УДК 681.3.06

М.М. Колодницький, к.т.н., проф.
Житомирський інженерно-технологічний інститут

АНАЛІЗ ПАРАДИГМИ ООП МОВОЮ С++. ("ЩО ТАКЕ КЛАСИ?")

Відомий з літератури матеріал – концепція "клас" та пов'язані з нею інші головні характерні риси об'єктно-орієнтованої мови програмування С++ – розглядається з метою проведення порівняльного аналізу (сучасної) парадигми ООП та (класичного) імперативного програмування. Наводяться мотиви виникнення концепції ОПП та пояснюються деякі особливості її реалізації в мові С++, нечітке розуміння яких може завести до певних пасток чи викликати труднощі у використанні С++. Запропоноване стисле викладення цих рис ООП в С++ може бути корисним для глибшого розуміння об'єктної парадигми при її вивченні та ефективному використанні в наукових дослідженнях чи інженерній діяльності.

Вступ

Хоча мова об'єктно-орієнтованого програмування С++ виникла не так давно, все ж вона вже має певну свою історію [1] та практику застосування [2]. Зважаючи на це, виникає справедливе запитання: чи є тема розгляду концепції "клас" та пов'язаних з нею інших ідей об'єктно-орієнтованого програмування (ООП), реалізованих в мові С++, актуальною на сьогодні? Особливо гостро це запитання постає у навчальному процесі, де воно може бути переформульоване у трохи іншому вигляді: чи потрібно вивчати та застосовувати ще одну нову мову програмування та нову концепцію – парадигму ООП, якщо той чи інший інженер, студент чи учень вже володіє якоюсь іншою мовою програмування, скажімо, С? І якщо потрібно, то: 1) які переваги це надасть? 2) як вивчити цю нову мову якнайшвидше та якнайкраще? 3) чи можливо для цього використати порівняльний аналіз нової та попередньої мов? 4) і, врешті-решт, яким же є (лаконічний) перелік нових засобів (ідей, ключових концепцій) нової мови і чи може бути він використаний у майбутньому? Для відповіді на ці запитання достатньо звернутися до існуючих літературних джерел: підручників з ООП та С++ чи до комп'ютерної документації. Безперечно, класичним джерелом є книга самого автора мови С++ – Bjarne Stroustrup [2]. І чи віднайдемо ми відповіді там на поставлені вище запитання? На жаль – ні! Ні, хоча б тому, що згадана книга (третє видання) має більше 1000 сторінок і таке викладення аж ніяк не можна назвати лаконічним. Книга, безперечно, містить повне викладення матеріалу, але для початківця розібратися в ньому не так легко. Виявляється, що не так легко знайти відповіді на такі запитання і у великій масі інших джерел. Отже, питання про порівняльне вивчення мови С++ та концепцій ООП, тобто вивчення ООП у порівнянні з відомими імперативними методами програмування, таке питання є ще й досі актуальним. Це питання, в свою чергу, зводиться до питання "що таке клас?". Виясненню відповідей на ці питання і присвячена дана робота.

Аналіз

Головна мета введення поняття "клас" полягає не стільки в створенні нових типів даних, визначених користувачем, (user defined data types) – цього можна досягти використанням звичайних "структур" (struct), – тобто не в створенні нових типів даних у мові, а також не в створенні нових структур керування (while, switch тощо), тобто головна ідея класів націлена не стільки на ті основні елементи мови, на які ми звикли звертати увагу в першу чергу при вивченні нової мови (структури даних і структури керування), а головною ідеєю введення "класів" є *модульність* програм. Модульність на зразок тієї, яку ми досягаємо, розбиваючи проект спершу на процедури, а потім й на різні файли, об'єднавши в одному файлі групу "логічно зв'язаних" процедур, що, при тому, обробляють певний – "логічно зв'язаний" – набір даних, наприклад, стек. Однак при розбиванні на файли, згідно з синтаксисом мови С, існує, в першу чергу, проблема з глобальними даними і "обмеженням видимості" даних (scope). Якщо змінна з'являється всередині функції (певного файла) – вона "локальна" (має "local scope") із усіма властивостями локальних змінних ("local scope") – її не видно (вона недоступна) ззовні, для неї не виділяється пам'ять, доки вона (змінна) не буде проініціалізована. Якщо вона з'являється поза функцією, але всередині файла, що містить функцію, то вона видна для усіх функцій цього файла, але не видна для функцій з інших файлів (у неї область видимості і час життя "файлові"). Якщо змінна "глобальна" – вона видна всім, але ідея глобальних змінних погана і суперечить ідеї "модульності". До проблеми "видимості" (scope) при реалізації розбивки великої програми на модулі додаються проблеми з "часом життя" змінної, її ініціалізацією, лінковкою (особливо для функцій). Тому потрібно було мати якийсь засіб у мові програмування, який надавав би можливість створювати багатомодульні програми,

