МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

В. Б. Бобков,
Ю. Є. Грудзинський,
К. В. Крилов

ПРОГРАМУВАННЯ - 2
ОБ'ЄКТНО – ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ

Навчальний посібник

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для здобувачів ступеня бакалавра
за освітньою програмою
«Автоматизація та комп'ютерно – інтегровані технології кібер - енергетичних систем»
спеціальності 151 Автоматизація та комп'ютерно – інтегровані технології

Електронне мережне навчальне видання

Київ
КПІ ім. Ігоря Сікорського
2023

Рецензент *Гагарін О. О.*, канд.тех.наук, доц.,
кафедри ІПЗЕ, НН ІАТЕ, «КПІ ім. Ігоря Сікорського»

Відповідальний редактор *Любицький С. В.*, старший викладач кафедри АЕП, НН ІАТЕ,
«КПІ ім. Ігоря Сікорського»

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського
(протокол № 8 від 02.06.2023 р.)
за поданням Вченої ради НН ІАТЕ
(протокол № 12 від 08.05.2023 р.)

Розглядаються основи об'єктно-орієнтованого підходу (ООП) в програмуванні на прикладі мови C++ [1]. Значну увагу приділено висвітленню ООП як способу повторного використання коду. Викладення матеріалу є досить стислим. В той же час, наведеної інформації цілком достатньо, щоб зрозуміти ідеологію ООП та спосіб практичного використання цього підходу. Уникаючи прикладів, що містять великі фрагменти програмного коду, автори наводять код, що дає змогу глибше зрозуміти основні положення ООП та механізми його реалізації. Посібник призначений для здобувачів ступеня бакалавра за спеціальністю 151 «Автоматизація та комп'ютерно-інтегровані технології». Буде також корисним для всіх, хто вивчає ООП та прагне зрозуміти його основні принципи.

Реєстр. № НП 21/22-695. Обсяг 3,9 авт. арк.

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

Зміст

Вступ

Об’єктно-орієнтоване програмування (ООП) є сучасним підходом в програмуванні, що відображає спосіб опису оточуючого світу та задач з представлення та обробки інформації, які в ньому виникають. В той же час, цей підхід визначає спосіб організації повторно використовуваного коду, що значно розширяє процедурний підхід. Сучасне програмне забезпечення стає все більш складним. В той же час, окрім суто специфічних задач, воно містить велику кількість типових задач, для яких можна створити та повторно використовувати стандартні рішення. Ці рішення представляються у вигляді бібліотек або цілих фреймворків, що реалізують деякі стандартні механізми. Принципи ООП складають ідеологічну основу для створення таких фреймворків, даючи відповідь на такі питання:

- Як організувати програмний код для повторного використання і як, при цьому, його параметризувати.
- Як забезпечити цілісність коду, поєднавши дані з алгоритмами їх обробки.
- Як забезпечити можливості розширення існуючої функціональності або часткової зміни поведінки.
- Як забезпечити можливість впровадити специфічну функціональність в існуючі механізми.

Існує велика кількість мов програмування, що втілюють ООП. В даному посібнику ООП розглядається на прикладі C++ [1], що є однією з найбільш гнучких мов. Та загальні принципи залишаються немінними при переході до інших мов ООП, змінюються лише синтаксичні конструкції та деякі деталі реалізації принципів.

1 Основні положення ООП

1.1 Визначення ООП

ООП - це парадигма програмування, що будується на представленні програми у вигляді сукупності взаємодіючих об'єктів, які є представниками певних класів, що організовані у ієрархію спадкування.

1.2 Основні положення ООП

Були сформульовані засновником цієї парадигми Аланом Кеєм [2]. Ці положення є наступними:

1) Все є об'єктами.
2) Об'єкти взаємодіють між собою.
   a. Взаємодія здійснюється шляхом надсилання повідомлень один одному.
   б. Повідомлення представляє собою запит на виконання певної дії, доповнений, за необхідності, параметрами, що конкретизують в який спосіб ця дія має бути виконана.
3) Кожен об'єкт має незалежну пам'ять, в якій він зберігає свої данні.
4) Об'єкти є екземплярами певних класів. Об'єкт є представником (екземпляром) певного класу, що описує загальні спільні властивості об'єктів.
5) Класи описують поведінку групи об'єктів. Об'єкти, що належать до одного і того ж класу, таким чином, мають спільну поведінку.
6) Класи організовані в ієрархічну структуру з єдиною вершиною, що називається ієрархією спадкування. Потомкам доступні властивості верхніх рівнів ієрархії.

1.3 Класи та об’єкти

Класи визначаються двома аспектами:

1) Це деяке узагальнення, що виражає властивості та поведінку певної групи об’єктів (в рамках цього аспекту, ми сприймаємо класи як певну категорію, деякий узагальнений опис).
2) Класи є комплексним типом з точки зору програмування, що поєднує в собі значення різних типів та набір функцій, який і визначає поведінку.

Виходячи з другого аспекту, ми використовуємо класи шляхом оголошення змінних. Такі змінні називаються екземплярами класу або об’єктами (змінна класу - об'єкт, який отримує незалежну ділянку пам'яті).

1.4 Абстрагування

Абстрагування – основний методологічний принцип ООП. Абстрагування передбачає узагальнену характеристику групи об’єктів, при якому відкидаються несуттєві елементи цієї характеристики з концентрацією на суттєвих елементах.

1.5 Принципи ООП

ООП базується на трьох принципах:

1) Інкапсуляція.

2) Спадкування.

3) Поліморфізм.

1.5.1 Інкапсуляція

Інкапсуляція – це приховування внутрішньої реалізації об’єкта системи, поєднання в класі даних та методів для роботи з ними. Інкапсуляція вирішує такі задачі:

1) Запобігання переведення об’єкта в некоректний стан. Некоректний стан об’єкта пов'язаний з некоректними значеннями його властивостей. Якщо закрити прямий доступ до властивостей та забезпечити можливість оперування ними за допомогою методів (оперування функціями), то всі зміни будуть відбуватись під контролем цих функцій, а алгоритм можна побудувати так, щоб запобігти некоректному стану.

2) Мінімізація числа зв’язків. Менша кількість зв’язків означає більше можливості щодо зміни внутрішньої реалізації класу, без необхідності зміни коду, що цей клас використовує.

Все сказане значно полегшує супроводження коду.

1.5.2 Спадкування

Спадкування – це технологія , що дозволяє створювати нові класи на базі вже існуючих, при цьому нові утворювані класи, що називаються похідними, успадковують властивості та поведінку тих класів, на базі яких вони утворюються. Спадкування вирішує дві задачі:

1) Організовує класи, певну ієрархію, що дозволяє їх впорядкувати, виходячи з загальноприйнятих принципів класифікації.

2) Повторне використання коду.

В нових класах можна змінити певну функціональність, перевизначивши відповідні функції, згідно із задачею що вирішується з використанням класів. Також можна додати нову функціональність, додавши нові функції.

1.5.3 Поліморфізм

Це механізм, що реалізує різну поведінку синтаксично однакового програмного коду, залежно від типу об'єкта , що в ньому використовується.

Базується на так званому поліморфному виклику функції, при якому виклик пов'язується з кодом, що буде виконуватись, шляхом динамічного зв'язування. Динамічне означає, що зв’язування буде відбуватись на етапі виконання програми. Поліморфізм вирішує задачу створення деякого універсального програмного коду, поведінка якого може змінюватись залежно від конкретної задачі. Зокрема, за його допомогою можна створювати універсальні системні механізми в рамках певних платформ (фреймворків), в яких можна забезпечувати точки входу для виконання специфічного користувацького коду.

1.6 Контрольні запитання

1. Сформулюйте основні положення ООП та поясніть їх.
2. Якими двома аспектами визначаються класи? Що являють собою класи з точки зору конструкцій мови програмування?
3. Що таке об’єкт? Чи є різниця між об’єктом та екземпляром класу?
4. Що таке інкапсуляція? Які задачі вона вирішує?
5. Що таке спадкування? Яким чином реалізується повторне використання коду за допомогою спадкування?
6. Що таке поліморфізм? Як здійснюється поліморфний виклик функцій?

Тема 2. Оголошення класів

2.1 Синтаксис оголошення класів

Синтаксис оголошення класу має наступний вигляд:

class <ім'я_класу>
{
    <оголошення елементів>
};

Для імені класу використовують ідентифікатори згідно загального правила створення ідентифікаторів мови С. При цьому слід надавати класам осмислені імена, що характеризують ту сутність описом якої клас є.

Ім'я класу надалі використовується в якості імені типу при об'явленні змінних класового типу, тобто об'єктів.

2.2 Елементи класу

Елементи класу можна поділити на дві категорії:

а) елементи-дані

б) елементи-функції

2.2.1 Елементи-дані

Елементи-дані використовуються для зберігання значення різних типів,

що характеризують об'єкт класу. Вони об'являються як звичайні змінні з вказанням їх типу та імені. Можна використовувати як базові типи, так і комплексні, в тому числі і класові.

Але слід зауважити, що при об'явленні елементів-даних вони не ініціалізуються. Це логічно: бо коли ми об'являємо клас, ми фактично об'являємо деякий шаблон, за яким будуть створюватись об'єкти цього класу. І доки не існує екземпляру - не існує змінної і не існує значення. Відповідно до об'явлень елементів-даних, буде виділятись пам'ять для їх збереження при створенні об'єкта.

2.2.2 Елементи-функції

Синтаксис дозволяє в об'явленні класу наводити, як оголошення функції у вигляді її прототипу, так і повну реалізацію функції (визначення функції). Але рекомендується використовувати варіант з прототипами, а визначати функцію окремо від оголошення та навіть в окремому файлі.

Зважаючи на ці рекомендації, слід використовувати спеціальний синтаксис заголовку функції:

<тип значення> <ім'я класу>::<ім'я функції>(<список формальних параметрів>)

:: - оператор визначення області бачення

2.3 Організація класів

Клас представляється у вигляді двох файлів:

а) заголовочний файл *.h з оголошенням класу, в якому функції представлені у вигляді прототипів.

б) файл реалізації *.cpp, в якому визначаються функції класу. При цьому у файл реалізації директивою #include включається відповідний заголовочний файл.

Заголовочний файл з оголошенням класу слід включати в усі файли з *.cpp-кодом, в яких використовується клас. Об'єкти класу об'являються як звичайні змінні, де в якості типу змінної виступає ім'я класу. При об’явленні об’єктів класу в C++ в такий спосіб, цей об’єкт утворюється і під нього виділяється необхідна пам'ять.

2.4 Доступ до елементів класу

Доступ до елементів класу будемо розглядати в двох аспектах:

а) управління можливістю доступу (як щось приховати та як щось відкрити)

б) реалізація (здійснення) доступу.

2.4.1 Управління можливістю доступу

Говорячи про доступ, ми маємо на увазі доступ до елементів класу з боку функції. По відношенню до класу всі функції можна поділити на дві категорії:

а) внутрішні функції, тобто функції, що належать класу

б) зовнішні функції, що не належать класу:

- функції інших класів
- глобальні функції

а) Внутрішні функції класу мають доступ до всіх його елементів.

б) Управління можливістю доступу з боку зовнішніх функцій.

Можливість доступу до елементів класу з боку зовнішніх функцій управляється за допомогою специфікаторів доступу. Специфікатор доступу зазначається у вигляді певного ключового слова з двокрапкою в кінці. Розрізняють три види доступу:

1) public:

Відкритий доступ.

Елементи класу, для яких визначено відкритий доступ, можуть використовуватись (чи доступні) в будь яких зовнішніх функціях.

