**Лекція 19.** **Поліморфізм та віртуальні функції. Базові поняття та основні принципи**

**Відношення між класами та об’єктами**

* Класи і об'єкти тісно пов'язані поняття. Будь-який об'єкт належить деякому класу. Клас породжує будь-яке число об'єктів.
* Класи статичні (всі їх особливості і зміст визначаються в процесі компіляції). Об'єкти динамічні, тобто створюються і знищуються в процесі виконання програми.
* На етапі аналізу і ранніх стадіях проектування вирішується дві основні задачі:
  + виявлення класів і об'єктів, що складають словник предметної області;
  + побудова структур, що забезпечують взаємодію об'єктів, при яких виконуються вимоги задачі.

***Пам’ятаємо про ці базові поняття*.**

**Проблеми при перевизначенні методів з різною поведінкою в базовому та похідному класі**

При спадкуванні часто виникає потреба в тому, щоб поведінка деяких методів базового класу та похідного класу різнилися. Очевидним рішенням є перевизначення відповідних методів в похідному класі, але тут виникає проблема, яку ілюструє такий приклад.

**#include <iostream.h>**

**#include <Windows.h>**

**using namespace std;**

**class Base // Базовий клас**

**{ public:**

**int f(const int &d) // Метод базового класу**

**{cout << "Base f" <<endl;**

**return 2\*d; }**

**int CallFunction(const int &d)**

**{ cout << "Base CallFunction" <<endl;**

**// Передбачається виклик базового класу**

**return f(d)+1;**

**}**

**};**

**class Derived: public Base // Похідний клас**

**{ public: // CallFunction спадкується**

**int f(const int &d) // функція f перевизначається**

**{ cout << "Derived f" <<endl;**

**return d\*d;**

**}**

**};**

**int main()**

**{ system("color F0");**

**Base a; // об’єкт базового класу**

**cout << a.CallFunction(5)<< endl; // отримуємо 11**

**Derived b; // об’єкт похідного класу**

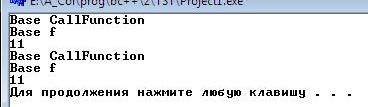
**cout << b.CallFunction(5)<< endl; // який метод f викликається?**

**system("pause");**

**return 0;**

**}**

Результат:



В базовому класі визначені два метода — **f()** и **CallFunction()**, Зауважимо, що в другому методі викликається перший. В класі-нащадку метод **f()** перевизначений, а метод **CallFunction()** успадкований. Метод **f()** перевизначено, щоб об’єкти базового класу та класу - нащадка вели себе по-різному. Визначаючи об’єкт b типу Derived, програміст очікує отримати результат **5 \* 5 + 1 = 26** — для цього і перевизначається метод **f()**. Але на консоль, як і для об’єкта а типу Base, виводиться число 11, яке обчислюється як **2 \* 5 + 1 = 11**. Незважаючи на перевизначення методу **f()** в класі-нащадку, в успадкованій функції **CallFunction()** викликається «рідна» функція **f()**, визначена в базовому класі!

Аналогична проблема виникає і в при підстановці посилання або покажчика на об’єкт похідного класу замість посилання або покажчика на об’єкт базового. Розглянемо приклад з часами и будильником.

**#include <iostream.h>**

**#include <Windows.h>**

**using namespace std;**

**class Clock // базовий клас - часи**

**{ public:**

**void print() const**

**{ cout << "Clock!" << endl; }**

**};**

**class Alarm: public Clock // похідний клас - будильник**

**{ public:**

**void print() const // перевизначеий метод**

**{ cout << "Alarm!" << endl; }**

**};**

**void settime(Clock &d) // функція установки часу**

**{ d.print(); } // чекаємо виклик методу базового класу**

**int main()**

**{ system("color F0");**

**Clock W; // об’єкт базового класу**

**settime(W); // виводиться "Clock"**

**Alarm U; // об’єкт похідного класу**

**settime(U); // посилання на похідний замість базового**

**Clock \*c1 = &W; // адрес об’єкта базового класу**

**c1->print(); // виклик базового метода**

**c1 = &U; // адрес об’єкта похідного типу замість базового**

**c1->print(); // який метод викликається, базовий чи похідний?**

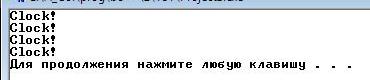
**// ((Alarm \*)c1)->print(); // "зайві" дужки потрібні!**

**system("pause");**

**return 0;**

**}**

Результат:

****

Знову в класі-нащадку перевизначимо метод, щоб забезпечити різну поведінку об’єктів базового та похідного класів. Але при передачі параметра по посиланню базового класу в функцію settime() та при явному виклику метода print() через покажчик базового класу спостерігаємо ту ж картину: завжди викликається метод базового класу, хоч бажали викликати метод похідного.