вирішуючи ці проблеми з "видимістю" та "областю доступу" (scope) даних, їх часом життя, динамічним виділенням (керуванням) пам'яті, ініціалізацією, статичною/динамічною лінковкою. Варіанти вирішення цієї проблеми існували в різних мовах до появи C++. Автор C++ Страуструп запропонував своє вирішення цієї проблеми і використав поняття класу. За висловом James O. Coplien [3], ідея класів реалізує "Object Inversion", а саме: раніше багато програм мали або бібліотеки, або модулі, організовані навколо "структур даних". Прикладом цього може бути простий стек. Дані для стека можуть бути представлені у вигляді структури "struct". Далі розробляється кілька процедур або функцій, що "обробляють" цю структуру і, власне кажучи, є сильно "прив'язаними" до неї. При виклику цих функцій дана структура, чи її частина, передається як параметри в кожну (!) з цих функцій. В класах же відбувається все ніби навпаки (inverse): не дані передаються у функції, а функції "належать" даним. Таким чином, вирішується проблема "видимості" (scope) даних та їх "спільне використання" (share) між різними модулями програми. Отже:

1. Клас має два (навіть три) різних види доступу до даних-членів класу:
   - public (як у структури);
   - private.

2.1. Змінна ("об'єкт" чи "екземпляр") класового типу оголошується як і змінна типу "структура". Тобто з оголошенням цієї змінної – цього об'єкта – в програмі реалізується ідея "типів, визначених користувачем" (*user-defined types*), а точніше –змінних такого типу. Але такі "нові типи" могли б бути реалізованими і звичайними структурами. Якщо ми до таких структур додамо (визначимо) ще і процедури, то отримаємо те, що прийнято було називати "абстрактний тип даних" (*abstract data type*). Але тоді ще залишилося невирішеним таке питання: як зробити так, щоб було легко оголошувати змінні (тобто об'єкти) такого типу, ініціалізувати їх, вирішувати питання з їх областю видимості тощо. Для вирішення всіх цих питань і було введене поняття "класу". Декларацію класу переважно (див. також нижче "*nested classes*") потрібно робити в області тексту програми, де декларуються глобальні змінні чи вводяться нові типи (тобто вище функції "main()"), тому що декларація класу – це, власне, введення до розгляду нового виду "типу даних" (на зразок int, float тощо); це новий user defined type. Крім того, (декларація) клас(у) містить декларацію, а потім і дефініцію своїх функцій-членів, включаючи власний конструктор (який є звичайною функцією, хоча яка і не повертає значення). І хоча, в принципі, клас може бути оголошений всередині main() чи будь-якої іншої функції, це оголошення, по суті, повинне містити відразу дефініції (а не декларації) функцій. При цьому звернемо увагу на той факт, що за синтаксисом мови C не можна *визначати* функцію всередині тіла іншої функції (хоча можна оголошувати); функції можна лише *викликати* з тіла іншої функції з будь-якою глибиною вкладеності таких викликів. Зате можна робити дефініцію функції всередині класу, що декларований всередині іншої функції, аналогічно до декларації некласових user defined типів, таких як typedef, enum, struct. (Декларація стає тепер дефініцією.) Їх можна оголошувати й всередині будь-якої функції (тобто "розмазувати" по всьому тексту програми). Оголошення ж змінних (тобто об'єктів) "класового типу" можна робити як у глобальній області, так і в локальній.

2.2. При цьому, якщо ми декларуємо класи в різних файлах (модулях), вони від цього не перестають бути глобальними новими типами даних.

2.3. Public члени глобальних *об'єктів* класового типу є глобальними змінними, подібно до глобальних структур (точніше, змінних "структурного" типу).

