ВІРТУАЛЬНІ ФУНКЦІЇ ТА ПОЛІМОРФІЗМ

## Поняття про поліморфізм

*Поліморфізм* є однією з трьох основних властивостей ООП.

Під ***поліморфізмом*** розуміють механізм реалізації методів, при якому методу з одним і тим же ім'ям може відповідати різний програмний код (поліморфний код) в залежності від того, об'єкт якого класу використовується при виклику даного методу. Наприклад, клас транспортних засобів може мати узагальнений метод руху. У класах-нащадках цей метод може бути конкретизований: автомобіль буде їздити, літак - літати, корабель - плавати. Тобто, метод з одним і тим же ім'ям може виконувати різні дії залежно від контексту, зокрема, від приналежності до того чи іншого класу.

З цієї причини поліморфізм іноді характеризують фразою "один інтерфейс, багато методів". Це означає, що до всіх однойменних методів класу можна отримати доступ одним і тим самим способом, незважаючи на можливу відмінність у конкретних діях кожного із методів.

У різних мовах програмування поліморфізм реалізується різними способами. Зокрема, у С++ підтримується два його різновиди: *простий поліморфізм*, що базується на механізмі *раннього зв'язування*, і *динамічний поліморфізм*, що використовує механізм *пізнього* *зв'язування*.

## Простий поліморфізм

При компіляції програми для коду звичайної функції (не покажчика на функцію) виділяється пам'ять і призначаються адреси для кожного її оператора. Перша адреса у визначенні є адресою функції. При виклику функції, процесор переходить на цю адресу і починає виконувати тіло функції. Найважливішим тут є те, що адреса функції призначається під час компіляції програми, і саме ця адреса використовується в подальшому при виклику функції.

Такий механізм виклику функції називають *раннім* (або *статичним*) *зв'язуванням.* Раннє зв'язування означає, що вся інформація, необхідна для виклику функції, відома при компілюванні програми.

Прикладами раннього зв'язування також можуть слугувати виклики стандартних функцій і виклики перевизначених функцій (звичайних і операторних).

Згадаємо приклад побудови класів TLong і TReal з використанням спадкування.

typedef unsigned long dlong;

class TLong // клас Ціле число

{ public:

dlong num;

TLong (){}

TLong (dlong an):num(an){}

~ TLong (){}

void print(void) { cout<<" Целое число : "<<num<<endl; }

void setnum(dlong an) { num=an; }

};

class TReal: public TLong

{ public:

dlong drob;

char \*real;

TReal (){}

TReal (char \*st) : TLong (){ setnumv(st); }

~ TReal () { delete real; }

void printr();

void setnumv(char \* st);

};

void TReal::setnumv(char \* st)

{ int l=strlen(st); char \*ptr;

real=new char[l+1]; strcpy(real,st);

ptr=strchr(real,'.'); \*ptr='\0';

drob=dlong(atol(ptr+1));

num=dlong(atol(real)); \*ptr='.';

}

void TReal::printr()

{ cout<<"Вещественное число: "<<real<<endl;

cout<<"Целая часть: "; print();

cout<<"Дробная часть: "<<drob<<endl;

}

void main ()

{ setlocale(0,"russian");

TReal a("456789.1234321"),

\*pa=new TReal ("456789.1234321"),

mask[3]= { TReal ("1748.5932"), TReal ("4567.34321"), TReal ("18689.9421") };

a.printr();

pa->printr();

delete pa;

for(int i=0;i<3;i++)

{ cout<<"Элемент массива "<<(i+1)<<": "<<endl;

mask[i].printr();

}

system("pause");

}

У цьому прикладі функції print() і printr() виконують однакові за змістом операції - виводять атрибути відповідного об'єкта (TLong або TReal) на екран. Отже, їх можна назвати одним ім'ям, наприклад print(), перевизначивши таким чином відповідний метод класу TReal. Тоді нова версія даної програми може бути представлена так:

typedef unsigned long dlong;

class TLong

{ public:

dlong num;

TLong (){}

TLong (dlong an) : num(an){}

~ TLong (){}

void print(void) { cout<<" Целое число : "<<num<<endl; }

void setnum(dlong an) { num=an; }

};

class TReal: public TLong

{ public:

dlong drob;

char \*real;

TReal (){}

TReal (char \*st) : TLong (){ setnumv(st); }

~ TReal () { delete real; }

void print();

void setnumv(char \* st);