2) private:

Закритий доступ.

Елементи, для яких визначено закритий доступ, доступні лише у внутрішніх функціях класу. Таким чином, інкапсуляцію ми здійснюємо приховуванням повністю внутрішньої реалізації шляхом визначення елементів як закритих.

3) protected:

Захищений доступ.

Елементи, для яких визначено захищений доступ, доступні крім внутрішніх функцій класу, ще й функціям похідного класу. Очевидно, що цей доступ використовується у зв'язку зі спадкуванням.

Специфікатор доступу може бути зазначений у будь-якому місці оголошення класу та розпочинає розділ доступу. Розділ доступу завершується іншим специфікатором, що розпочинає наступний розділ.

При об’явленні класу елементи-дані класу рекомендується об’являти як закриті, роблячи відкритими лише за умови якоїсь нагальної потреби. Загалом, вся відкрита частина класу складає його інтерфейс. Відповідно закрита частина класу складає його внутрішню реалізацію. Рекомендується розділи доступу розташовувати в певній послідовності: спочатку відкрита частина, потім захищена і в кінці закрита.

2.4.2 Дружні функції

Як зазначалось, закриті елементи класу доступні лише його внутрішнім функціям. Але існує можливість надати доступ до них окремим зовнішнім функціям. Такі функції називаються дружніми. Щоб визначити функцію як дружню, слід розмістити її прототип в об'явленні класу, використавши специфікатор friend на початку.

Зважаючи на те, що дружні функції порушують інкапсуляцію, їх слід використовувати за нагальної потреби.

2.4.3 Дружні класи

Можна зробити цілий клас дружнім до деякого іншого класу. Це означає, що всі функції дружнього класу мають доступ до закритих елементів. Щоб зробити клас дружнім, клас об’являється за допомогою такої конструкції:

friend class ім’я_класу

2.4.4 Реалізація доступу

а) Внутрішні функції.

Доступ до елементів класу здійснюється через об‘єкт класу або через вказівник на нього. Виключення становлять статичні змінні, що будуть розглянуті пізніше.

Внутрішні функції самі викликаються через об’єкт класу. Відповідно, вони мають доступ до елементів даних саме цього об’єкту. Доступ може бути здійснений просто за іменем елемента, при цьому неявно використовується доступ через спеціальний вказівник, який називається this. Отже, при виклику функції класу разом з аргументами цієї функції в стек передається адреса об’єкту, через який функцію викликано. Саме з цією адресою співставлений вказівник this. Для покращення читабельності коду вказівник this можна вказати явно при зверненні до елементу класу. Явний доступ через вказівник this стає обов’язковим у випадку конфлікту імен.

б) Здійснення доступу у зовнішніх функціях.

Доступ у зовнішніх функціях до елементів об'єкту здійснюється через об'єкт за допомогою операції "точка" (.), або через вказівник на об'єкт за допомогою операції "стрілка" (->). Такий доступ можливий тільки до відкритих елементів.

2.4.5 Функції геттери та сеттери

Згідно з наведеними вище рекомендаціями ми закриваємо дані, відносимо їх до внутрішньої реалізації. Це унеможливлює керування цими даними у зовнішніх функціях. Але існує потреба змінювати або отримувати значення цих елементів. Для здійснення таких операторів використовуються відповідні функції, що називаються геттерами та сеттерами:

GetXxx та *SetXxx*

У найпростішому випадку Get функція повертає значення елементу, а функція Set присвоює елементові значення аргументу. Але якщо існують обмеження на значення елемента, Set перевіряє ці обмеження. До того ж Get може попередньо зробити якесь обчислення, а не обов’язково повертає значення напряму.

2.5 Контрольні запитання

1. З яких частин складається оголошення класу?
2. На які дві категорії поділяються елементи класу? Як об’являються елементи обох категорій?
3. Як слід організовувати код класу?
4. Поясніть поняття доступу до елементів класу. Чому можливість управління доступом є важливою?
5. Як здійснюється управління доступом до елементів класу? Які існують типи доступу?
6. Як реалізується доступ до елементів класу у внутрішніх та зовнішніх функціях? Що таке вказівник this, куди він вказує?
7. Що таке дружня функція, для чого вона використовується? Що таке дружні класи?
8. Для чого використовуються функції-геттери та функції-сеттери?

3 Конструктори класу. Деструктор класу. Константні елементи класу. Константні об’єкти

При створенні об'єкту класу йому виділяється пам'ять. Вміст цієї пам'яті є випадковим. Відповідно елементи-дані класу набувають випадкових значень. Ці випадкові значення можуть виявитись некоректними. Отже, при створенні об'єкта він може опинитись в некоректному стані. Щоб цьому запобігти, слід забезпечити автоматичну ініціалізацію об'єкта коректними значеннями при його створенні. Оскільки ініціалізація – це певна дія, то для її здійснення слід описати відповідний алгоритм у вигляді деякої функції. Така функція називається конструктором класу.

Конструктор класу - це функція, що неявно (автоматично) викликається при створенні об'єкта для ініціалізації. На відмінну від інших функцій класу, конструктор має певні особливості.

3.1 Особливості конструктора

1) Назва конструктора збігається з назвою класу.
2) Оскільки конструктор викликається неявно, він не повертає значення оскільки його не можна використати. Тому в заголовку тип значення, що повертається, не вказується.

Якщо в класі не створено жодного конструктора, то конструктор за замовчуванням створюється автоматично. У разі, коли в класі є хоч один конструктор з аргументами, конструктор за замовчуванням автоматично не створюється.

3.2 Перевантаження конструкторів

Конструктор може мати набір перевантажених варіантів. Серед всіх перевантажених варіантів виділяється один, що називається конструктором за замовчуванням. Цей конструктор не потребує аргументів. Він може бути:

- конструктором без аргументів;
- конструктор, всі аргументи якого є аргументами за замовчуванням.

Якщо в класі явно не створено жодного конструктора, то конструктор за замовчуванням створюється автоматично. У разі, коли у класі є хоч один конструктор з аргументами, то він автоматично не створюється.

3.3 Оголошення об'єктів

При об'явленні об'єктів під них виділяється пам'ять і викликається конструктор. У разі, коли ми використовуємо простий синтаксис з вказанням просто імені об'єкта, для ініціалізації об'єкта буде викликано конструктор за замовчуванням.

Для того, щоб для ініціалізації об'єкта було викликано конструктор з аргументами, слід використовувати наступний синтаксис:

<ім'я класу> <ім'я змінної>(<список фактичних параметрів конструктора>);

В списку фактичних параметрів значення мають бути в порядку що відповідає порядку аргументів конструктора.

3.4 Функції конструктора

Конструктори класу використовуються для виконання наступних основних задач:

1) ініціалізація елементів-даних створюваного об'єкта

2) ініціалізація додаткових ресурсів, що використовується об'єктом

3) ініціалізація при перетвореннях в даний класовий тип з інших типів

4) ініціалізація копій об'єктів

3.4.1 Ініціалізація елементів-даних

Ініціалізацію елементів-даних можна здійснювати у два способи:

а) шляхом присвоювання значень у тілі конструктора

б) шляхом використання ініціалізаторів конструктора

Ініціалізатори використовуються для задання початкових значень елементам об’єкта. Ініціалізатори зазначаються в заголовку конструктора при його визначенні через двокрапку. Вони представляються у вигляді списку, де окремі ініціалізатори розділяються комами. Синтаксис окремого ініціалізатора є наступним:

<ім'я елемента> (<значення>)

В якості значення можна вказати як константу відповідного типу, так і аргументи конструктора

Приклад 3.1

```
class Complex
{
public:
        Complex();
        Complex(double x, double y)
private:
        double _x;
        double _y;
}

Comples::Complex(): _x(0), _y(0)
{
}

Comples::Complex(double x, double y): _x(x), _y(y)
{
}
```

3.4.2 Ініціалізація додаткових ресурсів

Об’єкт в процесі свого життя може використовувати додаткові ресурси. Прикладами є :

- дані, що зберігаються в додатковій динамічно виділеній області пам’яті;
- мережеві з’єднання;
- файлові ресурси.

Розглянемо цю функцію конструктора на прикладі класу, що оперує динамічним масивом цілих чисел Array.

Приклад 3.2

```
class Array
{
public:
        Array(int n);
private:
        int _n;
        int* _p;
}

Array::Array(int n)
{
        _n = n;
        _p = new int[n];
        for (int i = 0; i < n; i++)
```

```
        {
                _p[i] = 0;
        }
}

int main()
{
        Array a(10);
}
```

3.4.3 Конструктор при перетвореннях типів

Перетворення у значення класового типу (об'єкт) може здійснюватися явно та неявно.

Неявні перетворення здійснюються, наприклад, в таких випадках:

- при присвоюваннях, в разі невідповідності типів правого та лівого операндів;

- при виклику функції, коли один з формальних параметрів має класовий тип, а на його місці передається фактичне значення іншого типу;

- при поверненні з функції значення типу, відмінного від вказаного в прототипі, якщо в прототипі вказано даний класовий тип.

Процес перетворення значення деякого типу на об'єкт класового типу передбачає:

а) створення об'єкту з виділенням для нього відповідної ділянки пам'яті.

б) ініціалізацію створеного об'єкту.

Очевидно, що для ініціалізації має бути використане те значення, що перетворюється. Виконання такої задачі ініціалізації можливе за умови наявності відповідного конструктора. Тобто конструктора, що приймає на вхід перетворюваний тип (і це завжди конструктор з одним аргументом).

Приклад 3.3

```
class Complex
{
public:
        Complex();
        Complex(double x);
        void Print();
private:
        double _x;
        double _y;
```

```
}

Complex::Complex(): _x(0.0), _y(0.0)
{
}

Complex::Complex(double x)
{
        this->_x = x;
        this->_y = 0.0;
}

void Complex::Print()
{
        std::cout << "(" << _x << "," << _y << ")";
}

int main()
{
        Complex c;
        c = 5.0;
        c.Print();
}
```

3.4.4 Ініціалізація копій об’єкта

Копії об’єктів можуть створюватись явно та неявно. Явне створення копій відбувається при відповідному об’явленні об’єкта згідно такого синтаксису:

ім’я_класу ім’я_змінної(ім’я_оригіналу)

або

ім’я_класу ім’я_змінної = ім’я_оригіналу

Слід відрізняти останній приклад від прикладу нижче:

ім’я_об’єкту = ім’я_оригіналу

В такій ситуацій відбувається присвоювання, а не ініціалізація.
Неявне створення копій об’єктів має місце в таких випадках :

- при передачі об’єкта функції в якості аргументу за значенням;
- при поверненні з функції значення у вигляді об’єкту, якщо при цьому не використовується посилальний тип.

3.4.5 Конструктор копіювання

Якщо в класі явно не визначено конструктор копіювання, то його буде автоматично створено. Такий конструктор реалізує універсальний алгоритм копіювання, не залежний від структури об’єкта, що копіюється. Цей алгоритм базується на побітовому копіюванні (копіювання біт-у-біт).

Побітове копіювання не завжди може нас влаштовувати. Приклад – клас Array, що оперує динамічним масивом цілих чисел. Клас містить серед даних вказівник на динамічний масив. Це означає, що при створенні копії оригінал та копія будуть мати однакове значення цього вказівника. А це, в свою чергу, означає, що оригінал і копія будуть оперувати одним і тим же динамічним масивом, а це не є бажаним фактом. Внаслідок цього можуть виникати різні нештатні ситуації. Зокрема, оригінал та копія можуть мати різний час життя. І знищення одного з них може призвести до часткового руйнування іншого. Щоб запобігти вказаним проблемам, слід реалізовувати конструктор копіювання явно. При цьому його слід реалізувати так, щоб копія мала свій окремий динамічний масив з такими ж самими елементами, що й оригінал.