3. При оголошенні різних об'єктів того самого класового типу можна робити різну їх ініціалізацію (так само, як і у випадку змінних "структурного" типу). Але при цьому різниця між "класом" і "структурою" полягає у наступному:

– в класах для ініціалізації їхніх об'єктів використовується поняття "конструктор" (costructor);

– конструкторів може бути кілька різних видів;

– різні конструктори можуть містити різний список параметрів (для ініціалізації різних даних-членів); дані-члени, що не ввійшли в цей список, ініціалізуються нулем (0) за default'ом (хоча насправді це так, якщо класовий об'єкт оголошений як глобальна змінна – тобто в області видимості глобальних (статичних) об'єктів; тоді дані ініціюються нулем, але у зв'язку з іншим правилом: глобальні статичні змінні завжди ініціюються нулем (сегмент BSS)). Якщо об'єкт класового типу оголосити як локальну змінну, навіть якщо сам тип (тобто клас) продекларовано в області глобальних змінних (точніше, області описів типів, у цьому випадку), то для такого локального об'єкта класового типу ніякої ініціалізації нулем не буде!!!;

– існує default constructor, що або не має параметрів, або кожен параметр має значення за замовчуванням; default constructor ініціалізує дані-члени нулем (0), але, знову ж таки, лише для глобальних об'єктів!;

– параметри в списку конструктора можуть містити присвоювання значень – ініціалізацію за замовчуванням (by default); для такої ініціалізації даних-членів існує спеціальний (невдалий, на мою думку) синтаксис;

– таким чином, конструктор – це функція, а це біль прогресивно, ніж просто "список" ініціалізації для структур чи масивів;

– якщо в класі немає даних-членів покажчиків (pointer), то конструктор, по суті, виконує лише роль "ініціалізатора" початкових значень даних-членів класу;

– початкові значення можуть також копіюватися з іншого, раніше створеного, об'єкта;

4. Якщо оголошений "об'єкт" є покажчиком на "об'єкт класового типу", то замість функції malloc() використовується нове ключове слово мови – "new". Конструктор при цьому містить текст, в якому виділяється пам'ять для динамічних даних-членів (покажчиків).

5. Функції-члени класу декларуються в області "декларації класу" (тобто між відкриваючою "{" та закриваючою "}" фігурними дужками класу), але визначаються поза цими дужками (можливо, навіть в іншому файлі). Це, в свою чергу, значить, що поняття "видимості" даних поширюється також і на "видимість" функцій-членів класу та доступність їх для інших функцій чи класів. Якщо функція визначається "усередині декларації класу", то вона називається inline-функцією. (За допомогою ключового слова inline можна "примусити" функцію стати такою. Це означає, що в асемблерній реалізації компілятором вона буде реалізована як inline-функція, тобто як код тільки "тіла" функції, а не як виклик функції, (асемблерне) тіло якої зберігається за певною адресою і виклик якої вимагає додаткової спеціальної реалізації так званих "префіксів-суфіксів", тобто подання у стек і вилучення із стека певних даних). Всі ці операції призводять лише до додаткових витрат часу і, таким чином, зниження ефективності програми.

6. Об'єкти класового типу можна присвоювати один одному (якщо вони не містять даних-членів, що є покажчиками, – оскільки це небезпечно і неправильно!). У випадку, коли класи містять покажчики, створюється два спеціальних види конструктора:

Примечание [m1]: (см. стр. 104, 117, гл 8, с. 301)

– конструктор *копіювання*;

– конструктор *присвоювання*.

7. Об'єкти класового типу можуть бути параметрами функції чи значеннями, що повертаються (тобто так само, як і структури).

8. Можна будувати масиви об'єктів класового типу (на зразок масивів структур).

9. Доступ до public даних-членів об'єктів класового типу здійснюється як і для структур, доступ до private – тільки через функції-члени класу.

10. Існує також спеціальна конструкція friend: функції "чужого" класу можуть мати доступ до private даних зазначеного класу.

11. На основі одного класу можна побудувати інший – *спадковий* або, як кажуть також, *похідний* (*derived*) клас. Це називається *спадкуванням*, хоча насправді – це звичайні відношення "належності" ("включеності") між підмножиною (base class) і надмножиною (derived class). Тут також вводиться третій вид доступу – protected. Можна сказати, що базовий клас є типом-членом класу.

Причому, потрібно чітко розуміти відмінність між композицією класів (агрегуванням об'єктів), вкладеними (nested) класами і спадкуванням (відношенням "бути підмножиною"). Клас може бути задекларований в межах іншого класу. Такий клас називається "*вкладеним*" (nested class). Вкладені класи розглядаються як такі, що створюються в межах базового класу і можуть бути використані в цих межах. Для того, щоб посилатися на вкладений клас поза межами класу, наприклад, B, що містить клас, наприклад, A, необхідно використовувати його повне ім’я, наприклад, B::A. Вкладені класи декларують лише тип в межах класу. Вони не спричиняють створення об’єктів типу "вкладений клас", тобто об’єктів, які були б вкладеними елементами зовнішнього (контейнерного) класу. Тобто вкладений клас є типом, що є підмножиною типу-класу, де він описаний. Базовий клас (через механізм спадковості) також є підмножиною похідного класу, але область його дії не класова (class scope), а глобальна (global scope) з усіма випливаючими відмінностями в правилах створення об'єктів цих типів-класів. Композитний клас (точніше – композитний (чи агрегатний) об'єкт) містить як підмножину дані-члени різних типів об'єктів, породжених від різних (не базових!) класів. (При цьому, компілятор Visual C++ 7.0, Beta-2 не дозволяє створювати конструктори з параметрами ініціалізації для об'єктів-членів композиційних класів; дозволяється лише ініціалізація вбудованих (інтегральних) типів даних).