**Зв’язування**

Цю ситуацію пояснює зв’язування. **Зв’язування — це співставлення виклику функції з тілом.**

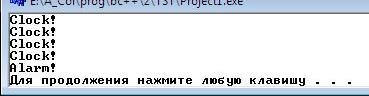
В наданих прикладах **зв’язування виконується на етапі трансляції** (до запуска) програми. Таке зв’язування називають раннім або **статичним**.

При трансляції класу Base (перший приклад) компілятор нічого не знає про класи-нащадки , тому він не може передбачити перевизначення методу f()в класі Derived. Його природньою поведінкою буде «міцно» зв’язати виклик f() с тілом метода класу Base. Аналогічно при трансляції функції settime() компілятору нічого не відомо про тип об’єкту, що реально передається під час виконання програми. Тому виклик метода print() зв’язується з тілом метода базового класу Clock, як і визначено в заголовку функції settime(). Точно так же покажчик на базовий клас «міцно» зв’язується з методом базового класу під час трансляції.

При виклику метода по покажчику в цьому конкретному випадку ми можемо викликати метод похідного класу, задав явне перетворення покажчика:

((Alarm \*)c1)->print(); // "зайві" дужки потрібні!

Якщо розкоментуємо передостанній оператор попереднього отримаємо результат:

****

Але для функції settime() та метода CallFunction() такий виклик зробити неможливо, оскільки нам потрібна саме різна поведінка в залежності від типу об’єкту. Окрім того, якщо такий виклик прописаний в середині функції, яка приймає покажчик як параметр (наприклад, settime(Clock \*c1)), то ми маємо ті ж самі проблеми.

**Визначення віртуальних функцій.**

В С++ існує механізм, за допомогою якого можна узнати тип об’єкта під час виконання програми. Цей механізм в С++ називається динамічною ідентифікацією типів (Run-Time Type Identificator– RTTI). Але в ситуаціях, подібних розглянутим, застосовується інший більш «сильний» та елегантний механізм С++ — механізм віртуальних функцій.

Щоб отримати різну поведінку в залежності від типу, необхідно оголосити функцію-метод віртуальною; в С++ це робиться за допомогою ключового слова **virtual**. Тоді в першому прикладі оголошення метода f() в базовому та похідному класі повинно бути таким:

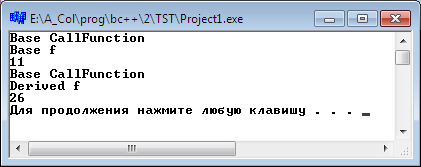
virtual int f(const int &d) // в базовому класі

{ return 2\*d; }

virtual int f(const int &d) // в похідному класі

{ return d\*d; }

Після цього для об’єктів базового та похідного класів отримуємо різні результати:



Аналогічно, в другому прикладі оголошення метода print() теж повинно починатися зі слова virtual:

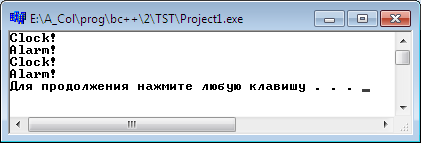
virtual void print() const // в базовому класі

{ cout << "Clock!" << endl; }

virtual void print() const // в похідному класі

{ cout << "Alarm!" << endl; }

Після цього виклик settime() с параметром базового класу забезпечує виведення на консоль слова «Clock», а з параметром похідного класу — слова «Alarm». При виклику по покажчику буде така ж картина.



Ключове слово virtual можна написати тільки один раз — в оголошенні функції базового класу, але краще завжди оголошення робити явним, щоб завжди за текстом було видно, що функція віртуальна.

Віртуальна функція (**virtual**) оголошується в базовому класі і може бути перевизначена у похідних класах. Сукупність класів, в яких визначається і перевизначається віртуальна функція, називається **поліморфним кластером**. У межах цього кластеру об'єкт пов'язується із конкретною віртуальною функцією-членом під час виконання програми. Звичайна функція-член також може бути перевизначена у похідному класі, як у попередньому прикладі. Проте без атрибуту **virtual** до неї буде застосоване лише раннє зв'язування.