Прототип конструктора копіювання :

ім’я_класу(const ім’я_класу& ім’я_аргументу);

Слід звернути увагу, що аргумент конструктора копіювання має посилальний тип, оскільки у разі передачі аргументу за значенням, ми б отримати нескінченну рекурсію. При цьому для запобігання зміни аргументу в конструкторі копіювання використано модифікатор const. Конструктор копіювання зазвичай використовується в класах, що містять вказівники. Але це не означає, що конструктор копіювання потрібен завжди, коли є вказівник. Зокрема, об’єкти можуть працювати з якоюсь спільною ділянкою пам’яті, яка розділяється, а не належить кожному особисто.

В наведеному прикладі з класом Array ми маємо забезпечити НЕЗАЛЕЖНІ динамічні масиви (чи динамічні області пам'яті з даними) для копії та оригіналу.

Приклад 3.4

```
class Array
{
public:
        Array(const Array& original);
private:
        int _n;
        int* _p;
}
```

```
Array::Array(const Array& original)
{
	_n = original._n;
	_p = new int[_n];
	for (int i = 0; i < _n; i++)
	{
		_p[i] = original._p[i];
	}
}
```

3.5 Деструктор класу

Якщо об’єкт протягом свого існування створює певні ресурси, то при знищення об’єкта ці ресурси мають бути звільнені. Прикладом таких ресурсів є динамічний масив. Якщо не звільнити пам’ять від цього масиву після знищення об’єкту, то матиме місце утікання пам’яті. Для звільнення ресурсу використовується відповідна функція, що викликається автоматично (неявно) при знищенні об’єкта. Така функція називається ДЕСТРУКТОРОМ класу.

Особливості деструктора :

1) Ім’я деструктора утворюється з імені класу, перед яким ставиться значок “тильда” ~ (хвиляста лінія):
   ~ім’я_класу()
2) Деструктор не має аргументів.
3) Для деструктора не вказується тип значення, що повертається.

З вказаних особливостей очевидно витікає, що деструктор у класі може бути лише один. Приклад реалізації деструктора у класі Array.

Приклад 3.5

```
class Array
{
public:
	Array(int n);
	~Array();
private:
	int _n;
	int* _p;
}

Array::Array(int n)
```

```
{
	_n = n;
	_p = new int[_n];
	for (int i = 0; i < _n; i++)
	{
		_p[i] = 0;
	}
}

Array::~Array()
{
	if (_p)
	{
		delete[] _p;
	}
}
```

3.6 Константні елементи класу

Константні елементи класу - це елементи-данні, що не змінюються протягом існування об'єкту (і не можуть бути змінені). Вони об'являються з модифікатором const. Оскільки до константних змінних не може застосовуватись операція присвоювання, немає можливості проініціалізувати їх в тілі конструктора. Тому залишається лише можливість використання ініціалізатора конструктора.

Приклад 3.6

```
class A
{
public:
	A();
	A(int i);
private:
	const int _i;
}

A::A(): _i(0)
{
}

A::A(int i): _i(i)
{
}
```

3.7 Константні об’єкти

Константний об'єкт - це об'єкт, стан якого не змінюється протягом існування. Певний стан об’єкту характеризується набором значень його елементів даних. Отже, для константних об’єктів значення його елементів даних не змінюється.

З врахуванням інкапсуляції стан об’єкту може бути змінений лише за допомогою виклику відповідних відкритих функцій класу. Втім функції, що складають інтерфейс класу, можна поділити на функції, що змінюють стан об’єкту та функції, що не змінюють стан об’єкту. Щоб відслідковувати можливі зміни стану для константних об’єктів, слід розуміти, до якої з категорій відноситься функція, що викликається (функція, що не змінює стан об’єкту не становить загрозу для константного об’єкту з точки зору того, що його не можна змінювати). Для того, щоб розрізняти вказані дві категорії функцій вводиться поняття константної функції. Отже, константна функція – це така функція, яка не змінює стану об’єкта. Виклик такої функції для константного об’єкту завжди дозволений компілятором. Щодо неконстантних функцій, то їх виклик для константного об’єкта викликає реакцію компілятора у вигляді помилки або попередження. Щоб об’явити константу функцію, слід поставити модифікатор const в кінці її прототипу.

3.8 Контрольні запитання

1. Що таке конструктор класу? До чого він використовується?
2. В чому полягають особливості конструктора в порівнянні з іншими функціями класу?
3. Що таке конструктор за замовченням?
4. Як слід об’явити об’єкт, щоб для його ініціалізації було викликано конструктор з аргументами?
5. Перерахуйте функції конструктора.
6. Які є два способи інціалізації елементів-даних в конструкторі? Що таке ініціалізатор конструктора?
7. Які додаткові ресурси може ініціалізувати конструктор?
8. Яку роль відіграє конструктор при перетвореннях типів? За яких умов можливе перетворення деякого типу в заданий класовий тип?
9. Що таке конструктор копіювання, в яких ситуаціях він викликається?

10. Коли та чому потрібно обов’язково реалізовувати конструктор копіювання?
11. Який вигляд має прототип конструктора копіювання? Поясніть вимоги щодо типу аргументу.
12. Що таке деструктор класу? До чого він використовується?
13. В чому полягають особливості деструктора в порівнянні з іншими функціями класу? Скільки деструкторів може мати клас?
14. В чому полягають особливості константних елементів класу? Як вони об’являються?
15. В який спосіб ініціалізуються константні елементи класу?
16. Що таке константний об’єкт?
17. В чому полягають особливості використання константних об’єктів?
18. Що таке константні функції? Як вони об’являються та для чого використовуються?

Тема 4. Статичні елементи класу. Композиція класів

Деякі данні можуть характеризувати клас в цілому, а не окремі його екземпляри. В цьому разі нема сенсу зберігати окремі копії цих даних в кожному екземплярі. Данні, в такому разі, мають існувати в єдиному екземплярі на весь клас. Для вирішення задачі зберігання та оперування такими даними використовуються СТАТИЧНІ ЕЛЕМЕНТИ КЛАСУ. Як і не статичні (ЕКЗЕМПЛЯРНІ) елементи, статичні елементи поділяються на елементи-данні та елементи-функції.

4.1 Статичні елементи-данні

4.1.1 Оголошення

Статичні елементи-данні об'являються з використанням модифікатора static. Для статичних елементів можуть використовуватись ті ж самі специфікатори доступу, що і до нестатичних елементів.

```
class A
{
public:
        static int count;
};
```

4.1.2 Ініціалізація

Ініціалізація статичних елементів-даних здійснюється на глобальному рівні.

Синтаксис ініціалізації є наступним:

<тип> <ім'я класу>::<ім'я елемента> = <значення>;

Приклад 4.1

```
class A
{
public:
        A();
        static int count;
};

int A::count = 0;

A::A() {
        count++;
}
```

4.1.3 Доступ до статичних елементів

Існує два способи доступу до статичних елементів:

а) через будь-який об'єкт класу

```
int main() {
        A a1;

        std::cout << a1.count;
}
```

б) через ім'я класу з використанням подвійної двокрапки

```
int main() {
        A a1;
        std::cout << A::count;
}
```

Рекомендується використовувати саме варіант б).

Значення статичних елементів-даних в деяких випадках слід приховувати, тобто відповідні статичні елементи робити закритими. Така потреба виникає в ситуації, коли зміна значення має відбуватись лише під контролем певних функцій. В цьому разі ми використовуємо специфікатор private при об'явленні елемента. А при оперуванні цим елементом (встановлення/отримання значення) слід визначити так звані СТАТИЧНІ ФУНКЦІЇ.

4.2 Статичні функції

Статичні функції об'являються в класі з модифікатором static. Головною особливістю статичної функції є те, що в її тілі НЕ МОЖНА здійснювати доступ до нестатичних змінних (нестатичних даних), оскільки ця функція не отримує вказівник this. Зворотнє твердження не вірне: нестатичні функції можуть звертатись до статичних елементів.

4.3 Композиція класів

Композицією класів називається ситуація, коли елементом якогось класу є об’єкт іншого класу. При композиції є певні особливості ініціалізації. Як відомо за ініціалізацію всіх елементів класу відповідає конструктор класу. Але у разі, коли клас містить якісь внутрішні об’єкти, то очевидно, що він має бути проініціалізований конструктором того класу, до якого цей внутрішній об’єкт

належить. Для цього слід забезпечити виклик конструктора класу внутрішнього об’єкту, коли викликається конструктор “зовнішнього класу”, який цей об’єкт містить. Власне, подібний неявний виклик і здійснюється.

Очевидно, що без використання якихось додаткових синтаксичних конструкцій може бути викликаний лише конструктор класу внутрішнього об’єкту за замовчуванням. Втім, такий конструктор здійснить ініціалізацію за замовчуванням. Якщо слід встановити якісь інші значення для внутрішнього об’єкта, то це треба буде зробити в тілі зовнішнього класу. Виходить, що викликаний конструктор за замовчуванням зробить зайву роботу. До того ж, виникає критична проблема, якщо в класі внутрішнього об’єкту немає конструктора за замовчуванням. Для вирішення цієї проблеми використовується ініціалізатор конструктора зовнішнього класу. В ньому в якості елемента, що ініціалізується, вказується внутрішній об’єкт, а в дужках зазначаються фактичні значення, що передаються конструктору класу внутрішнього об’єкту.

Приклад 4.2

```
class Person {
    public:
        Person(int day, int month, int year);
    private:
        Date _birthday;
};

Person::Person(int day, int month, int year): _birthday(day, month, year)
{
}
```

4.4 Контрольні запитання

1. Для чого використовуються статичні елементи класу?
2. Як ініціалізуються статичні елементи-дані?
3. Як здійснюється доступ до статичних елементів класу?
4. Для чого використовуються статичні функції класу? Якими елементами класу вони можуть оперувати?
5. Чи можуть нестатичні функції звертатись до статичних елементів класу?
6. Що таке композиція класів?
7. В чому полягають особливості ініціалізації об’єкту при використанні композиції класів?

Тема 5. Перевантаження операторів

Операція – це дія над деякими значеннями, що називають операндами, з отриманням певного результуючого значення. Оскільки з цього визначення видно, що операція схожа на функцію, то цілком природньо, що механізм перевантаження операторів базується на реалізації спеціальних функцій, що називаються операторними функціями. Операторна функція має назву, що складається з ключового слова operator та знаку відповідної операції, що реалізується за допомогою цієї функції. Кількість аргументів операторної функції визначається кількістю операндів відповідної операції. Значення, що повертається операторною функцією, в загальному випадку, може бути будь-якого типу. Для більшості операторів перевантаження може бути здійснене у два способи:

- за допомогою внутрішньої операторної функції
- за допомогою глобальної (зовнішньої) операторної функції.

Операторна функція здійснює дії над операндами, які є об’єктами класового типу. Тому, в загальному випадку, вона повинна мати доступ до всіх (в тому числі закритих) елементів класу. Для того щоб забезпечити певній зовнішній функції доступ до закритих елементів, її потрібно об’явити як дружню функції.