12. Ініціалізація даних-членів базового класу здійснюється за тим же принципом і синтаксисом, що і для даних-членів похідного класу.

13. Клас може містити функції з однаковими іменами, але різним списком параметрів (класичний приклад – конструктори). Такі функції називаються overloaded (перезавантаженими, тобто з перевантаженням, а точніше – заміною імен. "Стандартні" оператори, такі як "=", "+" тощо також бути можуть overloaded.

14. Крім того, базовий та похідний класи можуть містити функції з однаковими іменами і списком параметрів. Такі функції називаються *virtual*. Насправді, суть полягає в способі лінковки цих функцій. Якщо лінковка статична – такі функції називають *overloaded* (але для того, щоб її здійснити, список параметрів у функцій повинний бути різним). Якщо замість *статичної* лінковки (іноді кажуть також

"зв'язування" – binding) використовується *динамічне* зв'язування, тоді (невдале, на мою думку) слово virtual, по суті, означає використання "покажчиків на функцію" при звертанні до цих функцій. Це, в свою чергу, призводить до того, що в тексті програми (на design-time) записується те саме ім'я функції (з префіксом – ім'ям об'єкта визначеного класу, а точніше – покажчиком на об'єкт), а в run-time викликається "придатна" функція (або базового, або похідного класів), і це визначається (обчислюється) на підставі значень покажчика-префікса даної функції (а точніше, покажчика на об'єкт (класового типу) і покажчика на функцію цього об'єкта). Допоміжна інформація зберігається в допоміжній таблиці vftbl. Така технологія названа (невдалим, на мою думку) словом – *поліморфізм*.

15. Віртуальну функцію, що не має реалізації в базовому класі, а лише декларує (структуру) інтерфейс(а), красноомовний Страуструп (чи хто ще до нього?) назвав *суто* віртуальною. Її оголошення має спеціальний синтаксис " = 0".

16. Якщо клас має хоча б одну *суто* віртуальну функцію, його називають *абстрактним*. (Якщо базовий клас є абстрактним, то, крім vftbl, створюється також vbtbl.) Не можна створити об'єкт абстрактного класу. (Воно і зрозуміло: принаймні одна його функція (тобто віртуальна функція) не має реалізації; що це за об'єкт із невизначеною поведінкою?) При цьому можна використовувати покажчик чи посилання на такий клас (тобто можна створити об'єкт, тип якого є "покажчик на абстрактний клас"; у такого об'єкта його зміст є визначеним – це адреса, у попередньому випадку – зміст є невизначеним).

17. Конструкція template – це параметризований тип (клас). Використовується спеціальний синтаксис для роботи з template.

18. Як ми бачимо, ідея класів у C++ не містить ніякої додаткової ідеї, пов'язаної з *подіями*, і значить, ООП, саме по собі, не включає поняття подій та їхню обробку в класах, як це подекуди неправильно викладається в літературі деякими авторами (які не розуміють того, про що вони пишуть). Задача обробки подій є областю застосування (*application*) для класів (тобто для ООП), а класи є *технологією* для вирішення таких задач. Реалізація застосування цієї технології для обробки подій здійснюється, як правило, із залученням третьої сторони – операційної системи чи ще якогось модуля, який є "головним менеджером" чи "диспетчером" і керує (handle) всім складним (інтерактивним і "паралельным" (concurrent)) процесом обробки подій. Отже (ще раз), класи C++ як такі ніякого відношення до подій не мають. У Java [7], Visual Basic, Visual FoxPro або C# ("Сі шарп") [8], на відміну від C++, вже є поняття подій, бо визначається більш широкою областю застосування таких мов програмування чи інтегральних систем розробки.

