елемента
v2.pop_back();          }               // видалення останнього елемента
                                        // результат: 4.4 3.3 2.2 1.1
cout << endl;
return 0;
        }
```

У цій програмі використано два нових конструктора векторів. Перший ініціалізує вектор **v1** значеннями звичайного масиву C++, переданого йому в якості аргументу. Аргументами цього конструктора є покажчики на початок масиву і на елемент «після останнього» (діапазон). У другому конструкторі вектор **v2** ініціалізовано установкою його розміру рівним 4. Обидва вектора можуть містити дані типу **double**.

Метод **swap**() обмінює дані одного вектора на дані іншого, при цьому порядок проходження елементів не змінюється. У цій програмі вектор **v2** порожній, тому він замінюється на вектор **v1**. Результат роботи програми такий:

```
4.4 3.3 2.2 1.1
```

Метод **back**() повертає значення останнього елемента вектора. Значення виведене за допомогою **cout**. Метод **pop_back ()** видаляє останній елемент вектора.

Таким чином, при кожному проходженні циклу останній елемент буде мати різні значення. Зазначимо, що **pop_back()** тільки видаляє останній елемент, але не повертає його значення, як **pop()** при роботі зі стеком. Тому, завжди потрібно використовувати **pop_back**() і **back**() в парі.

Деякі методи, наприклад **swap()**, існують і у вигляді алгоритмів. У таких випадках краще віддати перевагу методу. Робота з методом для конкретного контейнера зазвичай виявляється більш ефективною. Іноді має сенс використовувати і те, і інше. Наприклад, такий підхід можна використовувати для обміну елементів двох контейнерів різних типів.

Методи **insert()** і **erase()**, відповідно, вставляють і видаляють дані при зверненні до довільної позиції контейнера. Їх не рекомендується використовувати з векторами, оскільки в цьому випадку при кожній вставці або видаленні доводиться перекроювати всю структуру – розтискати, стискати вектор. Все це не дуже ефективно. Проте, коли швидкість не грає дуже велику

роль, використовувати ці методи і можна, і потрібно. Наступний приклад показує, як це робиться.

```
// demonstrates insert(), erase()
#include <iostream>
#include <vector>
using namespace std;

int main()
        {
        int arr[ ] = { 100, 110, 120, 130 }; // масив цілих чисел

        vector<int> v(arr, arr+4);           // ініціалізація вектора масивом

  cout << "\nПеред вставкою: ";
  for(unsigned  int j=0; j<v.size(); j++)    // вивід всіх елементів
                cout << v[j] << ' ';

  v.insert( v.begin()+2, 115);               // вставити 115 в 2-й елемент

  cout << "\nПісля вставки:  ";
  for(unsigned  j=0; j<v.size(); j++)        // вивід всіх елементів
                cout << v[j] << ' ';

  v.erase( v.begin()+2 );                    // видалення 2-го елемента

  cout << "\nПісля видалення:    ";
  for(unsigned  j=0; j<v.size(); j++)        // вивід всіх елементів
                cout << v[j] << ' ';
  cout << endl;
  return 0;
        }
```

Метод **inser()** має два параметри: майбутнє розташування нового елементу в контейнері і значення елемента. Додаємо дві позиції до результату виконання методу **begin()**, щоб перейти до елементу № 2 (третій елемент в контейнері, починаючи з нуля). Елементи від точки вставки до кінця контейнера зсуваються, щоб було місце для розміщення елемента, що вставляється. Розмір контейнера автоматично збільшується на одиницю.

Метод **erase**() видаляє елемент з вказаної позиції. Решта елементів зсуваються, розмір контейнера зменшується на одиницю. Ось як виглядають результати роботи програми:

```
Перед  вставкою:  100 110 120 130
```

```
Після  вставки:   100 110 115 120 130
Після  видалення:   100 110 120 130
```

Списки

Контейнер **STL** під назвою ***список*** являє собою двічі зв'язаний список, в якому кожен елемент зберігає покажчик на сусіда ліворуч і праворуч. Контейнер містить адресу першого і останнього елементів, тому доступ до обох кінців списку здійснюється дуже швидко.

Перший приклад роботи зі списками демонструє вставку, читання і вилучення даних.

```
// demonstrates push_front(), front(), pop_front()
#include <iostream>
#include <list>
using namespace std;

int main()
  {
  list<int> ilist;

  ilist.push_back(30);              // вставка елемента в кінець
  ilist.push_back(40);
  ilist.push_front(20);             // вставка елемента перед першим
  ilist.push_front(10);

  int size = ilist.size();          // кількість елементів

  for(int j=0; j<size; j++)
    {
    cout << ilist.front() << ' ';   // читання першого елемента
    ilist.pop_front();              // вилучення першого
    }
  cout << endl;
  return 0;
  }
```

Дані вставлено в кінець і початок списку таким чином, щоб при виведенні на екран і видаленні з початку зберігався б наступний порядок їх слідування:

```
10 20 30 40
```

Методи **push_front()**, **front()** і **pop_front()** аналогічні методам **pop_back()**, **push_back()** і **back()**, які вже використовувались при роботі з векторами.

Треба пам'ятати, що довільний доступ до елементів списку використовувати небажано, так як він здійснюється занадто повільно для нормальної обробки даних. Тому оператор "**[ ]**" навіть не визначений для списків. Якби довільний доступ був реалізований, цьому оператору довелося б проходити весь список, переміщаючись від посилання до посилання, аж до досягнення потрібного елемента. Це була б дуже повільна операція. Якщо програмі може знадобитися довільний доступ до контейнера, використовуйте вектори або черги з двостороннім доступом.

Списки доцільно використовувати при частих операціях вставки і видалення будь-де в середині списку. При цих діях використовувати вектори і черги нелогічно, тому що всі елементи у місці вставки або видалення повинні при цьому бути зрушені. Що стосується списків, то при тих же операціях змінюються лише значення декількох покажчиків. Проте, можуть виникнути ситуації, коли пошук потрібної позиції вставки в списку буде теж досить тривалою операцією.

Для роботи методів **insert()** і **erase()** потрібні ітератори, тому їх викладено в подальших розділах.

Деякі методи використовуються тільки зі списками. Інших контейнерів, для яких вони були б визначені, просто немає, хоча є, звичайно, алгоритми, які виконують практично ті ж функції. У наступному прикладі показані деякі з таких методів. Програма починається з заповнення двох цілочисельних списків вмістом двох масивів.

```
// demonstrates reverse(), merge(), and unique()
#include <iostream>
#include <list>
using namespace std;

int main()
  {
  int j;
  list<int> list1, list2;

  int arr1[] = { 40, 30, 20, 10 };
  int arr2[] = { 15, 20, 25, 30, 35 };

  for(j=0; j<4; j++)
    list1.push_back( arr1[j] );       // list1: 40, 30, 20, 10
  for(j=0; j<5; j++)
    list2.push_back( arr2[j] );       // list2: 15, 20, 25, 30, 35
```

```
  list1.reverse();                  // реверсія list1: 10 20 30 40
  list1.merge(list2);               // об’єднання list2 з list1
  list1.unique();                   // видалення дублювання 20 и 30

  int size = list1.size();
  while( !list1.empty() )
    {
    cout << list1.front() << ' ';   // читати перший елемент
    list1.pop_front();              // видалити (виштовхнути) перший елемент
    }
  cout << endl;
  return 0;
  }
```

Перший список складено з елементів, розташованих в зворотному порядку, тому для початку він перевертається в програмі так, щоб він був упорядкований по зростанню. Після цієї дії списки виглядають так:

```
10 20 30 40
15 20 25 30 35
```

Потім виконується функція **merge()**. З її допомогою об'єднуються списки **list2** і **list1**, результат зберігається в **list1**. Його новий вміст:

```
10 15 20 20 25 30 30 35 40
```

Після цього, застосовано метод **unique**() до списку **list1**. Ця функція знаходить сусідні елементи з однаковими значеннями і залишає тільки один з них. Вміст списку **list1** виводиться на екран:

```
10 15 20 25 30 35 40
```

Для виведення списку використовуються функції **front**() і **pop_front()**. Кожен елемент по черзі від голови до хвоста виводиться на екран, потім виштовхується зі списку. Попереду після кожного кроку з'являється новий елемент. В результаті виходить, що операція виведення на екран знищує список. Можливо це не завжди те, що потрібно, але, тим не менш, це єдиний спосіб продемонструвати можливості послідовного доступу до списків. Ітератори в цьому сенсі спрощують справу, але їх розгляд наведено далі.

Черги с двостороннім доступом

Черга з двостороннім доступом (**deque**) являє собою щось схоже і на вектор, і на зв'язаний список. Як і вектор, цей тип контейнера підтримує довільний доступ (операція '**[ ]**'). Як і до списку, доступ може бути отриманий до початку і кінця черги. В цілому це нагадує вектор з двома кінцями. Підтримуються функції **front**(), **push_front**() і **pop_front**().

Для векторів і двосторонніх черг пам'ять резервується по-різному. Вектор завжди займає суміжні осередки пам'яті. Тому при його розростанні резервується нова ділянка пам'яті, де контейнер може помістится цілком. З іншого боку, черга з двостороннім доступом може розміщуватися в декількох сегментах пам'яті, не обов'язково суміжних. У цьому є і плюси, і мінуси. Черга практично завжди буде гарантовано розміщена в пам'яті, але доступ до сегментованих об'єктів завжди здійснюється з більш повільною швидкістю. Метод **capacity**() повертає найбільше число елементів вектора, які можна розмістити в пам'яті без будь-яких додаткових дій, але вона не була вказана для контейнера **deque** (черга з двостороннім доступом), оскільки в даному випадку ніяких переміщень по пам'яті не потрібно.