5.1 Аргументи операторних функції

Кількість аргументів операторної функції визначається кількістю операндів операторної функції. Принаймні один з цих операндів має відноситись до того класового типу, до якого перевантажуються операції. Кількість аргументів при цьому залежить від способу перевантаження:

а) У випадку перевантаження за допомогою внутрішньої функції, ця функція викликатиметься через об’єкт класового типу. Цей об’єкт буде виступати в якості лівого операнду операції. Відповідно, кількість аргументів буде на один менше за кількість операндів.

б) У випадку перевантаження за допомогою зовнішньої функції кількість аргументів відповідає кількості операндів. Передаються обидва операнди через аргументи.

5.2 Правила перевантаження операторів

1) Можуть перевантажуватись лише ті операції, що визначенні для базових типів. Нові операції утворювати не можна. Проте існує ряд виключень. Не перевантажуються наступний операції:

a) .
b) .*
c) ?:
d) ::
e) sizeof

2) Кількість операндів перевантажуваної операції має співпадати з кількістю операндів відповідної операції над базовими типами.

3) Пріоритет та асоціативність перевантажених операторів такий самий, як для операторів для базових типів.

4) Операції мають бути інтуїтивно зрозумілими. Їх поведінка має відповідати поведінці для операції над базовими типами.

5.3 Перевантаження унарних операторів

Унарні операції перевантажуються шляхом реалізації або внутрішньої операторної функції без аргументів, або зовнішньої функції з одним аргументом. Розглянемо перевантаження унарних операторів на прикладі операції "унарний мінус" для класу комплексних чисел.

5.3.1 Перевантаження за допомогою внутрішньої функції

а) Фрагмент оголошення класу.

```
class Complex
{
public:
        Complex(double x, double y);
        void Print();
        Complex operator-();

private:
        double _x;
        double _y
};

Complex::Complex(double x, double y)
{
        _x = x;
        _y = y;
}
```

```
Complex::Print()
{
	std::cout << "(" << _x << ", " << _y << ")";
}
```

б) Реалізація операторної функції.

Об'єкт, через який опрераторна функція буде викликатись, змінюватись не буде. Натомість буде утворюватися новий об'єкт, у якого зміняться знаки на протилежні у дійсної та уявної частинах.

```
Complex::Complex(double x, double y)
{
	_x = x;
	_y = y;
}

Complex::Print()
{
	std::cout << "(" << _x << ", " << _y << ")";
}

Complex Complex::operator-()
//1. операнд не змінює свого знаку
//2. утворюється нове значення з протилежним знаком
{
	Complex res(-this->_x, -this->_y);
	return res;
}
```

в) Використання операції.

Коли компілятор зустрічає вираз виду –c, в якому застосовується перевантажена операція, він генерує виклик c.operator-():

```
-c -> c.operator-()

int main()
{
	Complex c(1.0, 2.0);
	(-c).Print();
	std::cout << std::endl;
	c.Print();
}
```

5.5.2 Перевантаження за допомогою зовнішньої функції

а) Фрагмент оголошення класу.

```
class Complex
{
        friend Complex operator-(const Complex& c);
        ...
};
```

Якщо операція не передбачає зміни операндів, ми завжди ці операнди будемо передавати у вигляді посилання на константний об'єкт (ефективна передача аргументу).

б) Реалізація операторної функції.

```
Complex operator-(const Complex& c)
{
        Complex res(-c._x, -c._y);
        return res;
}
```

в) Використання операції.

Коли компілятор зустрічає вираз виду –c, в якому застосовується перевантажена операція з використанням зовнішньої функції, якій передається операнд як аргумент, він генерує виклик operator-(c):

```
-c  ->  operator-(c)
```

5.6 Перевантаження операції інкременту і декременту

В порівнянні з іншими унарними операціями операції інкременту/декременту мають такі особливості:

1) Операції є різновидами операції присвоювання. Тобто передбачається, що вони змінюють операнд.

2) Операції існують в двох формах: префіксна та постфіксна.

Префіксна форма повертає в якості результату вже змінений операнд, а постфіксна - операнд в стані, що був до зміни.

Наявність двох форм вимагає реалізацію двох операторних функцій. Але аргумент операторної функції чітко визначений. Тому, щоб мати змогу реалізувати два варіанти застосовується прийом, що полягає у введенні штучного

формального аргументу для постфіксної форми. Цей формальний аргумент має тип int.

Розглянемо перевантаження операторів інкременту/декременту на прикладі інкременту для класу комплексних чисел.

5.6.1 Перевантаження за допомогою внутрішньої функції

а) Фрагмент оголошення класу.

```
class Complex
{
public:
        Complex operator++();
        Complex operator++(int);
        ...
};
```

Перше оголошення операторної функції відповідає префіксній формі, друге - постфіксній.

б) Реалізація операторних функцій.

```
Complex Complex::operator++()
{
        Complex res(++this->_x, this->_y);
        return res;
}
Complex Complex::operator++(int)
{
        Complex res(this->_x++, this->_y);
        return res;
}
```

в) Використання операції.

Зустрівши вираз ++c компілятор згенерує виклик c.operator++():

```
++c -> c.operator++()
```

Таким чином, буде виконано інкремент в префіксній формі.

Зустрівши ж вираз c++ компілятор згенерує виклик c.operator(0):

```
c++  ->  c.operator++(0)
int main()
{
        ...
        c.Print();
```

```
        std::cout << std::endl;
        (++c).Print();
        std::cout << std::endl;
        c.Print();
        std::cout << std::endl;
        (c++).Print();
        std::cout << std::endl;
        c.Print();
}
```

5.6.2 Перевантаження за допомогою зовнішньої функції

а) Фрагмент оголошення класу.

```
class Complex

{
        friend Complex operator++(const Complex& c);
        friend Complex operator++(const Complex& c, int);
        ...
};
```

Зауважте, що не слід використовувати модифікатор const для аргументу операторної функції, оскільки операція має змінювати операнд, тобто об’єкт, що передається в якості фактичного параметра.

б) Реалізація операторних функцій.

```
Complex operator++(Complex& c)
{
        Complex res(++c._x, c._y);
        return res;

}
Complex operator++(Complex& c, int)
{
        Complex res(c._x++, c._y);
        return res;
}
```

в) Використання операції.

Так само як в попередньому випадку.

5.7 Перевантаження унарної операції перетворення типу

Перетворення типів можуть бути явними і неявними. Явні перетворення типів використовуються шляхом явного застосування у вигляді назви типу, до якого здійснюється перетворення, в круглих дужках.

Неявні перетворення здійснюються в таких випадках:

а) При присвоюванні, коли типи лівого і правого операнду не співпадають.

Правий = Лівий

б) При передачі значень у функцію, коли тип фактичного значення не співпадає з типом аргументу.

в) При повернені значення з функції, коли тип виразу оператора return не співпадає з типом значення відповідно до прототипу.

Очевидно, що неявне та явне перетворення можливі лише за умови, коли визначений алгоритм такого перетворення.

Для класових типів алгоритми перетворень до інших типів мають бути реалізовані у вигляді відповідних перевантажених операторів. Для цього ми реалізуємо операторну функцію з назвою наступного виду:

operator ім'я_типу_до_ якого_здійснюється_перетворення()

Перевантаження операції приведення типу має особливості:

1) операція може перевантажуватись за допомогою лише внутрішньої функції.

2) для операторної функції не вказується тип значення що повертається, оскільки тип результату визначається типом до якого здійснюється приведення та вказується в назві операторної функції.

Розглянемо перевантаження на прикладі класу Complex. Реалізуємо операцію приведення до типу double. Результат приведення - дійсна частина комплексного числа.

а) Фрагмент оголошення класу

```
class Complex
{
public:
        operator double();
        ...
}
```

б) Реалізація операторної функції.

```
Complex::operator double()
{
        return this->_x;
}
```

в) Використання операції.

Зустрівши вираз (double)c, компілятор згенерує виклик виду c.operator double():

(double)c -> c.operator double()

Аналогічний виклик буде згенеровано у випадку неявних перетворень Complex до double.

```
int main()
{
        Complex c(1.0, 2.0);
        std::cout << std::endl;
        std::cout << (double)c; // явне перетворення
        std::cout << c; // неявне перетворення
}
```

5.8 Перевантаження бінарних операторів

Перевантаження бінарних операторів здійснюється:

1) за допомогою внутрішньої функції з одним аргументом

2) за допомогою зовнішньої функції з двома аргументами

5.8.1 Перевантаження за допомогою внутрішньої функції

Розглянемо приклад операції додавання для класу Complex.

а) Фрагмент оголошення класу

```
class Complex
{
public:
        Complex(double x);
        Complex(double x, double y);
        void Print();
        Complex operator+(const Complex& c);
private:
```

```
        double _x;
        double _y;
};
```

б) Реалізація операторної функції.

```
Complex::Complex(double x): _x(x), _y(0)
{
}
Complex::Complex(double x, double y): _x(x), _y(y)
{
}
void Complex::Print()
{
        std::cout << "(" << _x << ", " << _y << ")";
}
Complex Complex::operator+(const Complex& c)
{
        Complex res(this->_x + c._x, this->_y + c._y);
        return res;
}
```

в) Використання операції.

```
int main()
{
        Complex c1(1.0, 2.0), c2(7.0, 25.0);
        (c1 + c2).Print();
        std::cout << std::endl;
        (c1 + 10.0).Print();
}
```

Якщо є два об'єкти (комплексні числа) c1 та c2, то зустрівши вираз c1+c2 компілятор згенерує виклик c1.operator+(c2):

```
c1 + c2   ->   c1.operator+(c2)
```

Зустрівши ж вираз c1 + 10.0 компілятор згенерує, за зразком, наступний вираз c1.operator+(10.0):

```
c1 + 10.0  ->   c1.operator+(10.0)
```

В такому виклику тип фактичного параметра (10.0) відрізняється від типу формального параметра. Але, оскільки в класі передбачено конструктор з одним аргументом типу double, буде здійснене неявне перетворення 10.0 -> (10.0, 0.0) на об'єкт класу Complex. І вже цей об'єкт буде переданий в операторну функцію (тобто брати участь в додаванні).

Інша ситуація матиме місце в наступному випадку:

10.0 + c1 -> ???

Коли зустрічається вираз 10.0 + c, то відповідний виклик операторної функції не може бути згенерований, оскільки лівий операнд не відповідає типу Complex. Таке додавання виявляється неможливим. Це означає, що через перевантаження за допомогою внутрішньої функції операція + втратила комутативність у разі, коли один з операндів НЕ має тип Complex та може бути перетворений.

Отже, при реалізації операції додавання у вигляді внутрішньої операторної функції, операція не є повністю симетричною щодо операндів. Це пов'язано з тим, що операнди не рівноправні: через лівий операнд викликається операторна функція, а правий передається їй в якості аргументу. Ситуацію можна виправити, якщо здійснити перевантаження за допомогою зовнішньої функції.

5.8.2 Перевантаження за допомогою зовнішньої функції

а) Фрагмент оголошення класу

```
class Complex
{
        friend Complex operator+(const Complex& c1, const Complex& c2);
};
```

б) Реалізація (визначення) операторних функцій.

```
Complex operator+(const Complex& c1, const Complex& c2)
{
        Complex res(c1._x + c2._x, c1._y + c2._y);
        return res;
}
```

в) Використання операції.

Зустрівши вираз c1 + c2 компілятор згенерує виклик operator+(c1, c2):

c1 + c2 -> operator+(c1, c2)

Зустрівши вираз c1 + 10.0 компілятор згенерує виклик operator+(c1, 10.0):

c1 + 10.0 -> operator+(c1, 10.0)   10.0 -> (10.0, 0.0)

Зустрівши вираз 10.0 + c1 компілятор згенерує виклик operator+(10.0, c1):

10.0 + c1 -> operator+(10.0, c1)

Очевидно, що при такому перевантажені операція стає комутативною. Це має місце тому, що в даному випадку операнди є рівноправними.