**Механізм віртуальних функцій**

На етапі компіляції програми, якщо у описі методів зустрічається ключове словo virtual для даного об’єкту автоматично створюється таблиця віртуальних методів (Virtual Methods Table – VMT)

Ця таблиця розміщується у ініціалізованій частині сегменту даних і виглядає так:



Об'єкт класу розміщається в суцільній області пам'яті, адреса якої зберігається в неявному вказівнику this. При виклику звичайної функції-члена цей вказівник передається їй як додатковий аргумент. Створюючи об'єкт похідного класу, компілятор поєднує його поля і поля базового класу в одне ціле. Якщо базовий клас містить віртуальні функції, їх адреси заносяться в таблицю віртуальних функцій. Усі класи, що утворюють поліморфічний кластер, містять вказівник на цю таблицю, у якій зберігаються вказівники на усі віртуальні функції-члени класів.

Таким чином, для віртуальних функцій забезпечується не статичне, а динамічне (відкладене) зв’язування, яке реалізується під час виконання програми. Александреску вказує, що в С++ реалізовані два типи поліморфізму:

* статичний поліморфізм, або поліморфізм часу компіляції (compile-time polymorphism), що реалізується за рахунок перевантаження та шаблонів функцій;
* динамічний поліморфізм, або поліморфізм часу виконання (run-time polymorphism), що реалізується віртуальними функціями.

З перевантаженням функцій «розбирається» компілятор, правильно підбираючи варіант функції в тій чи іншій ситуації. Вибір здійснюється статично. Вибір же віртуальної функції здійснюється с динамічно — при виконанні програми. Клас, який включає віртуальні функції, називається *поліморфним*.

**Віртуальний деструктор**

Якщо вказівник базового класу посилається на об'єкт похідного класу, необхідно застосовувати віртуальний деструктор. Розглянемо конкретний приклад.

**#include <stdio.h>**

**#include <string.h>**

**#include <Windows.h>**

**using namespace std;**

**class TBase**

**{**

**public:**

**char\* a;**

**TBase(const char\* s){a = strdup(s); printf("Ctor TBase\n");}**

**~TBase(){delete a; printf("Dtor TBase\n");}**

**//virtual ~TBase(){delete a; printf("Dtor TBase\n");}**

**virtual void print() {printf("TBase::a = %s\n",a);}**

**};**

**class TDerived: public TBase**

**{**

**public:**

**char\* b;**

**TDerived(const char\* s1, const char\* s2):TBase(s2)**

**{ b = strdup(s2); printf("Ctor TDerived\n");}**

**~TDerived(){delete b; printf("Dtor TDerived\n");}**

**void print() {printf("TDerived::b = %s\n",b);}**

**};**

**class TDerived2: public TDerived**

**{**

**public:**

**char\* c;**

**TDerived2(const char\* s1,const char\* s2, const char\* s3):TDerived(s2,s3)**

**{c = strdup(s3); printf("Ctor TDerived2\n");}**

**~TDerived2(){delete c; printf("Dtor TDerived2\n");}**

**//virtual ~TDerived2(){delete b; printf("Dtor TDerived2\n");}**

**void print() {printf("TDerived2::c = %s \n",c);}**

**};**

**int main()**

**{ system("color F0");**

**TBase\* pObj = new TDerived2("TBase", "TDerived","TDerived2");**

**delete pObj; // Помилка!**

**system("pause");**

**return 0;**

**}**

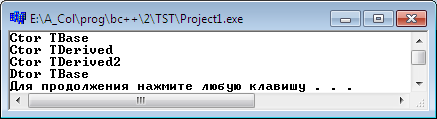
Клас TBase містить один рядок. Його конструктор виділяє для неї стільки пам'яті, скільки займає константний аргумент s, а його деструктор звільняє пам'ять, зайняту рядком a. Клас TDerived успадковує цей рядок із класу TBase і додає свою. Його конструктор спочатку викликає конструктор класу TBase, ініціалізуючи успадковане поле a, а потім виділяє пам'ять для власного рядка b, ініціалізуючи її константним аргументом.

Клас TDerived2 успадковує цей рядок із класу TDerived і додає свою. Його конструктор спочатку викликає конструктор класу TDerived, ініціалізуючи успадковані поля a і b, а потім виділяє пам'ять для власного рядка c, ініціалізуючи її константним аргументом.