};

void TReal::setnumv(char \* st)

{ int l; char \*ptr;

l=strlen(st); real=new char[l+1]; strcpy(real,st);

ptr=strchr(real,'.'); \*ptr='\0';

drob=dlong(atol(ptr+1));

num=dlong(atol(real));

\*ptr='.';

}

void TReal::print()

{ cout<<"Вещественное число: "<<real<<endl;

cout<<"Целая часть: "; TLong::print();

cout<<"Дробная часть: "<<drob<<endl;

}

void main()

{ setlocale(0,"russian");

TLong i(174832);

i.print();

TReal a("1748.5932");

a.print();

system("pause");

}

У даній програмі перевизначений метод print() виконує різні дії залежно від того, до якого класу він відноситься. У такому випадку кажуть, що метод print(), є поліморфним.

Окреме визначення метода в своєму класі називають *аспектом* поліморфного методу. Тобто, реалізація метода print() в класі TLong і реалізація метода print() в класі TReal – різні аспекти поліморфного методу print().

Нагадаємо, *перевантаженням методів* називають використання декількох методів з одним і тим самим іменем, але різними параметрами. Компілятор визначає, який саме метод потрібно викликати, за типом фактичних параметрів.

Оскільки при перевантаженніметодів реалізується *раннє* (*статичне*) *зв'язування* (вся інформація, необхідна для виклику методів, відома на етапі компіляції програми), то такий різновид поліморфізму називають *простим* (або *статичним*) *поліморфізмом.*

Таким чином, статичий поліморфізм підтримується мовою С++ на етапі компіляції і реалізується за допомогою механізму перевизначення (перевантаження) методів. Тому такі поліморфні методи називаються в С++ перевизначеними. У відповідності із загальними правилами перевизначення функцій такі методи повинні відрізнятися *сигнатурою*, тобто кількістю, типом і порядком перерахування переданих параметрів.

Перевантаження методів дозволяє будувати гнучкі і досконалі ієрархії класів, перевизначаючи в похідних класах методи відповідно до вимог розроблюваної програми або системи, що використовує ці класи.

Але, використання перевизначених методів не завжди безпечне. Зокрема, відомі випадки, коли при перевизначенні методів виникають помилки, пов'язані з некоректним визначенням типу об'єкта на етапі компіляції, а отже і необхідного аспекта метода, що викликається.

Для демонстрації помилок, що виникають при некоректному використанні простого поліморфізму введемо в опис базового класу попереднього прикладу новий метод show(). Цей метод буде викликати статичний поліморфний метод print() і успадковуватися в похідних класах. Крім цього, додамо в програму зовнішню функцію show\_ext() з параметром-посиланням на базовий клас, щоб показати особливості раннього зв'язування.

typedef unsigned long dlong;

class TLong

{ public:

dlong num;

TLong (){}

TLong (dlong an) : num(an){}

~ TLong (){}

void print(void) { cout<<" Целое число : "<<num<<endl; }

void setnum(dlong an) { num=an; }

**void show() { print(); }**

};

class TReal: public TLong

{ public:

dlong drob;

char \*real;

TReal (){}

TReal (char \*st) : TLong (){ setnumv(st); }

~ TReal () { delete real; }

void print();

void setnumv(char \* st);

};

void TReal::setnumv(char \* st)

{ int l; char \*ptr;

l=strlen(st); real=new char[l+1]; strcpy(real,st);

ptr=strchr(real,'.'); \*ptr='\0';

drob=dlong(atol(ptr+1));

num=dlong(atol(real));

\*ptr='.';

}

void TReal::print()

{ cout<<"Вещественное число: "<<real<<endl;

cout<<"Целая часть: "; TLong::print();

cout<<"Дробная часть: "<<drob<<endl;

}

**void show\_ext(TLong &par)**

**{ par.print(); }**

void main()

{ setlocale(0,"russian");

TLong i(174832);

i.show();

TReal a("1748.5932");

a.show(); // виводить тільки цілу частину числа (помилка!)