5.9 Перевантаження операторів присвоювання

Відомо, що операції присвоювання діляться на два типи:

- складене (являє комбінацію операції присвоєння та додавання лівого операнду);

- просте.

Зосередимось на перевантаженні простого присвоєння. Виходячи зі змісту, який є очевидним для базових типів, при застосуванні цієї операції лівий операнд має стати копією правого операнду.

Розглядаючи конструктор копіювання, ми зазначили, що копіювання може бути здійснено за допомогою стандартного універсального алгоритму побітового копіювання незалежно від типу об’єкту, що копіюється. Це означає, що може бути реалізований універсальний алгоритм присвоювання. Саме цей алгоритм побітового копіювання застосовується, якщо для певного класу не перевантажено операцію присвоювання.

Побітове копіювання не завжди може нас влаштовувати. Приклад – клас Array. Клас містить серед даних вказівник на динамічний масив. Це означає, що при присвоюванні лівий операнд та правий будуть мати однакове значення цього вказівника. А це означає, що вони будуть оперувати одним і тим же динамічним масивом, внаслідок чого можуть виникати різні нештатні ситуації, аналогічні тим, що описані при розгляді конструктора копіювання. Алгоритм операції присвоювання буде схожий на конструктор копіювання.

Операція присвоювання може перевантажуватись лише за допомогою внутрішньої функції.

При реалізації операції присвоєння для певного класу слід дотримуватись особливостей поведінки, притаманних присвоюванню базових типів. Зокрема, оператор присвоювання має повертати значення, що співпадає з новим значенням лівого операнду після присвоювання. Звідси витікає тип значення, що повертається операторною функцією.

Різниця конструктора копіювання і оператора присвоювання: конструктор копіювання викликається для ініціалізації об’єкта, пам'ять для динамічного масиву ще не було виділено, а оператор присвоєння застосовується, коли об'єкт вже існує і існує певний динамічний масив.

Отже, слід врахувати наступні особливості при реалізації оператора присвоювання:

а) оператор присвоєння викликається для вже існуючого об’єкта. Тому перед виділенням нової динамічної пам’яті треба звільнити стару, що була попередньо виділена.

б) окремо слід обробити ситуацію із само присвоюванням, коли лівий і правий операнди – це один і той самий об’єкт. В цьому разі жодних дій робити не треба. Само присвоювання виявляється за допомогою умовного this == &a.

в) з операції присвоювання слід повернути копію лівого операнду в стані, якого він набув після присвоювання. Для цього використовують оператор виду return *this.

Можна реалізувати декілька операторів присвоювання, що відрізняються типом правого операнду. Розглянемо перевантаження операції присвоювання на прикладі класу Array.

а) Фрагмент оголошення класу

```
class Array
{
public:
        Array operator=(const Array& a);
};
```

б) Реалізація операторної функції.

```
Array Array::Array operator=(const Array& a)
{
        if (this == &a) // адреса лівого і правого операнда
        {
                return *this;
        }

        if (_p)
        {
                delete[] _p; // звільнили пам'ять, яку до цього використовували
        }
        _N = a._N;
        _p = new int[_N]; // пам`ять для лівого операнду

        for (int i = 0, i < _N, i++)
        {
                _p[i] = a._p[i];
        }
return *this // повернути копію лівого операнда
}
```

5.10 Перевантаження первинних операторів

Первинні операції можуть бути перевантажені лише за допомогою внутрішніх функцій.

5.10.1 Перевантаження операції індексації

Операція індексації використовується для доступу до конкретного елементу певного набору. Для ідентифікації елементів, доступ до яких здійснюється, використовується індекс. Загалом індекс може бути будь-якого типу. Відповідно, операторна функція для перевантаження операції індексації має один аргумент довільного типу, який і виступає в якості індексу.

Найчастіше при цьому використовується цілочисельний індекс. Ще одним прикладом часто вживаного індексу є символьний рядок. Операція індексації дозволяє створювати асоціативні масиви.

Розглянемо особливості реалізації на прикладі класу динамічного масиву Array. Операція індексації буде повертати деякий елемент масиву за цілочисельним індексом.

а) Фрагмент оголошення класу

```
class Array
{
public:
        Array(int N);
        void Print();
//      int operator[](int index);
//      Це неправильний варіант, оскільки повертається копія значення, що не може бути L-виразом
        int& operator[](int index); //повертається оригінал значення, який може бути L-виразом

private:
        int _N;
        int* _p;
};
```

Слід звернути особливу увагу на тип значення що повертається операторною функцією operator[](). Значення повертається за посиланням. В результаті з функції повертається не копія елемента масиву, а його оригінал. Це робить значення, що повертається лівостороннім значенням (L-виразом), тобто надає можливість його використання в лівій частині операції присвоювання.

б) Реалізація операторної функції.

```
Array::Array(int N)
{
	_N = N;
	_p = new int[_N];

	for (int i = 0; i < _N; i++)
	{
		_p[i] = 0;
	}
}
void Array::Print()
{
	for (int i = 0; i < _N; i++)
	{
		std::cout << _p[i] << " ";
	}
}
```

```
int& Array::operator[](int index)
{
// Тут повинна бути обов'язкова перевірка виходу індексу за межі масиву!!!!!
	return _p[index];
}
```

в) Використання операції.

```
std::cout << std::endl << a[2];
a[2] = 5;
std::cout << std::endl << a[2];
a.Print();
```

Зустрівши вираз a[2] компілятор згенерує виклик функції:

```
a[2]  -> a.operator[](2)
```

Буде повернуто елемент з індексом 2.

Якщо б операторна функція не повертала значення посилального типу, то присвоювання

```
a[2] = 5
```

було б неможливим.

5.10.2 Перевантаження операції круглі дужки

Головною особливістю операції круглі дужки є той факт, що відповідна операторна функція може мати довільну кількість аргументів. Кількість

аргументів визначається задачею, яку вирішує операція. Операторна функція для операції круглі дужки може навіть мати змінну кількість аргументів.

Розглянемо перевантаження операції круглі дужки на прикладі класу Array.

Задача операції: повернути об'єкт, що містить частину масиву (елементи вибираються підряд).

а) Фрагмент оголошення класу

```
class Array
{
public:
        ...
        Array operator() (int startIndex, int count);
};
```

б) Реалізація операторної функції.

```
Array Array::operator() (int startIndex, int count)
{
        if (startIndex > count || startIndex + count > count || count < startIndex)
        {
                exit(1);
        }

        Array res(count);
        for (int i = 0; i < count; i++)
        {
                res._p[i] = this->_p[startIndex+i];
        }
        return res;
 }
```

в) Використання операції.

```
Array a(5);
a[2] = 5;
a[3] = 7;
a.Print();
Array a1 = a(2, 2);
a1.Print();
```

5.11 Контрольні запитання

1. Що лежить в основі перевантаження операторів? Що треба зробити, щоб перевантажити операцію?

2. Які два способи перевантаження операторів існують? З яких міркувань обирається один з цих способів?
3. Назвіть правила перевантаження операторів. Чи можна визначати додаткові операції, яких немає для базових типів? Чи можна управляти пріоритетом операторів при перевантаженні?
4. В чому полягають особливості операторних функцій в порівнянні зі звичайними функціями класу?
5. Від чого залежить кількість аргументів операторної функції?
6. Як здійснюється перевантаження унарних операторів?
7. В чому полягають особливості перевантаження операторів інкременту та декременту?
8. Як здійснюється перевантаження бінарних операторів?
9. В чому полягають особливості перевантаження операторів присвоювання?
10. Чи можна здійснити присвоювання для змінної класового типу, якщо у відповідному класі не перевантажено операцію присвоювання?
11. Назвіть спільні риси перевантаженої операції присвоювання та конструктора копіювання? В чому полягають відмінності між ними?
12. Назвіть особливості перевантаження операції індексації.
13. Назвіть особливості перевантаження операції ().

Тема 6. Спадкування

Спадкування - це можливість створення нових класів на базі вже існуючих. Спадкування допомагає реалізувати принцип повторного використання коду. При спадкуванні може бути здійснено:

1. Зміну певної функціональності існуючого класу у похідному.
2. Додавання нової функціональності.
3. Обмеження функціональності.

Клас, на основі якого утворюється новий клас, називають БАЗОВИМ. А той клас, що утворюємо, називається ПОХІДНИМ. При спадкуванні похідний клас успадковує ВСІ елементи базового класу. Отже, похідний клас складається із двох категорій елементів:

1. Успадковані від базового класу.
2. Власні, визначені у похідному класі.

Синтаксис оголошення похідного класу є наступний:

```
class ім'я_похідного_класу : тип_спадкування ім'я_базового_класу
{
    оголошення_елементів
}
```

Тут наведено простий синтаксис, коли базовий клас лише один. Надалі буде розглянуто множинне спадкування, при якому є декілька базових класів.

Нижче наведено приклад класу B, похідного від класу A. При цьому B містить успадкований елемент i та власний j.

Приклад 6.1

```
class A
{
public:
   int i;
};

class B : public A
{
public:
   int j;
};
```

6.1 Перевизначення елементів в похідному класі

Перевизначення можливе як для елементів-даних, так і елементів-функцій. Перевизначення має місце, коли в похідному класі є такі ж самі елементи, що і в базовому класі. Під співпадінням ми розуміємо співпадіння типу та імені у випадку елементів-даних та співпадіння прототипів у випадку елементів-функцій. Перевизначення не означає зникнення успадкованого елемента. Елемент буде існувати у двох варіантах:

1. успадкований
2. власний

Приклад 6.2 Перевизначення елементів-даних та елементів-функцій.

```
class A
{
public:
        int i;
        void F();
};

class B : public A
{
public:
        int i;
        int j;
        void F();
};
```

6.2 Доступ до елементів похідного класу

6.2.1 Доступ у внутрішніх функціях

а) Можливість доступу.

Очевидно, що власні внутрішні функції похідного класу мають доступ до всіх власних елементів. При цьому, власні функції мають доступ лише до відкритих та захищених успадкованих елементів. Закриті успадковані елементи власним функціям похідного класу недоступні. Очевидно також, що успадковані функції мають доступ до успадкованих елементів.

б) Реалізація доступу.

У внутрішніх функція похідного класу доступ до елементів, якщо він можливий, здійснюється через явний або неявний вказівник this. У випадку, коли має місце перевизначення елемента, до якого здійснюється доступ, неявно (за

замовченням) доступ здійснюється до власного елемента. Втім є можливість доступитись і до успадкованого елемента вказавши це явно. Для цього використовується операція області бачення (подвійна двокрапка):

ім'я_базового_класу :: ім'я_елементу

Часто при перевизначенні функцій у похідних класах в їх тілі в певний момент викликається функція базового класу. Розглянемо приклад.

Приклад 6.3

```
class A
{
public:
   int i;
   void Print();
};

void A::Print()
{
   std::cout  << "A::i=" << i << std::endl;
}

class B : public A
{
public:
   int i;
   int j;
   void Print();
};

void B::Print()
{
   //std::cout << "A::i=" << A::i<< std::endl;
   A::Print();
   std::cout << "B::i=" << i << std::endl;
   std::cout << "B::j=" << j << std::endl;
}
```

У прикладі перевизначення в похідному класі функції для виводу успадкованих елементів викликається функція Print() базового класу з явним вказанням цього класу (Приклад 6.3). Отже, реалізуючи перевизначену функцію, ми розширяємо

функціональність успадкованої функції, зберігаючи тим не менш успадковану функціональність шляхом виклику цієї успадкованої функції в функції похідного класу.