Всі деструктори видаляють лише поля, виділені для власних членів класу, думаючи, що за успадковані поля відповідальність несе деструктор базового класу.

Уся ця конструкція працює цілком надійно, поки програма не спробує знищити вказівник базового класу pObj, що посилається на об'єкт похідного класу; у даному випадку вказівник посилається на об'єкт objDerived2 класи TDerived2.

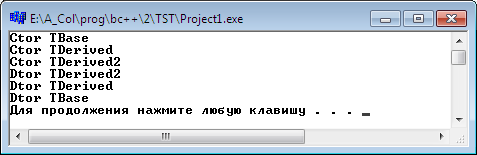
Програма виводить на консоль:



Як бачимо, пам'ять звільнена не цілком. Що відбулося? Деструктор звільнить пам'ять, виділену конструктором базового класу, але для звільнення пам'яті, зайнятої додатковими полями, необхідно викликати деструктори похідних класів! У звичайного деструктора немає такої можливості. Для цього його оголошують віртуальним.

В тексті програми віртуальні деструктори класів TBase і TDerived (вони обоє є базовими для класів TDerived і TDerived2 відповідно, закоментовані. Розкоментуємо їх та закоментуємо не віртуальні деструктори.

Це приведе до наступного результата.



Тепер пам'ять звільнена коректно.

**Правила опису та використання віртуальних функцій-методів**:

1. Віртуальна функція може бути тільки методом класу.

2. Будь-яку перевантажену операцію-метод класу можна зробити віртуальною, наприклад, операцію присвоювання або операцію перетворення типу.

3. Віртуальна функція, як і сама віртуальність, спадкується.

4. Віртуальна функція може бути константною.

5. Якщо в базовому класі визначена віртуальна функція, то метод похідного класу з таким же самим ім’ям та прототипом (включно з типом значення, що повертається, та константністю метода) автоматично стає віртуальним (слово virtual вказувати не обов’язково) та заміщує функцію-метод базового класу.

6. Конструктори не можуть бути віртуальними.

7. Статичні методи не можуть бути віртуальними.

8. Деструктори можуть (частіше — повинні) бути віртуальними — це гарантує коректне повернення пам’яті через покажчик базового класу.

9. Якщо клас містить хоч одну віртуальну функцію, деструктор класу теж слід зробити віртуальним.

10. Віртуальна функція не може бути дружньою.

Майерс визначає це такі важливі правила.

[Правило 7: Об’являйте деструктори віртуальними в поліморфному базовому класі.](http://cpp.com.ru/meyers_1/ch2.html)

[Правило 9: Ніколи не викликайте віртуальні функції в конструкторі або деструкторі.](http://cpp.com.ru/meyers_1/ch2.html)

[Правило 35: Розглядайте альтернативи віртуальним функціям.](http://cpp.com.ru/meyers_1/ch6.html)

[Правило 36: Ніколи не перевизначайте успадковані невіртуальні функції.](http://cpp.com.ru/meyers_1/ch6.html)

Дейтел щодо віртуальних функцій дає такі настанови щодо техніки програмування (ТП), типових помилок програмування (ТПП) та хорошого стилю програмування (ХСП).

* Цікавим наслідком використання віртуальних функцій та поліморфізму є те, що програми отримують простіший вигляд (ТП 10.1).
* Якщо функція один раз була оголошена віртуальною, то вона залишається віртуальною на будь-якому нижчому рівні ієрархічної структури (ТП 10.2).
* Не зважаючи на те, що деякі функції можуть бути неявно віртуальними, оскільки вони оголошені такими на більш високому рівні ієрархії, деякі програмісти віддають перевагу явному оголошенню функції на кожному рівні ієрархії для забезпечення ясності програми (ХСП 10.1).
* Якщо в похідному класі прийнято рішення не описувати віртуальну функцію, то похідний клас безпосередньо успадковує опис віртуальної функції з базового класу
* Перевизначена віртуальна функція повинна мати той же самий тип значення, що повертається, та ту ж саму сигнатуру, що і віртуальна функція базового класу (ТПº10.3).
* Якщо в похідному класі перевизначається віртуальна функція базового класу і ця функція не має тип значення, що повертається, та ту ж саму сигнатуру, що і відповідна функція базового класу, то виникає синтаксична помилка (ТПП 10.1).

**Якість об’єктів**

Якість класів і об'єктів визначається критеріями:

* зчеплення - міра глибини зв'язків між окремими модулями або класами
* суперечність: треба прагнути до мінімального зачеплення, але успадкування передбачає сильне зачеплення.