TReal \*pa=new TReal("456789.1234321"); // покажчик на об’єкт похідного класу

pa->print();

pa->show(); // виводить тільки цілу частину числа (помилка!)

delete pa;

TLong \*pb=new TReal("234567.34765"); // покажчик базового класу, об’єкт похідного класу

pb->print(); // виводить тільки цілу частину числа (помилка!)

delete pb; // неявно викликає деструктор класу TLong (помилка!)

show\_ext(a); // виводить тільки цілу частину числа (помилка!)

system("pause");

}

Порівняння останніх двох програм показує, що результати явного і опосередкованого викликів методу print() розрізняються. При явному виклику для змінних базового і похідного класу, а також, якщо покажчики та об'єкти збігаються за типом, ніяких проблем не виникає: викликається аспект методу класу, до якого належить об'єкт. Опосередкований же виклик може призводити до помилок. Це пояснюється наступними причинами.

По-перше, при ранньому зв'язуванні метод show() жорстко з'єднується з методом print() базового класу на етапі компіляції. Отже, з методу show() завжди буде викликатися метод print() базового класу, для якого атрибути похідних класів не доступні.

По-друге, при виконанні стандартного перетворення покажчика похідного класу в покажчик на базовий, атрибути і методи похідного класу стають невидимими для покажчика на об'єкти базового класу, тому звернення до методу print() призведе до виклику однойменного методу базового класу.

По-третє, зовнішня функція show\_ext() описана з формальним параметром-посиланням на об'єкт базового класу. За правилами синтаксису, в якості фактичного параметра при виклику такої функції їй можна передати ім'я об'єкта похідного класу. При цьому передача параметра здійснюється за адресою, яка стає відомою тільки на етапі виконання, а ув'язка всіх адрес визначається на етапі компіляції, тому реально у функції буде реалізовано звернення до методу print() базового класу.

У всіх трьох випадках об'єкт, для якого викликається метод, визначається адресою, що зберігається в покажчику, який за типом може не збігатися з типом об'єкта:

1-й випадок - *якщо наслідуваний метод для об'єкта похідного класу викликає метод, перевизначений в похідному класі* - в цьому випадку викликаючий метод визначений у базовому класі, тому компілятор вважає, що і аспект поліморфного методу необхідний базового класу, що не коректно для випадку, коли наслідуваний метод викликаний для об'єкта похідного класу.

2-й випадок - *якщо об'єкт похідного класу через покажчик базового класу звертається до методу, перевизначеному похідним класом* - в цьому випадку тип об'єкта компілятор визначає за типом покажчика, відповідно підключаючи аспект поліморфного методу базового класу, хоча реально об'єкт може належати похідному класу.

3-й випадок - *якщо процедура викликає перевизначений метод для об'єкта похідного класу, переданого в процедуру через параметр-змінну, описаний як об'єкт базового класу («процедура з поліморфним об'єктом»)* - в цьому випадку необхідний аспект поліморфного методу визначається типом параметра змінної, що не коректно, якщо в якості аргументу в процедурі передається об'єкт похідного класу.

Таким чином, використання перевизначених методів може приводити до помилок, пов'язаних з некоректним визначенням типу об'єкта на етапі компіляції.

Фіксація адреси методу *на етапі компіляції* є відмінною рисою раннього зв'язування. Однак, в перерахованих вище випадках для отримання правильного результата необхідно мати змогу вибирати необхідний аспект поліморфного методу *на етапі виконання*, коли клас об'єкта, для якого викликається метод, точно відомий. Такий механізм називають *динамічним* (або *пізнім*) *зв'язуванням*, а поліморфізм періоду виконання - *динамічним поліморфізмом.*

## Динамічний поліморфізм

### **Покажчики на похідні типи**

Основою для динамічного поліморфізму слугує покажчик на базовий клас. Покажчики на базові та похідні класи пов'язані такими відносинами, які не властиві покажчикам інших типів. Покажчик одного типу, як правило, не може вказувати на об'єкт іншого типу. Проте покажчики на об'єкти базових класів і об'єкти похідних класів – винятки з цього правила.

У C++ покажчик на базовий клас також можна використовувати для посилання на об'єкт будь-якого класу, виведеного з базового. Наприклад, у нас є базовий клас baseClass і похідний клас derivedClass, який виведено з класу baseClass. У C++ будь-який покажчик, оголошений як покажчик на базовий клас baseClass, може бути також покажчиком на похідний клас derivedClass. Отже, після цих оголошень

baseClass \*p; // створення покажчика на об'єкт базового типу

baseClass ObjB; // створення об'єкта базового типу

derivedClass ObjD; // створення об'єкта похідного типу

обидві такі присвоєння є абсолютно законними:

p = &ObjB; // покажчик p вказує на об'єкт типу baseClass

p = &ObjD; // присвоєння покажчику адреси об'єкта похідного класу derivClass

У наведеному прикладі покажчик р можна використовувати для доступу до всіх членів об'єкта ObjD, що є виведеним з об'єкта ObjB.