```
// demonstrates push_back(), push_front(), front()
#include <iostream>
#include <deque>
using namespace std;

int main()
  {
  deque<int> deq;

  deq.push_back(30);       // проштовхування через хвіст
  deq.push_back(40);
  deq.push_back(50);
  deq.push_front(20);      // проштовхування через голову
  deq.push_front(10);

  deq[2] = 33;             // зміна середнього елемента

  for(unsigned int j=0; j<deq.size(); j++)
    cout << deq[j] << ' ';     // вивід
  cout << endl;
  return 0;
  }
```

Вище за текстом вже розглянуто приклади використання методів **push_back()**, **push_front()** і операції '**[ ]**'. До черг з двостороннім доступом вони

застосовуються точно так же, як і до інших контейнерів. Результат роботи програми:

```
10 20 33 40 50
```

На рисунку 3.2 показана робота деяких особливо важливих методів для трьох послідовних контейнерів.

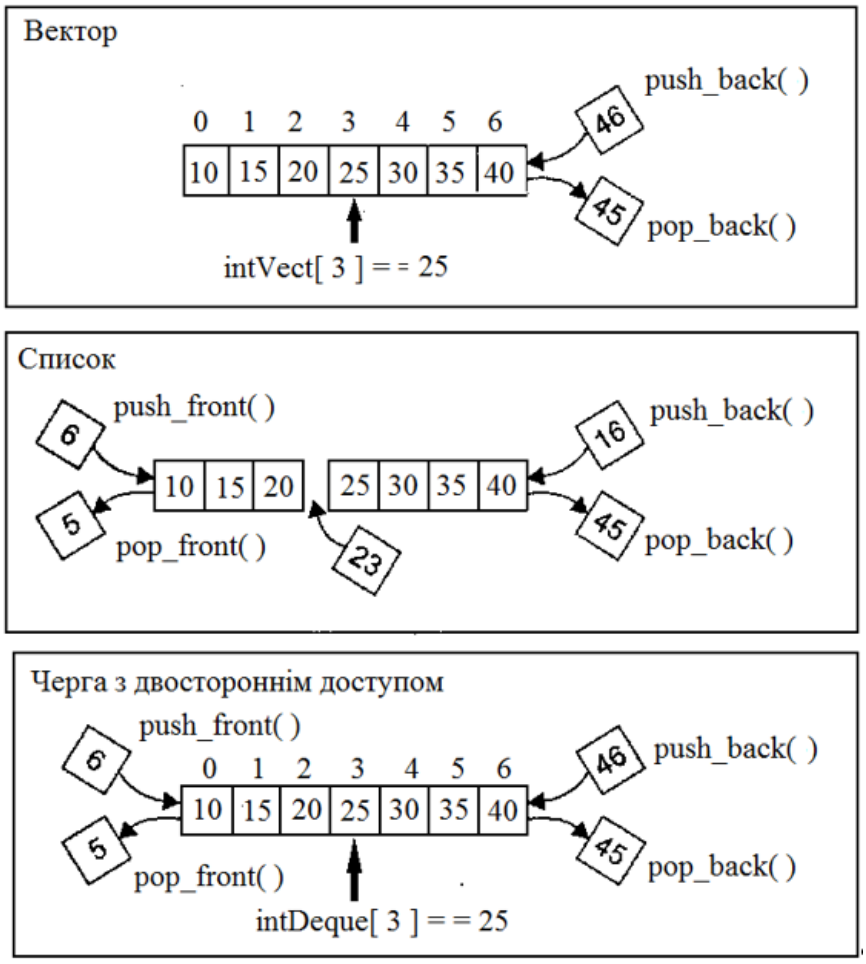


Рисунок 3.2 – Послідовні контейнери

3.2.6. Ітератори стандартної бібліотеки шаблонів. Відповідність алгоритмів контейнерам

Ітератори відіграють важливу роль для успішного виконання дій в **STL**. Їх можна розглядати у якості «інтелектуальних покажчиків», або як «мости», що з'єднують алгоритми з контейнерами.

Ітератори як інтелектуальні покажчики

Часто необхідно виконувати певну операцію над усіма елементами контейнера або над якимось діапазоном даних. Наприклад, це може бути операція для виведення вмісту контейнера на екран або обчислення суми всіх елементів. У звичайних масивах C++ для цього використовується обробка в циклі всіх елементів за допомогою покажчика (або оператора '**[ ]**', що також грунтується на механізмі покажчиків). Наприклад, у наступному фрагменті коду проводиться прохід (ітерація) масивом типу **float** з виведенням кожного елемента:

```
float * ptr = start_address;
for( int j=0; j < SIZE;  j++)
cout << *ptr++;
```

Для отримання значення елемента до покажчика **ptr** застосовано операцію '*****', після чого виконується інкремент покажчика **ptr**, використовуючи '++'. Після цього значенням покажчика є адреса наступного елемента контейнера.

Застосовувати покажчики до більш складних контейнерів, ніж прості масиви, доволі складно. Якщо елементи контейнера зберігаються не послідовно в пам’яті, а сегментовано, то методи доступу до них значно ускладнюються. Неможливо в цьому випадку просто інкрементувати покажчик для отримання наступного значення. Наприклад, при просуванні від елемента до елемента у зв’язаному списку неможливо припускати, що наступний елемент є сусіднім до попереднього. У списку треба рухатися по ланцюжку посилань.

В **STL** вирішенням цієї проблеми є створення класу «*інтелектуальних покажчиків*». Об’єкти цього класу також використовують методи роботи зі звичайними покажчиками. Але операції '++' та '*****' перевантажуються і тому вони «знають» як треба працювати з елементами контейнера, навіть якщо вони розташовані в пам’яті не послідовно або змінюють своє положення.

В **STL** інтелектуальні покажчики – цілком незалежні шаблонні класи, які мають власне ім’я – ***ітератори***. Для створення ітераторів необхідно оголошувати їх в програмі у якості об’єктів таких класів.

Крім того ітератори відіграють ще одну важливу роль. Вони визначають, які алгоритми можна використовувати з якими контейнерами. Теоретично можливо застосовувати будь-який алгоритм до будь-якого контейнера (ортогональність **STL**). Це одна з переваг **STL**. Але правил без виключень не буває. Наприклад, алгоритму **sort()** потрібен довільний доступ по індексу до елементів контейнера, що неможливо при роботі зі списком. Ітератори надають можливість визначати відповідність алгоритмів контейнерам.

В **STL** використовується п’ять типів ітераторів (рис. 3.3). В порядку збільшення їхньої складності це:

– вхідні та вихідні ітератори;
– прямі ітератори;
– двоспрямовані ітератори;
– ітератори довільного (випадкового) доступу.

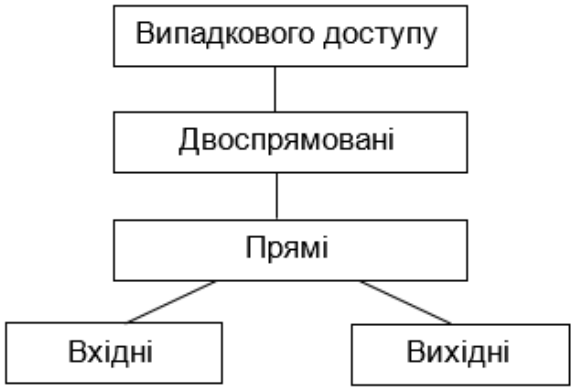


Рисунок 3.3 – Типи ітераторів

Вхідні та *вихідні* ітератори на практиці використовуються при зчитуванні (вхідні) та запису (вихідні) даних з файлів або потоку **cin** (вхідні) і в файли або в потік **cout** (вихідні). Вони дозволяють пересуватися лише на один крок по файлу/потоку.

Якщо алгоритму потрібно просуватися в одному напрямку (тільки вперед) і виконувати запис і зчитування, йому необхідно використовувати *прямий* ітератор.

Коли алгоритму необхідно пересуватись і вперед і назад по контейнеру, він використовує *двоспрямований* ітератор.

Зрештою, якщо алгоритму потрібен негайний доступ до довільного елемента контейнера без будь-яких покрокових просувань, використовується *ітератор довільного* (*випадкового*) *доступу*. Це виконується подібно до звертання до елементу масиву по його номеру.

В таблиці 3.9 показано, які операції підтримуються якими ітераторами.

Таблиця 3.9 – Можливості ітераторів різних типів

Тип ітератора	**«Крок вперед» ++**	**Читання value=*i**	**Запис *i=value**	**«Крок назад» --**	**Довільний доступ [n], операції +, -**
Довільного доступу	**x**	**x**	**x**	**x**	**x**
Двоспрямований	**x**	**x**	**x**	**x**	
Прямий	**x**	**x**	**x**		
Вхідний	**x**	**x**			
Вихідний	**x**		**x**		

Як можна бачити, всі ітератори підтримують операцію '++' для просування по контейнеру (вперед).

Вхідний ітератор можна використовувати з операцією '*' праворуч від знака операції привласнення (але не ліворуч!):