6.2.2 Доступ у зовнішніх функціях

а) Можливість доступу.

1) Власні елементи.

Доступ до власних елементів визначається специфікаторами доступу, вказаними при об’явленні цих елементів. Доступні лише відкриті власні елементи та для функцій наступного похідного класу – захищені.

2) Успадковані елементи.

Доступ до успадкованих елементів визначається специфікаторами доступу при об’явлені в базовому класі, а також типом спадкування. Тип спадкування вказується перед типом базового класу у вигляді одного з трьох специфікаторів: public, protected, private.

Типи спадкування:

- *Public* - доступ зберігається таким, який визначається при об’явленні елементів в базовому класі.
- *Protected* – при такому спадкуванні відкриті елементи стають захищеними.
- *Private* – всі успадковані елементи стають закритими. Все, що успадковується від базового класу закривається.

6.3 Розміщення полів похідного класу в пам’яті

При створені об’єкта похідного класу в пам’яті спочатку розміщуються успадковані елементи-дані, а потім власні.

6.4 Вказівники базового типу

Похідний клас успадковує всі елементи базового класу. Отже об’єкт похідного типу має всі властивості та поведінку базового типу. Виходячи з цього, з об’єктом похідного типу можна взаємодіяти через вказівник базового типу. При цьому вказівник базового типу може містити адресу будь-якого об’єкту похідного типу. При присвоюванні вказівникові базового типу адреси об’єкта похідного типу не вимагається приведення типу вказівника.

Приклад 6.4

```
A a;
B b; // B : A
A* p1 = &a;
```

Зауваження: через вказівник базового типу можна доступатися лише до успадкованих елементів об’єкта похідного типу. Виключення становлять віртуальні функції, що будуть розглянуті пізніше.

Для доступу до власних елементів об’єкту похідного класу через вказівник базового типу слід спочатку здійснити приведення вказівника до похідного типу.

Приклад 10.5

```
class A
{
public:
   void F1();
};

void A::F1()
{
   std::cout << "A::F1\n";
}

class B : public A
{
public:
   void F1();
   void F2();
};

void B::F1()
{
   std::cout << "B::F1\n";
}

void B::F2()
{
   std::cout << "B::F2\n";
}

int main()
{
```

```
    A a;
    a.F1();  // вивід – A::F1

    B b;
    b.F1(); // вивід – B::F1
    b.A::F1(); // вивід – A::F1

    A* p1 = &a;
    p1->F1(); // вивід – A::F1

    A* p2 = &b;
    p2->F1(); // вивід – A::F1

    //p2->F2(); ERROR!!!

    ((B*)p2)->F2(); // вивід – B::F2
}
```

Через вказівник p2, що вказує на об'єкт типу B, з двох функцій F1, які є в класі B викликається успадкована функція. Зверніть, також, увагу, що через вказівник p2 не можна викликати функцію F2, оскільки вказівник має тип A, а такої функції немає в класі A. Виклик стає можливим лише після приведення типу вказівника до B.

6.5 Конструктор похідного класу

Очевидно, що конструктор похідного класу відповідає за ініціалізацію об'єкта похідного класу. Згадаємо, що об'єкт похідного класу складається з двох частин : успадкованої та власної. Природньо, що за ініціалізацію власних елементів має відповідати безпосередньо конструктор похідного класу. Також природньо, що відповідальність за ініціалізацію успадкованих елементів має покладатись на конструктор базового класу. Отже, будь-який конструктор похідного класу має здійснити виклик конструктора базового класу. Цей виклик здійснюється неявно і завжди перед виконанням тіла конструктора похідного класу.

В загальному випадку повністю неявно може бути викликаний лише конструктор за замовчуванням. При виклику конструктора за замовчуванням базового класу маємо два недоліки:

1) Конструктор за замовчування проініціалізує успадковану частину якимись значеннями за замовчуванням. І може виникнути потреба в тілі конструктора похідного класу переприсвоїти успадкованим елементам якісь інші значення. Цей недолік полягає в зайвій роботі, яку виконає конструктор базового класу.

2) В базовому класі може бути відсутній конструктор за замовчуванням. В цьому випадку конструктор похідного класу не зможе виконатись.

Для подолання вказаних недоліків слід забезпечити можливість виклику конструктора з аргументами, знайшовши, при цьому, спосіб задати значення цих аргументів. Це здійснюється через ініціалізатор конструктора похідного класу. Відповідна синтаксична конструкція має вигляд:

ім'я_похідного_класу::ім'я_похідного_класу(аргументи_конструктора_похідного_класу) : ім'я_базового_класу (фактичні_параметри_конструктора_базового_класу)

Приклад 6.6

```
class Point
{
public:
   Point(double x, double y);
private:
   double _x;
   double _y;
};

Point::Point(double x, double y)
{
   _x = x;
   _y = y;
}

class Circle : public Point
{
public:
   Circle_(double x, double y,double radius);
private:
   double _radius;
};

Circle::Circle_(double x, double y, double radius) : Point(x,y)
{
   _radius = radius;
}
```

6.6 Деструктор похідного класу

Ресурси, що використовуються об'єктом похідного класу, так само як і дані, можна поділити на дві категорії: власні, успадковані. Звільнення власних ресурсів, які створені в конструкторі похідного класу, має здійснювати деструктор

похідного класу. Відповідальність за звільнення ресурсів, які створені та якими оперує успадкована частина, покладається на деструктор базового класу. Відповідно при виклику деструктора похідного класу здійснюється виклик деструктора базового класу. Але зауважимо, що порядок виклику деструкторів є зворотній в порівняння із викликом конструкторів.

6.7 Непряме спадкування

Непрямим називається спадкування, коли по відношенню до деякого похідного класу деякий інший не виступає в якості безпосереднього базового класу, але в той же час з'являється на одному з попередніх рівнів ієрархії. Наприклад, якщо маємо клас A і від нього походить клас B, а від нього – клас C, то має місце непряме спадкування класу A класом C. Клас A успадковується класом C через клас B. Фактично утворюється деяка багаторівнева ієрархія спадкування із загальною вершиною.

A -> B -> C

Приклад 10.7

```
class A
{
public:
   int i;
};

class B : public A
{
public:
   int j;
};

class C : public B
{
public:
   int k;
};
```

6.8 Множинне спадкування

Множинне спадкування - це спадкування, при якому похідний клас утворюється на основі декількох базових класів. При чому всі базові класи вказуються в прототипі через кому разом із типом спадкування.

Структура об’єкту при множинному спадкуванні передбачає успадковані та власні елементи.

Об’єкт похідного класу буде включати успадковані елементи та елементи базового класу. Спочатку в пам’яті розташовані елементи базового класу, потім власні. При чому порядок успадкованих елементів визначається порядком вказання базових класів.

6.9 Комбінація непрямого та множинного спадкування

При комбінації непрямого та множинного спадкування може виникнути ситуація з дублюванням елементів.

Розглянемо приклад. Нехай є базовий клас A. І нехай є похідні класи (клас B і клас C), у яких спільний предок A. Нехай клас D утворений від класів B і C. Має місце непряме спадкування класу A класом D. При цьому клас A успадковується класом D двічі, через клас B і через клас C. Відповідно в класі D, елементи класу A будуть дублюватись.

A
A -> B
A -> C
B , C -> D – множинне спадкування

Приклад 10.8

```
class A
{
public:
   int i;
   A() {i = 0;};
};

class B : public A
{
public:
   int j;
};

class C : public A
{
public:
   int k;
};
```

```
class D : public B, public C
{
public:
   int m;
   D() {B::i = 11;};
   void Print() { printf("Bi = % d; Ci = %d", B::i, C::i); };
};

int main()
{

   D d;
   d.Print();
   d.i = 10; // Не є однозначним. Треба d.B::i = 10 чи d.C::i = 10
}
```

Отже, в наведеному прикладі, якщо ми звертаємось до елемента і успадкованого від класу А, виникає двозначність, бо незрозуміло до якого з двох успадкованих елементів ми звертаємось. Для вирішення такої проблеми використовується віртуальне спадкування.

Віртуальний базовий клас може бути об’явлений за допомогою ключового слова virtual.

На початку об’єкту замість успадкованих елементів базового віртуального класу розміщується вказівник на них. Самі ж успадковані елементи розміщуються в іншому місці. Відповідно, якщо класи утворені за допомогою віртуального спадкування будуть базовими при множинному спадкуванні, то при наявності у них спільного предка його елементи увійдуть в одному екземплярі шляхом узгодження відповідних успадкованих вказівників.

Приклад 6.9

```
class A
{
public:
   int i;
   A() {i = 0;};
};

class B : virtual public A
{
public:
   int j;
};
```

```
class C : virtual public A
{
public:
   int k;
};

class D : public B, public C
{
public:
   int m;
   D() {B::i = 11;};
   void print() { printf("Bi = % d; Ci = %d", B::i, C::i); };
};

int main()
{
   D d;
   d.print();
   d.i = 10;
}
```

Розміщення в пам’яті елементів класу D:

- Без віртуального спадкування: *A::i B::j A::i C::k D::m*
- З віртуальним спадкуванням: *p B::j p C::k D::m ... A::i*

  Вказівник p вказує на A::i

6.10 Контрольні запитання

1. Що таке спадкування та для чого воно використовується?
2. Яким є синтаксис оголошення похідного класу?
3. Чим визначається набір елементів похідного класу? На які дві категорії можна поділити ці елементи?
4. До яких елементів похідного класу мають доступ функції похідного класу?
5. До яких елементів об’єкту похідного класу мають доступ зовнішні функції?
6. Що таке перевизначення елементів? Як звернутись до успадкованого елемента базового класу у випадку перевизначення?
7. На об’єкти яких класів може вказувати вказівник базового типу?

8. До яких елементів об’єкту можна доступитись через вказівник базового типу?
9. В чому полягають особливості роботи конструктора похідного класу? Які особливості це накладає на реалізацію цього конструктора?
10. Як необхідно реалізувати конструктор похідного класу, якщо в базовому класі немає конструктора за замовченням?
11. В чому полягають особливості роботи деструктора похідного класу?
12. Що таке непряме спадкування?
13. Що таке множинне спадкування?
14. Які особливості можуть мати місце при комбінації непрямого та множинного спадкування? Що таке віртуальне спадкування?

Тема 7. Поліморфізм

Поліморфізм дозволяє забезпечити різну поведінку одного й того ж програмного коду, залежно від того об’єкт якого класу в ньому використовується. Він базується на спеціальному виді функцій класу, що називаються віртуальними функціями.

Почнемо з демонстрації прикладу, що ілюструє поліморфізм в дії.