Висновки

Таким чином, розглянувши головні характерні риси об'єктно-орієнтованого програмування мовою C++ у порівнянні з базовою мовою C, автор гадає, що таке порівняння може бути корисним для більш глибокого розуміння об'єктної парадигми. Автор проводив подібний порівняльний аналіз мов програмування та стисле викладення деяких рис ООП і в своїх попередніх роботах [4, 5]. Проте тема порівняльного аналізу концепцій мов, а також глибокого їх розуміння (на основі такого аналізу) і на сьогодні не перестає бути актуальною. Так, наприклад, основні ідеї сучасної концепції Component Object Model, COM та побудованих на її основі технологій ActiveX/OLE/ [6], є свого роду продовженням розвитку об'єктної парадигми. На сьогодні більшість системного та прикладного програмного забезпечення такої компанії, як Microsoft, побудовано з використанням ActiveX/OLE/COM. Цей факт говорить сам за себе. Отже, краще розуміння ідей "класів", "об'єктів" тощо надасть розробникам програмного забезпечення незрівняно більших досягнень у використанні ActiveX-технологій.

Іншим прикладом корисності порівняльного вивчення об'єктної парадигми є (знову ж таки) нові мови програмування, такі як Java [7], та C# ("Сі шарп") [8]. Проведенний в даній роботі аналіз може також бути виконаним за анологією і для цих "нових" мов програмування.

Отже, автор вважає, що викладений вище матеріал може бути корисним читачам при освоєнні найновітніших софтверних технологій.

ЛІТЕРАТУРА:

1. *Stroustrup B. A* history of C++: 1979–1991 // The second ACM SIGPLAN conference on History of programming languages, 1993. – Pp. 271–297.
2. *Stroustrup B.* The C++ programming language. – Reading, Mass.: Addison-Wesley, 2000. – 1019 p.
3. *Coplien J.O.* Advanced C++ programming styles and idioms. – Reading, Mass.: Addison-Wesley, 1992. – 520 p.
4. *Колодницкий Н.М.* Порівняльне вивчення мов програмування (на прикладах Turbo Pascal та Borland C++) // Праці Житомирського філіалу КПІ. Серія А. Техніка. – Житомир: ЖФ КПІ, 1994. – Вип. 2. – С. 236–261.

5. *Колодницький М.М.* Технічне та програмне забезпечення комп'ютерних інформаційних технологій. – Житомир: ЖІТІ, 1995. – 231 с.
6. *Колодницький М.М.* Елементи теорії САПР складних систем. – Житомир: ЖІТІ, 1999. – 512 с.
7. The Java™ Language Specification. Second Edition. / James Gosling, Bill Joy, Guy Steele and Gilad Bracha. Addison-Wesley. 1996. – 532 p.
8. C# Language Specification. Version 0.28, 5/29/2002.

КОЛОДНИЦЬКИЙ Микола Михайлович — кандидат технічних наук, професор кафедри комп'ютерних інтегрованих систем Житомирського інженерно-технологічного інституту.

Наукові інтереси:
- математичне моделювання систем;
- комп'ютерні інформаційні технології.

Подано 13.03.2002

УДК 681.3.06
Колодницький М.М. Аналіз парадигми ООП мовою С++. ("Що таке класи?")
Колодницкий Н.М. Анализ парадигмы ООП с использованием языка С++. ("Что такое классы?")
Kolodnytsky M.M. The Analysis of OOP Paradigm with C++ /

УДК 681.3.06
Анализ парадигмы ООП с использованием языка С++. ("Что такое классы?") / Н.М. Колодницкий

Известний с литературы материал – концепция "класс" и связанные с ней другие главные черты объектно-ориентированного языка программирования С++ – рассматриваются с целью проведения сравнительного анализа (современной) парадигмы ООП и (классического) императивного программирования. Приводятся мотиви возникновения концепции ОПП и объясняются некоторые особенности ее реализации в языке С++, нечеткое понимание которых может привести к некоторым ловушкам или вызвать трудности при использовании С++. Предложенное сжатое изложение этих черт ООП в С++ может быть полезным для более глубокого понимания объектной парадигмы при ее изучении и эффективном использовании в научных исследованиях или инженерной деятельности.

UDK 681.3.06
THE ANALYSIS OF OOP PARADIGM WITH C++ / M.M. Kolodnytsky

The facts well known from literature – the concept of "class" and some related features of object-oriented programming with C++ – are considered for sake of comparative analysis of (modern) OOP paradigm and (classical) imperative programming. The motives of development of OOP concepts are described and then some peculiarities of its implementation are explained. Those peculiarities should be understood very well; otherwise it can cause some traps or difficulties while programming with C++. The laconic presentation of the C++ OOP features is worth to use for more deep understanding of the OOP paradigm during the teaching of C++ programming or in research and software engineering.