```
value = *iter;
```

Вихідні ітератори можна використовувати з операцією '*', але тільки ліворуч від знака операції привласнення:

```
*iter = value;
```

Прямий ітератор підтримує і запис і зчитування, а двоспрямований можливо як інкрементувати так і декрементувати.

Ітератор довільного (випадкового) доступу підтримує операцію "**[ ]**" і прості арифметичні операції '+' і '-' для швидкісного звертання до будь-якого елемента контейнера.

Алгоритм завжди може використовувати ітератори з більш широкими можливостями, ніж потрібно. Наприклад, якщо потрібний прямий ітератор,

цілком нормальною дією вважається використання двоспрямованого ітератора або ітератора з довільним доступом.

Відповідність алгоритмів контейнерам

Ітератори здійснюють зв'язок між алгоритмами і контейнерами. В таблиці 3.10 показано які контейнери підтримують які ітератори.

Таблиця 3.10 – Типи ітераторів, що підтримуються контейнерами

Тип ітератора	Вектор	Список	Deque	Мно-жина	Мульти-множина	Відобра-ження	Мульти-відображення
Довільного доступу	**x**		**x**				
Двоспрямований	**x**	**x**	**x**	**x**	**x**	**x**	**x**
Прямий	**x**	**x**	**x**	**x**	**x**	**x**	**x**
Вхідний	**x**	**x**	**x**	**x**	**x**	**x**	**x**
Вихідний	**x**	**x**	**x**	**x**	**x**	**x**	**x**

При оголошенні (визначенні) ітератора необхідно вказати, для контейнера якого типу його слід використовувати. Наприклад, якщо в програмі оголошено список значень типу **int**:

```
list<int>   iList;              //список int
```

то для того, щоб оголосити для нього ітератор, треба зазначити:

```
list<int>::iterator iter;       //ітератор для списку цілих чисел
```

Після цього **STL** автоматично робить ітератор двоспрямованим, оскільки саме такий тип потрібен контейнеру типу список. Ітератор для вектора чи черги із двостороннім доступом автоматично робиться ітератором довільного доступу.

Така автоматизація процесу досягається за рахунок того, що клас ітератора конкретного класу є спадкоємцем більш загального ітератора. Тому ітератори для векторів та черг із двостороннім доступом є похідними від класу **random_access_iterator,** а ітератори для списків – похідними від класу **bidirectional_iterator.**

Треба пам’ятати, що відповідні ітератори можуть використовуватись тільки з тими контейнерами, які їх підтримують (див. останню табл.). Так вектори

і черги з двостороннім доступом використовують тільки ітератори довільного доступу, а списки і асоціативні контейнери – тільки двоспрямовані ітератори.

Теперь треба розглянути, який алгоритм може бути застосованим до якого контейнера і як на це впливає ітератор. Зрозуміло, що кожний алгоритм, в залежності від його призначення, використовує конкретний ітератор. В наступній таблиці наведено приклади алгоритмів і необхідні їм ітератори (табл. 3.11).

Таблиця 3.11 – Типи ітераторів, які потрібні представленим алгоритмам

Алгоритм	Вхідний	Вихідний	Прямий	Двоспрямований	Довільного доступу
for_each	**x**				
find	**x**				
count	**x**				
copy	**x**	**x**			
replace			**x**		
unique			**x**		
reverse				**x**	
sort					**x**
nth_element					**x**
merge	**x**	**x**			
accumulate	**x**				

З наведених вище таблиць можна бачити з яким алгоритмом буде працювати той чи інший контейнер. Наприклад, алгоритму **sort()** необхідний ітератор довільного доступу. Але такі ітератори підтримуються тільки векторами і чергами з двостороннім доступом. Тому неможливо застосовувати **sort()** до списків, множин та відображень.

Будь-який алгоритм, який не вимагає наявності ітератора довільного доступу буде працювати з будь-яким типом контейнера, тому що всі контейнери використовують двоспрямовані ітератори.

Відносно невелика кількість алгоритмів реально вимагають наявності ітераторів довільного доступу. Тому більшість алгоритмів працюватиме з більшістю контейнерів. В цьому і полягає *ортогональність* **STL**.

Перекриття методів і алгоритмів

Деякі контейнери мають власні функції-методи, імена яких співпадають з іменами відповідних алгоритмів. Наприклад, алгоритму **find()** необхідний тільки вхідний ітератор, тому його можна використовувати з будь-яким контейнером. Але у множин і відображень є свій власний метод **find()** (чого немає у послідовних контейнерів). Яку версію **find()** слід використовувати? Загалом, якщо є метод, який дублює своїм ім'ям алгоритм, то це зроблено внаслідок того, що алгоритм в такому випадку виявляється неефективним. Тому, при наявності такого вибору, перевагу треба надавати методу.

І ще один приклад використання методу. У контейнера «список» немає ітератора довільного доступу, який потрібен алгоритму сортування. Проте є метод **sort()**, і саме його треба використовувати для сортування списків.

Робота з ітераторами

Використовувати ітератори значно простіше ніж розповідати про них. Деякі приклади їх використання було вже розглянуто, коли значення ітераторів поверталося методами **begin()** і **end()** контейнерів. При цьому припускається, що вони працюють, як покажчики. Теперь буде показано, як реально використовуються ітератори з цими та іншими функціями.

Доступ до даних. Перший приклад стосується використання ітераторів зі списками.

```
// iterator for loop and for output
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main()
  {
  int arr[] = { 2, 4, 6, 8 };
  list<int> theList;
  list<int>::iterator iter;              // ітератор для списку цілих чисел

  for(int k=0; k<4; k++)                 // заповнення списку з масива
    theList.push_back( arr[k] );

  for(iter = theList.begin(); iter != theList.end(); iter++)
    cout << *iter << ' ';                // вивід списка
```

```
  cout << endl;
  return 0;
  }
```

Програма виводить числа з контейнера **theList**:

```
2 4 6 8
```

В програмі оголошено ітератор типу **list<int>**, який відповідає типу контейнера. В циклі **for** здійснюється його ініціалізація від масиву. Другий цикл **for** використовує значення метода **theList.begin()**, який вказує на початок контейнера. Ітератор інкрементується операцією '++' для проходження по елементах, а операцією '*' надається доступ до значення, на який він вказує. Вихід з циклу виконується при досягненні кінця контейнера (**theList.end()**).

Заповнення списку парними числами можна було б зробити наступним чином.

```
  list<int> theList(5); int data = 0;
// заповнення списку парними числами
  for( list<int>::iterator  it = theList.begin(); it != theList.end(); it++)
     *it = data += 2;
```

Цикл заповнює контейнер цілими парними значеннями: **2 4 6 8 10**.

Наступний приклад демонструє використання ітераторів у якості параметрів алгоритмів. Це показано на прикладі алгоритму **find()**.

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

int main()
  {
  list<int> theList(5);                 // порожній список з 5 елементів
  list<int>::iterator iter;             // ітератор
  int data = 0;
// заповнення  списку:
  for(iter = theList.begin(); iter != theList.end(); iter++)
     *iter = data += 2;                 // 2, 4, 6, 8, 10
// пошук числа 8
  iter = find(theList.begin(), theList.end(), 8);
  if( iter != theList.end() )
     cout << "\nЗнайдено 8.\n";
  else
```

```
    cout << "\nDid not find 8.\n";
  return 0;
  }
```

Алгоритм **find()** має три параметри. Перші два – це ітератори, які визначають діапазон елементів для пошуку. Третій – значення, наявність якого перевіряється. Контейнер заповнюється числами **2**, **4**, **6**, **8**, **10**. Після цього алгоритм **find()** перевіряє наявність числа **8**. Алгоритм повертає значення **theList.end()**, якщо досягнуто кінця контейнера, а шуканий елемент не знайдено. Інакше, повертається значення покажчика (ітератора) на елемент списку зі значенням **8**, і це означає, що елемент знайдено. Тоді на екран виводиться: **Знайдено 8**.