Як конкретний приклад, розглянемо програму, яка визначає базовий клас baseClass і похідний клас derivedClass. У цьому коді програми проста ієрархія класів використовується для зберігання імен авторів і назв їх книг.

class baseClass

{ char author[80];

public:

void putAuthor(char \*s) { strcpy(author, s); }

void showAuthor() { cout <<"Автор: "<< author << endl; }

};

class derivClass : public baseClass

{ char title[80];

public:

void putTitle(char \*n) { strcpy(title, n); }

void showTitle() { cout << "Назва: " << title << endl; }

};

int main()

{ baseClass \*bp; // створення покажчика на об'єкт базового типу

baseClass ObjB; // створення об'єкта базового типу

derivClass \*dp; // створення покажчика на об'єкт похідного типу

derivClass ObjD; // створення об'єкта похідного типу

// -------------------- Доступ до класу baseClass через покажчик -----------------------------

bp = &ObjB; // присвоєння покажчику адреси об'єкта базового класу

bp->putAuthor("Емiль Золя");

// -------------------- Доступ до класу derivClass через "базовий" покажчик --------------------

bp = &ObjD; // присвоєння покажчику адреси об'єкта похідного класу

bp->putAuthor("Вiльям Шекспiр");

ObjB.showAuthor(); // покажемо, що кожен автор належить до відповідного об'єкта

ObjD.showAuthor();

cout << endl;

/\* Оскільки функції putTitle() i showTitle() не є частиною базового класу, то вони

недоступні через "базовий" покажчик bp, i тому до них потрібно звертатися

або безпосередньо, або через покажчик на похідний тип. \*/

dp = &ObjD; // присвоєння покажчику адреси об'єкта похідного класу

dp->putTitle("Буря");

dp->showAuthor(); // тут можна використовувати або покажчик bp, або покажчик dp.

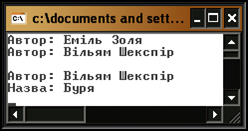
dp->showTitle();

getch();

return 0;

}

Внаслідок виконання ця програма відображає на екрані такі результати:



У наведеному прикладі покажчик bp визначається як покажчик на базовий клас baseClass. Але він може також посилатися на об'єкт похідного класу derivedClass. Однак, його можна використовувати для доступу тільки до тих елементів похідного класу, які успадковані від базового. Через "базовий" покажчик отримати доступ до тих членів, які належать похідному класу, неможливо. Ось чому до функції showTitle() звернення реалізується за допомогою покажчика dp, який є покажчиком на похідний клас.

Якщо виникає потреба за допомогою покажчика на базовий клас отримати доступ до елементів, що визначені певним похідним класом, то необхідно привести цей покажчик до типу покажчика на похідний тип. Наприклад, у процесі виконання цього рядка коду програми дійсно буде викликано функцію showTitle() об'єкта ObjD:

((derivClass \*)bp)->showTitle();

Зовнішній набір круглих дужок використовують для зв'язку операції приведення типу з покажчиком bp, а не з типом, що повертається функцією showTitle(). Незважаючи на те, що у використанні такої операції формально немає нічого некоректного, такого запису, за змогою, необхідно уникати, оскільки усі ці прийоми просто вносять у код програми плутанину.

Крім цього, необхідно розуміти: хоча "базовий" покажчик можна використовувати для доступу до об'єктів будь-якого похідного типу, зворотнє ж твердження є неправильним: використовуючи покажчик на похідний клас, не можна отримати доступ до об'єкта базового типу.

Покажчик можна інкрементувати та декрементувати щодо свого базового типу. Якщо покажчик на базовий клас використовують для доступу до об'єкта похідного типу, то механізм інкрементування або декрементування не примусить його посилатися на наступний об'єкт похідного класу. Натомість він вказуватиме на наступний об'єкт базового класу. Таким чином, інкрементування або декрементування покажчика на базовий клас необхідно розцінювати як некоректну операцію, якщо цей покажчик використовують для посилання на об'єкт похідного класу.