Приклад 7.1

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

class Figure // клас Фігура
{
public:
   virtual double Square();
};

double Figure::Square()
{
   return 0.0;
}

class Rectangle : public Figure
{
public:
   Rectangle(double _width, double _height);
   double Square();
private:
   double _width;
   double _height;

};

Rectangle::Rectangle(double width, double height)
{
   _width = width;
   _height = height;
}

double Rectangle::Square()
{
   return _width * _height;
```

```
}


class Circle : public Figure
{
public:
   Circle(double radius);
   double Square();
private:
   double _radius;
};

Circle::Circle(double radius)
{
   _radius = radius;
}

double Circle::Square()
{
   return 3.14159*_radius*_radius;
}

class Point
{
public:
   Point(double x, double y);
private:
   double _x;
   double _y;
};

Point::Point(double x, double y)
{
   _x = x;
   _y = y;
}

class Circle_: public Point
{
public:
   Circle_(double x, double y, double radius);
private:
   double _radius;
};
```

```
Circle_::Circle_(double x, double y, double radius) : Point(x,y)
{
   _radius = radius;
}

int main()
{
   Figure* figures[] =
   {
      new Rectangle(5.0,10.0),
      new Circle(50.0)
   };

   double sum = 0.0;
   for (int i = 0; i < 2; i++)
   {
      sum += figures[i] -> Square();
   }
   std::cout  <<  sum;
}
```

Класи Rectangle та Circle походять від спільного базового класу Figure, тому фігури різних типів можна об'єднати в одному масиві вказівників типу Figure. Функція Square, що обчислює площу, є віртуальною, що забезпечує виклик реалізацій цієї функції з класу відповідного об'єкта, коли ми проходимо по масиву вказівників. В цьому і проявляється поліморфізм: через вказівник одного й того ж типу викликаються функції з різних класів.

7.1 Віртуальні функції

Віртуальні функції є основою поліморфізму. Вони об'являються додаванням ключового слова virtual до прототипу функції. Функція, яка об'явлена віртуальною в деякому класі, залишається віртуальною при перевизначенні в похідних класах без явного вказання ключового слова virtual. Спосіб виклику віртуальних функцій відмінний від виклику звичайних функцій. Для віртуальних функцій виклик визначається типом об'єкта, на який вказує вказівник. Поліморфний виклик при цьому реалізується через вказівник базового типу.

Приклад 7.2

```
#include <iostream>
class A
{
```

```
public:
  void F1();
  virtual void F2();
};

void A::F1()
{
   std::cout << "A::F1\n";
}

void A::F2()
{
   std::cout << "A::F2\n";
}

class B : public A
{
public:
  void F1();
  void F2();
};

void B::F1()
{
   std::cout << "B::F1\n";
}

void B::F2()
{
   std::cout << "B::F2\n";
}

int main()
{
   A* p;
   A a;
   B b;

   p = &a;
   p->F1(); // вивід – A::F1
   p->F2(); // вивід – A::F2

   p = &b;
   p->F1(); // вивід – A::F1
```

```
    p->F2(); // вивід – B::F2

    ((B*)p)->F1(); // вивід – B::F1
}
```

Зверніть увагу на різницю у виклику функцій F1 та F2 через вказівник p, що вказує на об’єкт типу B (виділено жирним). Функція F1 – не віртуальна, тому, виходячи з типу вказівника, викликається функція з класу A. Функція ж F2 – віртуальна, тому виклик визначається типом об’єкту.

Отже, для звичайних (невіртуальних) функцій виклик визначається, у разі перевизначення, типом об’єкта або типом вказівника, через який цей виклик здійснюється. Для віртуальних функцій виклик, що здійснюється через вказівник базового типу, визначається типом об’єкта, на який вказує цей вказівник.

7.2 Механізм поліморфізму

При виклику функції відбувається зв’язування цього виклику з відповідним кодом.

Для звичайних функцій тип вказівника або об’єкта, через який здійснюється виклик, відомий на етапі компіляції. Тому і зв’язування відбувається на етапі компіляції. Таке зв’язування називається раннім або статичним.

Для віртуальних функцій код чи адреса тієї функції, що буде викликана, визначається на основі типу об’єкта. Тип об’єкта, на який вказує вказівник, стає відомим під час виконання програми. Тому і зв’язування здійснюється під час виконання програми. Подібне зв’язування називається пізнім або динамічним.

Динамічне зв’язування реалізується через таблицю віртуальних функцій. Таблиця віртуальних функцій містить адреси віртуальних функцій для даного класу. При динамічному зв’язування з таблиці береться адреса функції. І з цією адресою пов’язується виклик функції.

Нехай клас A має дві віртуальні функції F2 та F3, а в класі B перевизначається лише функція F2. Таблиці віртуальних функцій для класів A та B будуть мати такий вигляд:

```
class A
A::F2
A::F3
class B
B::F2
A::F3
```

Зауважте, що на місті функції F3 в таблиці віртуальних функцій класу B буде адреса успадкованої з класу A функції.

Кожен об’єкт зберігає адресу таблиці віртуальних функцій, як правило, на початку.

7.3 Чисто віртуальні функції

Чисто віртуальні функції об’являються в базових класах для того, щоб перевизначатись в похідних. При цьому в базових класах змістовна реалізація таких функцій не можлива (прикладом є функція площі з класу фігури). Отже, чисто віртуальна функція – це віртуальна функція, що не має і не потребує реалізації. В кінці її прототипу в об’явлені класу використовується конструкція «=0».

Фактично чисто віртуальні функції заявляють певну функціональність, що має бути реалізованою в похідному класу. Клас, в якому є хоча б одна чисто віртуальна функція називається абстрактним. Не можна створювати об’єкти абстрактних класів. Абстрактні класи призначені для створення похідних класів на їх основі. В похідному класі мають бути перевизначені всі успадковані чисто віртуальні функції. Якщо хоча б одна чисто віртуальна функція не буде перевизначена, вона залишиться такою в похідному класі, який також автоматично стане абстрактним.

Приклад 7.3

```
class AbstrSample
{
public:
        virtual void F() = 0;
};

class Derived : public  AbstrSample
{
public:
        void F()
};

void Derived::F()
{
        std::cout << "Derived::F\n";
}

int main()
```

```
{
	Derived d;
	AbstrSample* p = &d;
	p -> F();
}
```

7.4 Приклад використання поліморфізму. Запис об’єктів в файл

Поліморфізм використовується при створені універсальних програмних алгоритмів, які потребують виконання в окремі моменти деякого специфічного (неуніверсального) коду. Виконання специфічного коду забезпечується в так званих точках входу у вигляді виклику віртуальної функції.

Алгоритм запису об'єктів в файл складається з чотирьох простих кроків:

1) відкриття файлу
2) перевірка дескриптора, яка дозволяє переконатись, що файл успішно відкрито
3) запис об’єкта в файл з заданим дескриптором
4) закриття файлу

Перший, другий та четвертий кроки є універсальними, що не залежать від класу об'єкта, який записується. Третій крок є специфічним, бо залежить від структури об'єкту. І це заважає нам написати універсальний алгоритм (реалізувати алгоритм в універсальний спосіб). Втім, ми можемо це зробити, якщо реалізуємо третій крок в якості точки входу для коду, що записує об'єкт в файл. І цю точку входу ми реалізуємо шляхом виклику віртуальної функції. Відповідно, ми маємо створити деякий абстрактний базовий клас, що об'являє функціональність, пов'язану з записом даних об'єкту у відкритий файл. Класи об'єктів, що будуть записуватись в файл, ми будемо утворювати як похідні, реалізуючи в них відповідну чисту віртуальну функцію.

Приклад 7.4

```
#define _CRT_SECURE_NO_WARNINGS
#include <io.h>
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <windows.h>
#include <fcntl.h>
```

```
class WritableObject
{
public:
	int Write(const char* filename);
private:
//чисто віртуальна функція, реалізувати в похідному класі
	virtual int WriteData(int h) = 0;
};

int WritableObject::Write(const char* filename)
{
	int h = _open(filename, O_WRONLY, 0);	// дія 1
	if (h == -1)				// дія 2
	{
		return 0;
	}

	if (!WriteData(h)) // //дія 3: точка входу для специфічного коду
	{
		_close(h);			// дія 4
		return 0;
	}

	close(h);				// дія 4
	return 1;
}

class Complex : public WritableObject
{
public:
	Complex(double x, double y);
	int WriteData(int h);
private:
	double _x;
	double _y;
};

Complex::Complex(double x, double y)
{
	_x = x;
	_y = y;
}

int Complex::WriteData(int h)
```

```
{
	if (_write(h, &_x, sizeof(double)) == -1)
	{
		return 0;
	}
	if (_write(h, &_y, sizeof(double)) == -1)
	{
		return 0;
	}
	return 1;

	//write(h,this,sizeof(Complex)); // Це альтернаивний варіант, коли об’єкт пишеться в файл відразу весь, а не окремими властивостями
}

class Person : public WritableObject
{
public:
	Person(const char* name);
	int WriteData(int h);
private:
	char* _name;
};

Person::Person(const char* name)
{
	_name = new char[strlen(name) + 1];
	strcpy(_name, name);
}

int Person::WriteData(int h)
{
	if (_write(h, _name, strlen(_name) + 1) == -1)
	{
		return 0;
	}
	else
	{
		return 1;
	}
}

int main()
{
```

```
    Complex c(0.0, 2.0);
    c.Write("C:\\tmp0986\\a.txt");
    Person p("Petro");
    p.Write("C:\\tmp0986\\a.txt");
}
```

7.5 Приклад використання поліморфізму. Інверсія залежності

Нехай нам потрібно реалізувати вивід об'єктів певного класу, наприклад *Complex*. Очевидно, це можна зробити, написавши відповідну функцію в класі *Complex*. Але, якщо нам знадобиться змінити спосіб виводу, доведеться переписувати функцію, тобто змінювати клас. До того ж, за потреби змінювати спосіб виводу динамічно доведеться передбачити відповідні гілки у функції виводу. Додавання нового способу виводу знову веде до необхідності змінювати клас. Основою проблеми є прив'язка до конкретної низькорівневої реалізації. Для вирішення цієї проблеми можна делегувати вивід окремому компоненту, що реалізується відповідними класами. Для кожного способу виводу створюється свій клас виводу. При такому підході клас *Complex* буде зберігати вказівник на об'єкт потрібного класу та викликати функцію виводу через нього. І цей вказівник можна змінити у будь-який момент, змінивши, таким чином, спосіб виводу.

Кожен клас виводу повинен реалізовувати функцію виводу. Оголосити таку функціональність можна у абстрактному класі, наприклад, класі *PrintObject* з чисто віртуальною функцією *Print*. Класи виводу будуть похідними від *PrintObject,* отже будуть зобов'язані реалізувати функцію *Print.* Оскільки функція *Print* не належатиме класу *Complex*, то, щоб здійснити вивід, вона має отримати об'єкт цього класу через аргумент. В конкретному випадку цей аргумент міг би мати тип *Complex*. Але вивід є загальною функціональністю, тому є логічним для віртуальної функції *Print* використати такий тип аргументу, який дозволив би передати їй об'єкт будь-якого класу. Універсальним типом аргументу може бути вказівник деякого загального базового типу. Причому від самого базового класу нічого не вимагається, треба тільки, щоб всі класи, для яких ми вирішуємо задачу виводу, від нього походили. Ми можемо, наприклад, об'явити пустий клас *Object* і клас *Complex* зробити похідним від нього. Відповідно, функція *Print* буде мати аргумент типу *Object**.

Тепер функція *Print* класу *Complex* не буде реалізовувати вивід сама, вона буде делегувати його певному об'єкту класу, похідного від *PrintObject*, викликаючи функцію *Print* через вказівник на такий об'єкт. І цей вказівник має саме базовий тип *PrintObject*, тобто виклик буде поліморфним. В якості

аргументу функції *Print* слід передати вказівник на поточний об’єкт класу *Complex*. А оскільки сам зазначений поліморфний виклик здійснюється у функції класу *Complex*, очевидно, що цим вказівником буде *this*.

Вивід буде можливим лише за умови, що для об’єкту класу *Complex* визначений вказівник на об’єкт класу виводу. Тому в класі *Complex* слід передбачити деяку функцію *SetPrintObject* для встановлення такого вказівника. До виклику цієї функції спосіб виводу буде невизначений. Можна передбачити певний стандартний вивід за замовчуванням, якщо зазначений вище вказівник не встановлений. В свою чергу, для виявлення такої ситуації, треба, щоб вказівник мав початкове значення 0, що забезпечується конструктором.

Коли якась функціональність виноситься в окремий компонент, з’являється залежність від нього. Зміна функціонального інтерфейсу цього компонента може вимагати змін компонентів, що її використовують. В описаному прикладі таку небезпеку усунуто, бо інтерфейс, за яким відбувається взаємодія з компонентом, **визначений** в абстрактному класі та є **обов’язковим** для реалізації компонентом, оскільки його клас походить від цього абстрактного класу. Таким чином, згаданий компонент залежить від інтерфейсу абстрактного класу, тобто залежність **інвертовано**. Саме тому розглянутий підхід називається інверсією залежності, що становить один з так званих SOLID-принципів ООП.

Приклад 7.5