При роботі з вектором можна також вирахувати, де саме в контейнері розташоване число **8**. Це дозволяє зробити *ітератор довільного доступу*. Наступний фрагмент дозволяє це зробити, якщо в розглянутому прикладі замінити *список* на *вектор* **v** з відповідним ітератором **iter**.

```
iter = find(v.begin(), v.end(), 8);
if( iter != v.end() )
cout << "\nЧисло 8 розташоване по зміщенню " << ( iter - v.begin() );
else  cout << "\nЧисло 8 незнайдено.\n";
Результатом буде: Число 8 розташоване по зміщенню 3
```

Спеціалізовані ітератори

Далі розглядаються два спеціалізовані типи ітераторів: *адаптери ітераторів*, які можуть змінювати поведінку звичайних ітераторів, і *потокові ітератори*, завдяки яким вхідні та вихідні потоки можуть поводитися як *контейнери*.

Адаптери ітераторів

У **STL** розрізняють три варіанти модифікації звичайного ітератора. Це *зворотний* ітератор, ітератор *вставки* та ітератор *неініціалізованого зберігання*. Зворотний ітератор дозволяє проходити контейнер у зворотному напрямку. Ітератор вставки створений для зміни поведінки різних алгоритмів, таких як **copy**() і **merge**(), щоб вони саме вставляли дані, а не перезаписували існуючі. Ітератор неініціалізованого зберігання дозволяє зберігати дані в ділянці пам'яті,

яка ще не ініціалізована. Він використовується у досить специфічних випадках, тому тут не розглядається.

Зворотні ітератори (reverse iterator)

Для виконання реверсного проходу контейнерів використовуються **зворотні *ітератори***. У наступному прикладі показано застосування цього ітератора для виведення вмісту списку у зворотному порядку.

```
// demonstrates reverse iterator
#include <iostream>
#include <list>
using namespace std;

int main()
  {
  int arr[] = { 2, 4, 6, 8, 10 };        // масив цілих
  list<int> theList;

  for(int j=0; j<5; j++)                 // запис масиву
    theList.push_back( arr[j] );         // до списку

  list<int>::reverse_iterator  revit; // reverse iterator

  revit = theList.rbegin();              // ітерація з кінця
  while( revit != theList.rend() )       // до початку
    cout << *revit++ << ' ';             // вивід
  cout << endl;
  return 0;
  }
```

Результат:

```
10 8 6 4 2
```

При застосуванні *зворотного* ітератора слід використовувати методи **rbegin()** і **rend()** (не треба використовувати їх зі звичайними ітераторами). Проходження по контейнеру починається з кінця, при цьому викликається метод **rbegin()**. Значення ітератора треба інкрементувати при пересуванні від елемента до елементу! Не треба його декрементувати, бо результат буде неочікуваним. При використанні **reverse_iterator** рух виконується від **rbegin()** до **rend()**, за рахунок інкрементування.

Ітератори вставки

Деякі алгоритми, наприклад, **copy**() часто використовуються для перезапису існуючих даних в контейнері призначення. Проте, іноді було б краще, якби алгоритм **copy**() не перезаписував, а вставляв дані в контейнер. Такої поведінки алгоритму можна досягти за допомогою ітератора **вставки**. У нього є три корисні по своєму версії:

– **back_inserter** – вставляє нові елементи в кінець (після останнього);

– **front_inserter** – вставляє нові елементи в голову (перед першим);

– **inserter** – вставляє нові елементи у вказане місце.

Наступна програма показує, як виконувати вставку в кінець контейнера **deque**.

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

int main()
  {
  int arr1[] = { 1, 3, 5, 7, 9 };         // для заповнення d1
  int arr2[] = {2, 4, 6};                 // для заповнення d2
  deque<int> d1;
  deque<int> d2;

  for(int i=0; i<5; i++)                  // запис масивів до черг
    d1.push_back( arr1[i] );
  for(int j=0; j<3; j++)
    d2.push_back( arr2[j] );
// копіювання d1 в кінець d2:
  copy( d1.begin(), d1.end(), back_inserter(d2) );

  cout << "\nd2: ";                       //вивід d2:
  for(unsigned int k=0; k<d2.size(); k++)
    cout << d2[k] << ' ';
  cout << endl;
  return 0;
  }
```

Елементи з контейнера **d1** вставляються в кінець **d2**, слідом за існуючими елементами. Контейнер **d1** залишається без змін. Програма виводить на екран вміст **d2**:

```
d2:   2 4 6 1 3 5 7 9
```

Для вставки елементів до голови (початку) треба було б вказати:

```
copy( d1.begin(), d1.end(), front_inserter(d2) );
```

тоді нові елементи були б перед існуючими даними **d2**. При цьому обертається порядок елементів **d1**. В такому випадку вміст **d2** був би таким:

```
9 7 5 3 1 2 4 6
```

Можна вставляти дані і в будь-яке інше місце контейнера при використанні версії ***inserter*** ітератора. Наприклад, так нові дані можна вставити на початок d2:

```
copy( d1.begin(), d1.end(), inserter(d2, d2.begin() );
```

Першим параметром ***inserter*** є ім'я контейнера, в який копіюються елементи, другим - ітератор, який вказує позицію, починаючи з якої вставляються дані. При цьому порядок слідування елементів не змінюється. У контейнері **d2** після вставки будуть наступні дані:

```
1 3 5 7 9 2 4 6
```

Змінюючи другий параметр ***inserter*** можна задавати будь-яке місце у контейнері.

Треба зазначити, що ***front_inserter*** не можна використовувати з векторами, тому що для них доступ можливий тільки до кінця контейнера.

Потокові ітератори

Потокові ітератори дозволяють інтерпретувати файли та пристрої вводу/виводу (потоки **cin** та **cout**), як контейнери. Отже, можна використовувати файли та пристрої введення/виводу як параметри алгоритмів. З їх допомогою можна застосовувати відповідні алгоритми безпосередньо до потоків вводу/виводу.

Існує два потокові ітератори: **ostream_iterator** і **istream_iterator**.

Клас ostream_iterator

Об'єкт цього класу можна використовувати в якості параметра будь-якого алгоритму, який може працювати з вихідним ітератором. В наступному прикладі він використовується, як параметр **copy()**.

```
#include <iostream>
#include <algorithm>
#include <list>
#include <iterator>
using namespace std;

int main()
  {
  int arr[] = { 10, 20, 30, 40, 50 };
  list<int> theList;

  for(int j=0; j<5; j++)                          //масив у список
    theList.push_back( arr[j] );

  ostream_iterator<int> ositer(cout, ", ");       // вихідний ітератор

  cout << "\nВміст списку: ";
  copy(theList.begin(), theList.end(), ositer);   //відображення списку
  cout << endl;
  return 0;     }
```

Ітератор **ositer** в прикладі визначено для копіювання значень типу **int**. Параметрами конструктора виступають потік, в який відбудеться запис значень, і рядок, який виводитися після кожного з них. Потоком зазвичай є ім'я файлового об'єкта або **cout**; в даному прикладі це **cout**. При записуванні в потік може також передаватися рядок-роздільник, що складається з будь-який символів. Тут це кома і пробіл. Алгоритм **copy()** копіює вміст списку в потік **cout**. Ітератор вихідного потоку використовується в якості третього аргументу.

На екрані в результаті роботи програми можна буде побачити:

```
Вміст списку:   10, 20, 30, 40, 50,
```

В наступному прикладі показано, як можна записати вміст контейнера в файл за допомогою вихідного ітератора.