**Довідково. Адреса перевантаженої функції**

**Приклад.** Нехай задано 3 “перевантажені” функції Increment() для типів int, double, char. При оголошенні покажчика на функцію, потрібно явно задати тип покажчика при оголошенні.

Нижченаведений код демонструє використання покажчика на перевантажену функцію

#include "stdafx.h"

#include <iostream>

using namespace std;

int Increment(int i)

{

return i+1;

}

double Increment(double d)

{

return d+1;

}

char Increment(char c)

{

return c+1;

}

int \_tmain(int argc, \_TCHAR\* argv[])

{

int (\*pi)(int); // покажчик на функцію, яка отримує параметром int і повертає int

pi = Increment; // p вказує на int Increment(int)

int d = (\*pi)(4); // d = 5

char (\*pc)(char);

pc = Increment; // p вказує на char Increment(char)

char c = (\*pc)('F'); // c = 'G'

cout << c << endl;

return 0;

}

*Для самостійного вивчення*: Поглибити матеріал лекції за наданою літературою. Вивчення лекційного матеріалу та додаткових джерел. Розгляд запитань і виконання завдань для самостійної роботи, запропонованих на лекції.

*Література*

1. Джейс Либерти Освой самостоятельно С++ за 21 день: 3-е изд. пер. с англ.: Уч. пос. – М.: Издательский дом „Вильямс”, 2001. – 816 с.: ил..
2. Павловская Т.А. С/С++. Программирование на языке высокого уровня. СПб.: Питер, 2003. – 461 с. URL:  <http://www.ph4s.ru/bookprogramir_1.html>
3. Липпман С. Б., Лажойе Ж. Язык программирования С++: Вводный курс. — М.: ДМК, 2001. URL: <http://www.insycom.ru/html/metodmat/inf/Lipman.pdf>
4. Дейтел Х., Дейтел П. Основы программирования на С++. – М.: Бином, 1999. – 1024 с. URL:  <http://ijevanlib.ysu.am/wp-content/uploads/2018/03/deytel.pdf>
5. Бадд Т. Объектно-ориентированное программирование в действии. [2-е изд.] – СПб.: Изд-во "Питер". 1997.  URL: <http://khizha.dp.ua/library/Timothy_Budd_-_Introduction_to_OOP_(ru).pdf>.
6. *Герб Саттер,Андрей Александреску* "Стандарты программирования на С++ ", [*Вильямс, 2005*](http://www.williamspublishing.com/)*; 304 с.*
7. *Скотт Мейерс.* Эффективное использование C++. 50 рекомендаций по улучшению ваших программ и проектов. *"ДМК", 2000; 240 с.*
8. *Скотт Мейерс.* Наиболее эффективное использование C++. 35 новых рекомендаций по улучшению ваших программ и проектов.*"ДМК",2000;304 с.*

*Контрольні запитання*.

1. Який метод називається віртуальним?
2. Які синтаксичні й семантичні особливості віртуального методу.?
3. Чи успадковуються віртуальні функції?
4. Чи можуть конструктор і деструктор бути віртуальними?
5. Що таке зв'язування? Яка програма здійснює зв'язування викликів невіртуальних функцій?
6. Чим статичне зв'язування відрізняється від динамічного?
7. Як впливають віртуальні функції на розмір класу?
8. Навіщо потрібні віртуальні функції?
9. Чим об'єкт класу, що має віртуальні функції, відрізняється від об'єкта класу без віртуальних функцій?
10. Чи можна успадковувати суто віртуальну функцію?
11. Чи можна оголосити деструктор як суто віртуальний?

*Контрольні запитання для надання письмових відповідей*.

1. Похідні класи мають з базовим класом зв'язки двох видів***.*** Які саме?
2. Чому віртуальна функція не може бути дружньою?
3. Розгляньте наданий нижче приклад і поясніть, чому в тесті визначена операція присвоювання як неможлива. Відповідь проілюструйте власним прикладом.

**int main (void)**

**{**

**Base b;**

**SubBase sb;**

**// екземпляр базового класу створюється як похідний**

**Base bb = SubBase ();**

**// екземпляру базового класу присвоюється похідний**

**b = sb;**

**// посилання на базовий клас посилається на похідний**

**Base & bbb = sb;**

**// вказівник на базовий клас вказує на похідний**

**Base \*p = &sb;**

**// sb = b; // таке присвоєння неможливе!**

**return 0; }**