```
#include <iostream>

class Object
{

};

class PrintObject
{
public:
   virtual void Print(Object* pObject) = 0;
};

class Complex: public Object
{
public:
```

```
    Complex(double x, double y);
    void Print();
    double GetX();
    double GetY();
    void SetPrintObject(PrintObject*);
private:
    double _x;
    double _y;
    PrintObject* _pPrintObject;
};

Complex::Complex(double x, double y)
{
    _x = x;
    _y = y;
    _pPrintObject = 0;
}

void Complex::Print()
{
    if (_pPrintObject)
    {
        _pPrintObject->Print(this);
    }
    else
    {
        std::cout << "(" << _x << "," << _y << ")";
    }
}

double Complex::GetX()
{
    return _x;
}

double Complex::GetY()
{
    return _y;
}

void Complex::SetPrintObject(PrintObject* p)
```

```
{
   _pPrintObject = p;
}

class PrintComplexSimple1 : public PrintObject
{
public:
   void Print(Object* pObj);
};

void  PrintComplexSimple1::Print(Object* pObj)
{
   Complex* pValue = (Complex*)pObj;
   std::cout << "(" << pValue->GetX() << "," << pValue->GetY() << ")";
}

class PrintComplexSimple2 : public PrintObject
{
public:
   void Print(Object* pObj);
};

void  PrintComplexSimple2::Print(Object* pObj)
{
   Complex* pValue = (Complex*)pObj;
   std::cout << pValue->GetX() << (pValue->GetY() >= 0 ? "+" : "") << pValue-
>GetY() << "*i";
}

int main()
{
   Complex c1(1.0, 2.0);

   PrintObject* pPrint = (PrintObject*)new PrintComplexSimple1();

   c1.SetPrintObject(pPrint);

   c1.Print();

   delete pPrint;
```

```
    pPrint = (PrintObject*)new PrintComplexSimple2();

    c1.SetPrintObject(pPrint);

    std::cout << std::endl;

    c1.Print();

    delete pPrint;
}
```

7.6 Приклад використання поліморфізму. Інверсія залежності (варіант 2)

Попередній приклад базувався на конкретному класі *Complex*. Але очевидно, що у такий же спосіб можна визначити вивід інформації про об'єкт для будь-якого іншого класу. При цьому значна частина програмного коду буде такою ж. Зокрема, клас завжди повинен мати вказівник на об'єкт виводу та функцію *SetPrintObject* для встановлення цього вказівника, і вона завжди буде однаковою. До того ж згаданий вказівник завжди має ініціалізуватись нулем. Якщо ми хочемо визначити вивід за замовчуванням, то можемо реалізувати це у вигляді окремої функції та викликати її у гілці, що відповідає нульовому значенню вказівника на об'єкт виводу. І тоді функція *Print* теж завжди буде однією й тією ж. Це говорить про те, що описаний підхід можна ще більш універсалізувати. Подібна універсалізація і є метою даного прикладу.

В прикладі створюється абстрактний клас *PrintableObject*, що включає весь зазначений вище універсальний код. Але цей клас не є повністю самодостатнім. Для роботи функції *Print* в ситуації з нульовим значенням вказівника на об'єкт виводу потрібна наявність функції *PrintDefault*, що реалізує спосіб виводу за замовчуванням. Саме тому в класі об'являється відповідна чисто віртуальна функція. Класи, похідні від *PrintableObject*, повинні реалізувати функцію *PrintDefault*. Вони будуть мати весь функціонал виводу, успадкувавши його від класу *PrintableObject*.

Також зникає потреба в базовому класі *Object*, бо тепер роль цього класу буде відігравати саме *PrintableObject*. Функція *Print* класу *PrintObject_m* тепер має аргумент типу *PrintableObject**.

Приклад 7.6

```
#include <iostream>

class PrintableObject;

class PrintObject_m
{
public:
   virtual void Print(PrintableObject* pObject) = 0;
};

class PrintableObject
{
public:
   PrintableObject();
   void SetPrintObject(PrintObject_m* p);
   void Print();
   void virtual PrintDefault() = 0;
private:
   PrintObject_m* _pPrintObject;
};

PrintableObject::PrintableObject()
{
   _pPrintObject = 0;
}

void PrintableObject::Print()
{
      if (_pPrintObject)
   {
      _pPrintObject->Print(this);
   }
   else
   {
         PrintDefault();
   }
}

void PrintableObject::SetPrintObject(PrintObject_m* p)
{
```

```
    _pPrintObject = p;
}

class Complex_m : public PrintableObject
{
public:
    Complex_m(double x, double y);
    double GetX();
    double GetY();
    void PrintDefault();
  private:
    double _x;
    double _y;
};

Complex_m::Complex_m(double x, double y)
{
    _x = x;
    _y = y;
 }

double Complex_m::GetX()
{
    return _x;
}

double Complex_m::GetY()
{
    return _y;
}

void Complex_m::PrintDefault()
{
    std::cout << "(" << _x << "," << _y << ")";
}

class PrintComplexSimple1_m : public PrintObject_m
{
public:
    void Print(PrintableObject* pObj);
};
```

```
void  PrintComplexSimple1_m::Print(PrintableObject* pObj)
{
   Complex_m* pValue = (Complex_m*)pObj;
   std::cout << pValue->GetX() << (pValue->GetY() >= 0 ? "+" : "") << pValue-
>GetY() << "*i";
}


int main()
{

   Complex_m c1(1.0, 2.0);

   c1.Print();

   PrintObject_m* pPrint = (PrintObject_m*)new PrintComplexSimple1_m();

   c1.SetPrintObject(pPrint);

   std::cout << std::endl;

   c1.Print();

   delete pPrint;
}
```

7.7 Контрольні запитання

1. Що таке поліморфізм? Для чого він використовується?
2. Що таке віртуальні функції? Як вони об’являються?
3. Як здійснюється поліморфний виклик?
4. На чому базується механізм поліморфізму?
5. Що таке раннє та пізнє зв’язування? Як здійснюється пізнє зв’язування, що таке таблиця віртуальних функцій?
6. Що таке чисто віртуальні функції? Як вони об’являються та для чого використовуються?

7. В чому особливості використання класів, що мають чисто віртуальні функції?
8. Як забезпечити виконання в потрібні моменти специфічного коду в універсальних програмних алгоритмах?

Тема 8. Шаблони класів

Дуже часто різні алгоритми відрізняються лише тільки типами з якими вони працюють. При цьому сам код алгоритму для різних типів виявляється однаковим. Щоб уникнути дублювання коду, яке при цьому виникає, використовуються шаблони класів. В шаблонах класів типи виступають в якості параметрів.

8.1 Оголошення шаблону класів

Шаблон класу об'являється наступним чином:

```
template <class список_параметрів_типу>
class ім'я_шаблону
{
    оголошення_елементів
};
```

Таким чином, заголовок класу містить конструкцію template<…>, де в кутових дужках через кому перелічуються параметри типу. Перед кожним з яких додається слово class.

Синтаксис заголовку шаблону класу з одним параметром типу є наступним:

template <class T>

class ім'я_шаблону

В цій конструкції T - параметр типу. Параметрів типу може бути декілька. В цьому разі параметри типу визначаються через кому :

class T1, class T2, ...

Параметри типу можуть використовуватись:

1) В тілі оголошення шаблону класу:
    a) Оголошення елементів-даних
    b) В прототипах елементів-функцій
2) В реалізації елементів-функцій.

Шаблони класів (оголошення і реалізація функцій) розміщуються в заголовочних файлах.

Синтаксис заголовка функції шаблону класу з одним параметром типу виглядає наступним чином:

template <class T>

тип_значення ім'я_шаблону<T>::ім'я_функції (список_аргументів)

Розглянемо в якості прикладу шаблон класу динамічного масиву з довільним типом елементів.

Приклад 8.1

```
template <class T>
class Array
{
public:
        Array(int N);
        T& operator[](int index);
private:
        int _N;
        T* _p;
};

template  <class T>
Array<T>::Array(int N)
{
        _N = N;
        _p = new T[_N];
}

template  <class T>
T& Array<T>::operator[](int index)
{
        return _p[index];
}
```

8.2 Використання шаблону класів

На основі шаблону класу утворюються так звані шаблонні класи шляхом підстановки в якості значень на місце параметрів типу назв конкретних існуючих типів. Шаблонні типи утворюються при першому їх згадуванні в коді, наприклад, при об‘явленні об’єкту. При цьому в файл з кодом слід включити директивою include заголовочний файл з визначенням шаблону.

Приклад 8.2

```
Array<int> a1(2);
a1[1] = 5;
std::cout << a1[1];
```

```
Array<Complex> a2(5);
Complex c(1.0, 2.0);
a[2] = c;
a2[2].Print();
```

8.3 Контрольні запитання

1. Що таке шаблони класів? Для чого вони використовуються?
2. Як об’являються шаблони класів?
3. Що таке параметри типу? Де їх можна використати в шаблоні класу?
4. В чому полягають особливості реалізації функцій в шаблонах класів?
5. Як використовуються шаблони класів? Як при цьому зазначаються значення параметрів типу?

Перелік посилань

1. Bjarne Stroustrup. The C++ Programming Language (Second Edition). Addison – Wesley, ISBN 0-201-53992-6.
2. An Introduction to Object Oriented Programming (3rd Ed), by Timothy A. Budd, published by Addison-Wesley, 2002, ISBN 0-201-76031-2

Список рекомендованої літератури

1. Bjarne Stroustrup. The C++ Programming Language (Second Edition). Addison – Wesley, ISBN 0-201-53992-6.
2. Ted Faison. Borland C++ 4.5 Object-Oriented Programming (Fourth Edition). Sams Publishing, ISBN 0-672-30605-0.
3. C++ How to Program (10th edn) by Paul Deitel & Harvey Deitel, Pearson Education, 2017, ISBN 978-0-13-444823-7
4. C++ Primer Plus (6th edn) by Stephen Prata, Pearson Education, 2012, ISBN 978-0-321-77640-2
5. Starting Out with C++ from Control Structures to Objects (8th edn) by Tony Gaddis, Pearson Education, 2015, ISBN 978-0-13-379633-9
6. Об’єктно-орієнтоване програмування: [Підручник] / В.В. Бублик. – К.: ІТ-книга, 2015. – 624 с.: іл
7. Шпак З.Я. Програмування мовою С. – Львів: Оріяна-Нова, 2006. – 432 с.
8. Карнаух, Т. О. Вступ до програмування мовою С++. Організація даних / Т. О. Карнаух, Ю. В. Коваль, М. В. Потієнко, А. Б. Ставровський. – К. : Київський університет, 2015. – 151 с.