```
#include <fstream>
#include <algorithm>
#include <list>
```

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

int main()
  {
  int arr[ ] = { 11, 21, 31, 41, 51 };
  list<int>  theList;

  for(int j=0; j<5; j++)                          // запис масиву
    theList.push_back( arr[j] );                  // до списку
  ofstream outfile("ITER.DAT");                   // створення файлового об'єкта

  ostream_iterator<int> ositer(outfile, " ");     // вихідний ітератор
 // копіювання списку у файл:
  copy(theList.begin(), theList.end(), ositer);
  return 0;
  }
```

В даному прикладі визначено файловий об'єкт класу **ofstream** і асоційовано його з конкретним файлом (в програмі це файл **ITER.DAT**). При записуванні в файл бажано використовувати зручні роздільники. Тут між елементами вставляється просто символ пробіл. Вивід на екран в цій програмі не відбувається, проте за допомогою текстового редактора можна переконатися, що в файл **ITER.DAT** записано: **11 21 31 41 51**

Клас istream_iterator

Об'єкт цього класу може використовуватися як параметр будь-якого алгоритму, що працює з вхідним ітератором. В наступному прикладі числа, які вводяться з клавіатури (потік **cin**) записуються в контейнер типу *список*.

```
#include <iostream>
#include <list>
#include <algorithm>
#include <iterator>
using namespace std;

int main()
  {
  list<float> fList(5);                           // порожній список довжиною 5

  cout << "\n Введіть 5 чисел (типу float): ";
// istream ітератори:
  istream_iterator<float>  cin_iter(cin);         // cin
  istream_iterator<float>  end_of_stream;         // кінець потоку
  copy( cin_iter, end_of_stream, fList.begin() ); // копіювання з cin у fList
```

```
  cout << endl;                                          // вивід fList
  ostream_iterator<float> ositer(cout, "--");
  copy(fList.begin(), fList.end(), ositer);
  cout << endl;
  return 0;     }
```

Діалог програми з користувачем може бути таким:

```
Введіть 5 чисел (типу float): 1.1 2.2 3.3 4.4 5.5
1.1--2.2--3.3--4.4--5.5--
```

Програма з'єднує вхідний ітератор **cin_iter** з потоком **cin** за допомогою конструктора з одним параметром. Для **copy()** необхідно вказувати діапазон значень. Початок діапазону – вхідний ітератор **cin_iter**. Кінець діапазону – вхідний ітератор **end_of_stream**, який завжди необхідно створювати за допомогою конструктора без параметрів для зазначення кінця потоку. З свого боку, користувач повинен повідомити про закінчення вводу даних натисканням комбінації клавіш **Ctrl+Z**, в результаті чого в потік передається стандартний символ кінця файлу.

Будь-які операції виводу на екран необхідно виконати до визначення вхідного ітератора для **cin**, тому що з моменту його визначення вивід на екран стає недоступним: програма очікує вводу даних.

В наступному прикладі замість потоку **cin** використовується вхідний файл, з якого зчитуються дані для програми. Приклад також демонструє роботу з алгоритмом **copy()**.

```
#include <iostream>
#include <list>
#include <fstream>
#include <algorithm>
#include <iterator>
using namespace std;

int main()
  {
  list<int> iList;                          // пустий список
  ifstream infile("ITER.DAT");              // створення файлового об'єкту
                                            // (файл ITER.DAT повинен існувати)
  // вхідні ітератори:
  istream_iterator<int> file_iter(infile);  // файл
  istream_iterator<int> end_of_stream;  // кінець файлу
  // копіювання з файлу infile до списку iList
```

```
  copy( file_iter, end_of_stream, back_inserter(iList) );

  cout << endl;                                  // display iList
  ostream_iterator<int> ositer(cout, "- -");
  copy(iList.begin(), iList.end(), ositer);
  cout << endl;
  return 0;      }
```

Файл **ITER.DAT** вважається створеним при виконанні програми, яка демонструє роботу з вихідним ітератором. Результат роботи програми:

```
11- -21- -3l- -31- -41- -51- -
```

Об'єкт класу **ifstream** забезпечує програмі зв'язок з файлом **ITER.DAT**. Це об’єкт **infile**. Вхідний ітератор **file_iter** створено конструктором з параметром **infile**. Для позначення кінця файлового потоку використовується вхідний ітератор **end_of_stream**, створений за допомогою конструктора без параметрів.

Для вставки даних в список використовується ***зворотний*** ітератор **back_inserter()**. Список бажано створити спочатку порожнім, не вказуючи його розмір. Так має сенс робити при зчитуванні вхідних даних з файлу, оскільки заздалегідь невідомо, скільки елементів буде введено.

3.2.7. Асоціативні контейнери STL

Послідовні контейнери (вектори, списки та черги з двостороннім доступом) зберігають дані у фіксованій лінійній послідовності. Пошук конкретного елемента в такому контейнері (якщо невідомий чи недоступний його індекс або він не знаходиться в одному з кінців контейнера) – це тривала процедура, що включає покроковий перехід від позиції до позиції.

В *асоціативних контейнерах* елементи не розташовуються у послідовності. Вони є складнішими структурами, що дає великий виграш у швидкості пошуку. Зазвичай асоціативні контейнери мають деревоподібну структуру, хоча можливі й інші варіанти, наприклад таблиці. У будь-якому випадку, головною перевагою цієї категорії контейнерів є швидкість пошуку.

Пошук в асоціативних контейнерах проводиться за допомогою ключів. Ключі зазвичай представляються у вигляді чисельного або рядкового значення, яке може бути як атрибутом об'єкта в контейнері, так і самим об'єктом.

В **STL** існує дві основні категорії асоціативних контейнерів: *множини* і *відображення*.

Множина зберігає об'єкти, які одночасно є ключами. ***Відображення*** можна охарактеризувати як свого роду таблицю з двох стовпців, у першому з яких зберігаються об'єкти, що містять ключі (ключові об'єкти), а в другому об'єкти, що містять значення.

У множинах, як і у відображеннях, може зберігатися лише один екземпляр кожного ключа. А кожному ключу, в свою чергу, відповідає унікальне значення. Такий підхід використовується в словниках, коли кожному слову ставиться у відповідність лише одна стаття. Але в **STL** є спосіб обійти це обмеження. ***Мультимножини*** і ***мультивідображення*** організовано аналогічно, але одному ключу може відповідати кілька значень.

Асоціативні контейнери мають кілька спільних із іншими контейнерами методів. Тим не менш, деякі алгоритми, такі, як **lower_bound()** та **equal_range()**, характерні лише для них. Крім того, деякі методи можуть бути використані для будь-яких контейнерів, крім асоціативних. Серед них сімейство методів проштовхування та виштовхування (**push_back()** тощо). Дійсно, немає особливого сенсу в їх застосуванні до цієї категорії контейнерів, оскільки в них елементи повинні додаватися і вилучатися у порядку заданому ключами, але не в кінець або початок контейнера.

Множини та мультимножини

Множини часто використовуються для зберігання об'єктів класів користувача. Нижче у цьому розділі наведено приклади застосування множин до об'єктів класу **employee**. Але у множинах можуть зберігатися і більш тривіальні типи, таки як числа, рядки тощо. Об'єкти впорядковані і весь об'єкт є ключем.

У першому прикладі продемонстровано множину, що містить об'єкти класу **string** (клас-шаблон для представлення рядків символів).

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

int main()
  { // масив об’єктів типу string:
  string names[ ] = {"Juanita", "Robert",
```

```
                "Mary", "Amanda", "Marie"};
  // ініціалізація множини з масиву:
set<string, less<string> >   nameSet(names, names+5);
  // ітератор для множини:
set<string, less<string> >::iterator  iter;

nameSet.insert("Yvette");         // додаткова вставка імен
nameSet.insert("Larry");
nameSet.insert("Robert");         // не виконується бо вже є в множині
nameSet.insert("Barry");
nameSet.erase("Mary");            // видалення імені
// вивід на екран множини:
cout << "\nРозмір=" << nameSet.size() << endl;
iter = nameSet.begin();
while( iter != nameSet.end() )
   cout << *iter++ << '\n';

string searchName;                // рядок для пошуку
cout << "\n Введіть ім'я для пошуку: ";
cin >> searchName;
// пошук вказаного імені:
iter = nameSet.find(searchName);
if( iter == nameSet.end() )
   cout << " Ім'я " << searchName << " ВІДСУТНЄ у множині.";
else
   cout << " Ім'я " << *iter << " є у множині.";
cout << endl;
return 0;
}
```

Для оголошення множини необхідно вказати тип об'єктів, що зберігаються в ньому. У прикладі це об'єкти класу **string**. До того ж можна вказати ***функціональний об'єкт***, який потрібен для впорядкування елементів множини. Тут використовується **less<string>** для впорядкування по зростанню значень рядкових об'єктів. Для впорядкування по зменшенню треба використовувати **greater<string>**. При відсутності ***функціонального об'єкта*** впорядкування залежатиме від типу об'єктів у множині.

Додавання ще одного імені **nameSet.insert("Robert")** немає сенсу для множини, але цілком можливе для мультимножини.

Як можна бачити, множина має інтерфейс, багато в чому схожий з іншими контейнерами **STL**. Це дає можливість ініціалізувати множину масивом, вставляти дані методом **insert()**, виводити їх за допомогою ітераторів.

Для знаходження елементів використовується метод **find()**. (Послідовні контейнери мають однойменний алгоритм). Далі показано приклад взаємодії із даною програмою, якщо користувач вводить для пошуку ім'я **"George"**:

```
Розмір=7
Amanda
Barry
Juanita
Larry
Marie
Robert
Yvette
Введіть ім’я для пошуку: George
Ім’я George ВІДСУТНЄ  у множині.
```

Звичайно, швидкість пошуку неможливо помітити на прикладі таких крихітних множин, проте, при досить великій кількості елементів асоціативні контейнери значно виграють у пошуку порівняно з послідовними.

В наступному прикладі розглядається ще одна досить важлива пара методів, які можна використовувати лише з асоціативними контейнерами. Це методи **lower_bound()** та **upper_bound()**.

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

int main()
  {  // множина об’єктів типу string:
  set<string, less<string> > organic;
  // ітератор для множини:
  set<string, less<string> >::iterator iter;

  organic.insert("Curine");          // ввести органічні сполуки
  organic.insert("Xanthine");
  organic.insert("Curarine");
  organic.insert("Melamine");
  organic.insert("Cyanimide");
  organic.insert("Phenol");
  organic.insert("Aphrodine");
  organic.insert("Imidazole");
  organic.insert("Cinchonine");
  organic.insert("Palmitamide");
  organic.insert("Cyanimide");
```

```
  iter = organic.begin();              // display set
  while( iter != organic.end() )
    cout << *iter++ << '\n';

  string lower, upper;                 // рядки для діапазону
  cout << "\n Введіть діапазон (наприклад  C Czz): ";
  cin >> lower >> upper;
  iter = organic.lower_bound(lower);
  while( iter != organic.upper_bound(upper) )
    cout << *iter++ << '\n';
  return 0;
  }
```

Програма спочатку виводить повний перелік елементів множини класу **organic** (органічні сполуки) Потім користувача просять ввести пару ключових значень, що задають діапазон всередині множини. Елементи, що входять до цього діапазону, виводяться на екран. Приклад роботи програми для діапазону **Aaa Curb**:

```
Aphrcciine
Cinchonine
Curarine
Curine
Cyanimide
Imidazole
Melaninc
Palmitamide
Phenol
Xanthinc

Введіть діапазон (наприклад  C Czz): Aaa Curb
Aphrodine
Cinchonine
Curarine
```

Метод **lower_bound()** в якості аргументу використовує значення того ж типу, що і ключ. Він повертає ітератор, що вказує на перший запис множини, значення якої не менше аргументу (що означає «не менше», у кожному конкретному випадку визначається конкретним функціональним об'єктом, що використовується при визначенні множини). Функція **upper_bound()** повертає ітератор, що вказує на перший елемент, значення якого більше ніж аргумент. Обидві ці функції дозволяють задавати діапазон значень у контейнері.

Програми роботи с **мультимножинами** працюють аналогічно. Зміни стосуються оголошень. Оголошення мультимножини цілих чисел може мати вигляд:

```
multiset <int>  ms1;                    // Оголошення мультимножини
multiset <int>::iterator  ms1_Iter;     // Оголошення ітератора мультимножини
```

Відображення та мультивідображення

У *відображенні* зберігаються пари значень, одне з яких є *ключовим об'єктом*, а інше – *об'єктом*, *що містить значення*. Ключовий об'єкт містить *ключ*, за яким шукається значення. Об’єкт-значення містить деякі дані, які зазвичай цікавлять користувача, який шукає щось в контейнері за допомогою ключа. У ключовому об'єкті можуть зберігатися рядки, числа або складніші об'єкти. Значеннями часто є рядки, числа або інші об'єкти, навіть інші контейнери.

Наприклад, ключем може бути слово, а значенням – число, яке говорить про те, скільки разів це слово зустрічається в тексті. Таке відображення задає частотну таблицю. Або, наприклад, ключем може бути слово, а значенням – список номерів сторінок. Таке рішення може бути використане для створення предметного покажчику.

Одним із найбільш поширених прикладів використання відображень є *асоціативні масиви*. У звичайних масивах індекс завжди є цілим числом. За його допомогою, як відомо, здійснюється доступ до конкретних елементів. Так, у виразі **anArray[3]** число 3 є індексом.

Дещо по-іншому виконується доступ до елементів в асоціативних масивах. Тип індексу асоціативного масиву може бути не тільки числовим. При цьому з'являється можливість написати, наприклад, такий вислів: **anArray ["jane"]**.

Асоціативний масив

Розгляд відображення в якості асоціативного масиву наведено в наступному прикладі програми. Ключами будуть назви штатів США, а значеннями – кількість їх населення.

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

```
int main()
  {
  string name;
  int pop;

  string states[ ] = { "Wyoming", "Colorado", "Nevada",
                  "Montana", "Arizona", "Idaho"};
  int pops[ ] = { 470, 2890, 800, 787, 2718, 944 };

  map<string, int, less<string> > mapStates     // map
  map<string, int, less<string> >::iterator iter;   // iterator

  for(int j=0; j<6; j++)
    {
    name = states[j];                           // назва штату з масиву
    pop = pops[j];
    mapStates[name] = pop;                      // запис у відображення
    }
  cout << " Введіть назву штату: ";             // назва від користувача
  cin >> name;
  pop = mapStates[name];                        // пошук населення по назві штату
  cout << "Населення: " << pop << ",000\n";

  cout << endl;                                 // вивід всього відображення
  for(iter = mapStates.begin(); iter != mapStates.end(); iter++)
    cout << (*iter).first << ' ' << (*iter).second << " 000\n";
  return 0;
  }
```

При запуску програми у користувача запитується назва штату. Отримане значення використовується як ключ для пошуку у відображенні значення кількості населення та виведення його на екран. Після цього виводиться весь вміст відображення: усі пари: штат – населення. Приклад роботи програми:

```
Введіть назву штату: Colorado
Населення: 2890,000
Arizona 2718 000
Colorado 2890 000
Idaho 944 000
Montana 787 000
Nevada 800 000
Wyoming 470 000
```

При використанні множин і відображень пошук виконується дуже швидко. Програма швидко знаходить значення кількості населення за введеною назвою

штату, хоча реально наочно помітити ефективність цього підходу буде можливо при зберіганні великої кількості пар. Проте ітерація в даному типі контейнерів не така швидка, як у послідовних контейнерах. Ще треба звернути увагу на те, що імена штатів розташовані за абеткою, хоча були задані в масиві хаотичним чином.

Оголошення відображення використовує три шаблонних параметри:

```
map<string, int, less<string> > mapStates;
```

Перший з них задає тип ключа; в даному випадку це назва штату типу **string**. Наступний параметр визначає тип значень, що зберігаються в контейнері (**int** в прикладі – це кількість населення в тисячах). Третій аргумент задає порядок сортування ключів. Для впорядкування по алфавіту визначено **less<string>**. Також для цього відображення задано ітератор.

Вхідні дані програми зберігаються в двох масивах. Занесення їх в контейнер виконується оператором:

```
mapStates[name] = pop;
```

Цей компактний вираз нагадує вставку у звичайний масив, проте індексом такого масиву є рядок **name**, а не деяке ціле число.

Після введення користувачем назви штата програма виконує пошук відповідного значення кількості населення:

```
pop = mapStates[name];
```

Замість звертання до елементів по індексу є можливість отримати доступ одночасно до обох частин елементів відображення – до ключів і значень. Це можливо робити за допомогою ітераторів. Ключ отримується по **(*iter).first**, а значення – по **(*iter).second**. В інших варіантах звертання до відображення ітератор працює як в інших контейнерах.

Зберігання об'єктів користувача

В усіх розглянутих прикладах в контейнерах зберігались об’єкти базових типів. Між тим, одна з переваг **STL** полягає саме в тому, що об'єкти користувальницьких типів також є повноправними даними, які можна зберігати і над якими можна виконувати ті самі операції.

В якості приклада розглядається клас **person**, в якому містяться прізвища, імена і номери телефонів людей. Програма створює об'єкти цього класу і вставляє їх в множину, заповнюючи у такий спосіб віртуальну телефонну книгу. Користувач, працюючий з програмою, вводить прізвище та ім'я людини. На екран виводиться результат пошуку даних, що відповідають вказаній людині. Для рішення цієї задачі застосовано *мультимножину*, щоб два і більша кількість об'єктів **person** могли мати однакові імена та прізвища.

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

class person
  {
  private:
    string lastName;
    string firstName;
    long phoneNumber;
  public:  // конструктор без параметрів:
    person() : lastName("blank"),
               firstName("blank"), phoneNumber(0)
      { }
        // конструктор з трьома параметрами
    person(string lana, string fina, long pho) :
          lastName(lana), firstName(fina), phoneNumber(pho)
      { }
    friend bool operator<(const person&, const person&);
    friend bool operator==(const person&, const person&);

    void display() const   // вивід даних людини
      {
      cout << endl << lastName << ",\t" << firstName
           << "\t\tPhone: " << phoneNumber;
      }
  };
  // перевантаження операції < для класу person:
bool operator<(const person& p1, const person& p2)
  {
  if(p1.lastName == p2.lastName)
    return (p1.firstName < p2.firstName) ? true : false;
  return (p1.lastName < p2.lastName) ? true : false;
  }
  // перевантаження операції == для класу person:
bool operator==(const person& p1, const person& p2)
  {
```

```
  return (p1.lastName == p2.lastName &&
        p1.firstName == p2.firstName ) ? true : false;
  }
//////////////////////////////////////////////////////////////
int main()
  {   // створення об'єктів класу person
  person pers1("Deauville", "William", 8435150);
  person pers2("McDonald", "Stacey", 3327563);
  person pers3("Bartoski", "Peter", 6946473);
  person pers4("KuangThu", "Bruce", 4157300);
  person pers5("Wellington", "John", 9207404);
  person pers6("McDonald", "Amanda", 8435150);
  person pers7("Fredericks", "Roger", 7049982);
  person pers8("McDonald", "Stacey", 7764987);
    // мультимножина об'єктів persons
  multiset< person, less<person> > persSet;
    // ітератор мультимножини об'єктів person:
  multiset<person, less<person> >::iterator iter;
    // запис об'єктів в мультимножину:
  persSet.insert(pers1);
  persSet.insert(pers2);
  persSet.insert(pers3);
  persSet.insert(pers4);
  persSet.insert(pers5);
  persSet.insert(pers6);
  persSet.insert(pers7);
  persSet.insert(pers8);

  cout << "\nNumber of entries = " << persSet.size();
  // вивід мультимножини
  iter = persSet.begin();
  while( iter != persSet.end() )
    (*iter++).display();
  // ввести прізвище та ім'я:
  string searchLastName, searchFirstName;
  cout << "\n\nEnter last name of person to search for: ";
  cin >> searchLastName;
  cout << "Enter first name: ";
  cin >> searchFirstName;
  // створення об'єкта person
  person searchPerson(searchLastName, searchFirstName, 0);

  // вивід кількості однакових об'єктів:
  int cntPersons = persSet.count(searchPerson);
  cout << "Number of persons with this name = " << cntPersons;

  // вивід всіх знайдених
  iter = persSet.lower_bound(searchPerson);
  while( iter != persSet.upper_bound(searchPerson) )
```

```
    (*iter++).display();
  cout << endl;
  return 0;
  }  // end main()
```

Результат роботи програми:

```
Number of entries = 8
Bartoski,       Peter          Phone: 6946473
Deauville,      William        Phone: 8435150
Fredericks,     Roger          Phone: 7049982
KuangThu,       Bruce          Phone: 4157300
McDonald,       Amanda         Phone: 8435150
McDonald,       Stacey         Phone: 3327563
McDonald,       Stacey         Phone: 7764987
Wellington,     John           Phone: 9207404

Enter last name of person to search for: Bartoski
Enter first name: Peter
Number of persons with this name = 1
Bartoski,       Peter          Phone: 6946473
```

Контрольні запитання та завдання

1. Дати визначення і пояснити сутність понять «узагальнене програмування» та «параметричний поліморфізм».
2. Пояснити, які переваги надає використання шаблонів функцій порівняно зі звичайними функціями C++. Як оголошуються і викликаються шаблонні функції?
3. Пояснити призначення і формат оголошення шаблонів класів.
4. Дати характеристику різним способам визначення функцій-методів шаблонних класів: всередині шаблону і поза його межами.
5. Перелічити і охарактеризувати основні складові стандартної бібліотеки шаблонів **STL**. Що означає ортогональність бібліотеки?
6. Пояснити переваги і недоліки послідовних контейнерів відносно один до одного та до звичних масивів C++.
7. Навести приклади алгоритмів **STL** та особливостей їх застосування для обробки даних у контейнерах та звичайних масивах C++.
8. Пояснити, що собою являє ітератор **STL** та чим він відрізняється від звичайного покажчика C++.
9. Дати характеристику різним типам ітераторів **STL**.
10. Для заданого контейнеру, визначити, які алгоритми можуть бути застосовані для обробки його вмісту.
11. Перелічити і охарактеризувати асоціативні контейнери **STL**. Які їх переваги та недоліки, порівняно з послідовними контейнерами?
12. Пояснити сутність та особливості роботи з множинами **STL**.
13. Поясніть сутність та особливості роботи з відображеннями **STL**.
14. Розробити програму сортування елементів масиву, вектору і списку.
15. Розробити програму-довідник успішності навчання студентів ВНЗ. Ключом доступу повинно бути прізвище студента, значенням, що зберігається – середній бал успішності (рейтингова оцінка). Вивести на екран рейтинговий список ВНЗ.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Balagurusamy E. Object Oriented Programming with C++. Gautam Buddha Nagar, Uttar Pradesh: Mc Graw Hill India, 2019. 537 p.
2. Bancila M. Modern C++ Programming Cookbook: Master C++ core language and standard library features. Birmingham, UK: Packt Publishing, 2020. 750 p.
3. Briggs W. C++20 for Lazy Programmers: Quick, Easy, and Fun C++ for Beginners. New York, NY: Apress, 2001. 1012 p.
4. Deitel P., Deitel H. C++20 for Programmers. London, UK: Pearson Education Limited, 2021.1008 p.
5. Kirk D. R. Deciphering Object-Oriented Programming with C++: A practical, in-depth guide to implementing object-oriented design principles to create robust code. Birmingham, UK: Packt Publishing, 2022. 594 p.
6. Kusswurm D. Modern Parallel Programming with C++ and Assembly Language. New York, NY: Apress, 2022. 656 p.
7. Meyers S. Effective C++: 55 Specific Ways to Improve Your Programs and Designs. Boston, MA: Addison-Wesley Professional, 2008. 320 p.
8. Meyers S. Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14. Sebastopol, CA: O'reilly Media, 2014. 334 p.
9. Nibedit D. Cross-Platform Development with Qt 6 and Modern C++. Birmingham, UK: Packt Publishing, 2021. 442 p.
10. Object-Oriented Analysis and Design with Applications / G. Booch et. al. Boston, MA: Addison-Wesley Professional, 2007. 692 p.
11. Roth St. Clean C++20: Sustainable Software Development Patterns and Best Practices. New York, NY: Apress, 2021. 508 p.
12. Stroustrup B. A Tour of C++. Boston, MA: Addison-Wesley Professional, 2018. 256 p.
13. Stroustrup B. Programming: Principles and Practice Using C++. Boston, MA: Addison-Wesley Professional, 2014. 1312 p.
14. Tsetsekas H. Object-Oriented Programming Exercises with C++. Independently published, 2023. 112 p.
15. Булгакова О.С., Зосімов В.В., Ходякова Г.В. Алгоритмізація і програмування: теорія та практика: навч. посіб. для дист. навч. Миколаїв: СПД Румянцева, 2021. 138 с.

16. Войтенко В.В., Морозов А.В. С / С++: Теорія та практика: навч.-мет. посіб. Житомир: ЖДТУ, 2004. 324 с.
17. Гаркуша І.М. Конспект лекцій з дисципліни «Програмування». Частина 1. Для студентів галузі знань 12 «Інформаційні технології» спеціальності 126 «Інформаційні системи та технології». Дніпро: НТУ «ДП», 2020. 103 с.
18. Грицюк Ю.І., Рак Т.Є. Програмування мовою С++: навч. посіб. Львів: Вид-во Львівського ДУ БЖД, 2011. 292 с.
19. Ковалюк Т.В. Алгоритмізація та програмування: Підручник. Львів: «Магнолія 2006», 2013. 400 с.
20. Куліков В.М., Іващенко О.В., Успенський О.А. Конспект лекцій з навчальної дисципліни «Об'єктно-орієнтоване програмування» Частина І. Київ: Вид-во ІСЗЗІ НТУУ "КПІ", 2010. 280 с.
21. Трофименко О.Г., Прокоп Ю.В., Швайко І.Г., Буката Л.М. С++. Основи програмування. Теорія та практика: підручник / за ред. О.Г. Трофименко. Одеса: Фенікс, 2010. 544 с.

ГЛОСАРІЙ

n-арна асоціація (n-ary association) – асоціація між трьома і більшим числом класів.

UML (Unified Modeling Language – уніфікована мова моделювання) – мова графічного опису для об'єктного моделювання у галузі розробки програмного забезпечення, для моделювання бізнес-процесів, системного проектування та відображення організаційних структур.

Абстрактний клас (abstract class) – клас, який не має екземплярів або об’єктів.

Агрегація (aggregation) – спеціальна форма асоціації, яка служить для представлення відношення типу «частина-ціле» між агрегатом (ціле) і його складовою частиною.

Актор (actor) – будь-який об’єкт, суб’єкт або система, що взаємодіє з системою ззовні. Це може бути людина, технічний пристрій, програма або будь-яка інша система, яка є джерелом впливу на систему.

Активний об'єкт (active object) – об’єкт, що має власний процес управління і може ініціювати діяльність з управління іншими об'єктами.

Асоціація (association) – семантичне відношення між двома і більше класами, які специфікують характер взаємозв'язків між відповідними екземплярами даних класів

Атрибут (attribute) – змістовна характеристика класу. Вона описує множину значень, які можуть приймати окремі об’єкти цього класу.

Видимість (visibility) – якісна характеристика опису елементів класу. Вона характеризує потенційну можливість інших об’єктів моделі впливати на окремі аспекти поведінки даного класу.

Відношення (relationship) – семантичний зв'язок між окремими елементами моделі.

Відношення узагальнення – є звичайним таксономічним відношенням або відношенням класифікації між більш загальним елементом (батьком або пращуром (предком)) й більш окремим (частковим) або спеціальним елементом (дочірнім або нащадком). Див. *Узагальнення*.

Включення (include) – в UML це різновид відношення залежності між базовим варіантом використання і його спеціальним випадком.

Граничний клас (boundary class) – клас, який розташовується на межі системи з зовнішнім середовищем і безпосередньо взаємодіє з акторами, але є складовою частиною системи.

Деструктор – спеціальний метод, що є одним з членів класу та використовується для припинення існування об'єктів. Найбільш корисне застосування деструкторів при роботі з об’єктами, для яких пам’ять розподіляється динамічно.

Діаграма варіантів використання (use case diagram) – вихідне концептуальне представлення або концептуальна модель системи в процесі її проектування розробки.

Діаграми діяльності – частковий випадок діаграм станів. Вони дозволяють реалізувати в мові UML особливості процедурного і синхронного управління, обумовленого завершенням внутрішніх дій і діяльності.

Діаграма класів (class diagram) – діаграма мови UML, на якій представлено сукупність статичних декларативних елементів моделі, таких як класи з атрибутами і операціями, а також відношення між класами.

Діаграма послідовності (sequence diagram) – діаграма, на якій показана взаємодія об'єктів, впорядкована за часом.

Доріжка (swimlane) – графічна область діаграми діяльності, що містить елементи моделі, відповідальність за виконання яких належить окремим підсистемам.

Зв'язок (link) – будь-яке семантичне відношення між деякою сукупністю об'єктів.

Інтерфейс (interface) – іменована множина операцій, які характеризують поведінку окремого елемента моделі. В контексті мови UML – це спеціальний випадок класу, у якого є операції, але відсутні атрибути.

Кінець асоціації (association end) – кінцева точка асоціації, яка зв’язує асоціацію з класифікатором.

Кінцевий стан (final state) – різновид стану, що позначає припинення процесу зміни станів системи, що моделюється.

Клас (class) – абстрактний опис множини однорідних об’єктів, які мають однакові атрибути, операції і відношення з об’єктами інших класів.

Клас-асоціація (association class) модельний елемент, який одночасно є асоціацією і класом. Може розглядатися як асоціація, яка має

		властивості класу, або як клас, що має також властивості асоціації.
Клас-сутність (entity class)	–	пасивний клас, інформація про який повинна зберігатися постійно і не знищуватися зі знищенням об'єктів даного класу або виключенням системи.
Композит (composite)	–	клас, який пов'язаний відношенням *композиції* з одним або більшою кількістю класів.
Композиція (composition)	–	різновид відношення агрегації, при якій складові частини цілого мають такий же час життя, як і ціле.
Конкретний клас (concrete class)	–	клас, на основі якого можуть бути безпосередньо створені об’єкти (екземпляри класу).
Конструктор	–	спеціальна функція-метод класу, яку C++ автоматично викликає щоразу при створенні об'єкта типу даного класу.
Контейнери	–	об'єкти, що містять інші об'єкти.
Кооперація (collaboration)	–	специфікація множини об'єктів окремих класів, які спільно взаємодіють з метою реалізації окремих варіантів використання.
Кратність (multiplicity)	–	специфікація потужності множини значень, які може мати відповідний атрибут.
Лінія життя об'єкту (object lifeline)	–	це вертикальна лінія на діаграмі послідовності, яка представляє певний період часу, протягом якого існує об'єкт.
Маніпулятори	–	об'єкти особливих типів, які керують тим, як **ostream** або **istream** обробляють подальші аргументи.
Множинне успадкування	–	успадкування при якому похідний клас породжується з декількох базових класів.
Мультиоб'єкт (multiobject)	–	множина анонімних об’єктів, які можна утворити з одного класу.
Об'єкт (object)	–	сутність з визначеними межами і індивідуальністю, яка інкапсулює стан і поведінку.
Об'єкт-композит (composite object)	–	об’єкт, призначений для представлення об'єкта, що має власну структуру і внутрішні потоки (нитки) управління.
Об’єктно-орієнтований аналіз (Object-Oriented Analysis, OOA)	–	методологія, при якій вимоги до системи сприймаються з погляду класів та об'єктів, виявлених у предметній області.

Об'єктно-орієнтована декомпозиція – декомпозиція складної системи, що заснована на об'єктах, а не на алгоритмах. Є основою об'єктно-орієнтованого проектування.

Об'єктно-орієнтоване програмування (Object-Oriented Programming, OOP) – методологія програмування, заснована на поданні програми у вигляді сукупності об'єктів, кожний з яких є екземпляром певного класу, а класи утворюють ієрархію успадкування

Об'єктно-орієнтоване проектування (Object-Oriented Design, OOD) – методологія проектування, що поєднує в собі процес об'єктної декомпозиції та прийоми подання логічної й фізичної, а також статичної й динамічної моделей проектованої системи.

Обмеження (constraint) – деяка логічна умова, яка обмежує семантику обраного елемента моделі.

Операція (operation) – це сервіс, який надається кожним екземпляром або об'єктом класу на вимогу своїх клієнтів, у якості яких можуть виступати інші об'єкти, в тому числі і екземпляри цього ж класу.

Перетворення – це явна або неявна зміна значення залежно від типу. Перетворення забезпечує форму *поліморфізму*.

Перехід (transition) – відношення між двома станами, яке вказує на те, що об'єкт в першому стані повинен виконати певні дії і перейти в другий стан.

Повідомлення (message) – специфікація передачі інформації від одного елемента моделі до іншого з очікуванням виконання певних дій з боку приймаючого елемента.

Поліморфізм – це засіб для надання одному повідомленню різних значень залежно від типу оброблюваних даних.

Помічені значення (tagged value) – явне визначення властивості у вигляді «ім'я = значення». У поміченому значенні ім'я називають тегом (tag).

Похідний атрибут (derived element) – атрибут класу, значення якого для окремих об'єктів можна обчислити за допомогою значень інших атрибутів цього ж об'єкта.

Похідний клас – модифікація базового класу, яка успадковує загальні та захищені елементи базового класу.

Початковий стан (start state) – різновид стану, що позначає початок виконання процесу зміни станів модельованої системи.

Прямий ациклічний граф (ПАГ) – орієнтований граф успадкування без петель.

Розширення (extend) – тип відношення, що визначає взаємозв'язок базового варіанту використання з іншим варіантом використання, функціональна поведінка якого задіюється базовим не завжди, а тільки при виконанні додаткових умов.

Роль (role) – специфічне поводження деякої сутності, яке розглядається у певному контексті і має власне ім'я. Роль може бути статичною або динамічною.

Складений об'єкт (composite object) – Див. *об'єкт-композит*.

Стан дії (action state) – спеціальний випадок стану з деякою вхідною дією і, щонайменше, з одним переходом, що виходить зі стану.

Стан діяльності (activity state) – стан у графі діяльності, який слугує для подання процедурної послідовності дій, що вимагають певного часу.

Стан піддіяльності (subactivity state) – стан на діаграмі діяльності, який слугує для представлення неатомарної послідовності кроків процесу.

Стереотип (stereotype) – елемент моделі в нотаціях UML, який розширює семантику моделі. Стереотипи повинні базуватися на існуючих в UML типах або класах.

Сторожова умова (guard condition) – записана в прямих дужках логічна умова, яка представляє собою булевський вираз.

Узагальнене програмування – програмування без прив'язки до конкретного типу даних.

Узагальнення (generalization) – таксономічне відношення між більш загальним поняттям та менш загальним поняттям.

Управляючий клас (control class) – клас, який відповідає за координацію дій інших класів.

Успадкування (inheritance) – спеціальний концептуальний механізм, за допомогою якого більш спеціальні елементи включають в себе структуру і поведінку більш загальних елементів.

Фокус управління (focus of control) – спеціальний символ на діаграмі послідовності, який вказує період часу, протягом якого об'єкт виконує деяку дію, перебуваючи в активному стані.

Чиста віртуальна функція – віртуальна функція-метод класу, тіло якої не визначено

Шаблонні параметри – параметри, у яких в якості специфікаторів типу використовуються ідентифікатор зі списку параметрів шаблону.

Шаблон функції – оголошення функції, якому передує список параметрів шаблону.

A-B

Г-І

К

Л-М

О

П-Р

С

У-Ш