Той факт, що покажчик на базовий тип можна використовувати для посилання на будь-який об'єкт, який є виведеним з базового типу, надзвичайно важливий і принциповий для мови C++. Ця гнучкість є ключовим моментом для механізму реалізації динамічного поліморфізму у C++.

Слід зазначити, що подібно до покажчиків, посилання на базовий клас також можна використовувати для доступу до об'єкта похідного типу. Ця можливість особливо часто застосовується при передачі аргументів функціям. Параметр, який має тип посилання на базовий клас, може приймати об'єкти базового класу, а також об'єкти будь-якого іншого типу, виведеного з нього.

### **Поняття віртуальної функції**

При звертанні за покажчиком (або посиланням) на базовий тип до об'єктів похідного типу, на етапі компіляції не можна встановити, функція якого з похідних класів (аспект поліморфного методу) має бути викликана. В ході виконання програми потрібно перевіряти на об'єкт якого класу посилається покажчик і після такої перевірки викликати необхідну функцію. Ці дії називають *динамічним* (*пізнім*) *зв'язуванням*, на відміну від *статичного* (*раннього*) *зв'язування*, при якому вже на етапах компіляції можна встановити адресу точки входу метода, що викликається.

Тому, у даному випадку, необхідне використання *динамічного поліморфізму*, який реалізується механізмом пізнього зв'язування, що дозволяє вибирати необхідний аспект поліморфного методу вже на етапі виконання програми, коли клас об'єкта, для якого викликається метод точно відомий.

Реалізація динамічного поліморфізму базується на використанні двох засобів: механізму успадкування і так званих *віртуальних функцій*.

***Віртуальною*** називають функцію, яка оголошується в базовому класі з використанням ключового слова virtualі перевизначається в одному або декількох похідних класах. При цьому прототипи віртуальних функцій у різних класах повинні збігатися не тільки за іменами, а й по сигнатурі, хоча алгоритми, реалізовані такими функціями, різні. Якщо прототипи функцій не збігаються, то механізм віртуальності для них не включається.

Формат визначення віртуальної функції:

virtual тип\_результату ім'я\_функції (параметри)

{ тіло\_функції

}

У С++, на відміну від інших мов програмування, зокрема, Pascal, при перевизначенні віртуальної функції у похідному класі ключове слово virtual повторювати не потрібно (хоча це не буде помилкою).

При використанні віртуальних функцій потрібний аспект поліморфної функції визначається на етапі виконання, коли відомо, для якого об'єкта викликаний метод: об'єкта базового чи об'єкта похідного класу.

У цьому випадку C++ визначає, яку саме версію віртуальної функції необхідно викликати, за типом об'єкта, який адресується покажчиком. Іншими словами, саме за типом об'єкта (а не за типом самого покажчика), який адресується, визначається, яку версію віртуальної функції буде виконано.

Таким чином, якщо базовий клас містить віртуальну функцію і якщо з цього базового класу виведено два (або більше) інші класи, то при адресації різних типів об'єктів через покажчик на базовий клас виконуватимуться і різні версії віртуальної функції. Аналогічний механізм працює і при використанні посилання на базовий клас.

Розглянемо варіант виправлення помилок попереднього прикладу для класів TLong і TReal, який передбачає реалізацію механіму пізнього зв'язування. Для цього оголосимо функцію print() і деструктор класу TLong віртуальними:

**virtual** ~TLong()

{ cout<<"Виртуальный деструктор класса TLong"<<endl; }

**virtual** void print(void);

Тоді потрібний аспект поліморфної функції буде визначатися на етапі виконання і програма буде працювати вірно:

void main ()

{ setlocale(0,"russian");

TReal a("1748.5932");

a.show(); // викликає метод рrint() похідного класу

TReal \*pa=new TReal("456789.1234321"); // покажчик і об’єкт похідного класу

pa->show(); // викликає метод рrint() похідного класу

delete pa;

TLong \*pb=new TReal("234567.34765"); // покажчик базового класу, об’єкт похідного класу

pb->print(); // викликає метод рrint() похідного класу

delete pb; // викликає деструктор похідного класу

show\_ext(a); // викликає метод рrint() похідного класу

system("pause");

}

Розглянемо ще одну програму, у якій продемонстровано механізм застосування віртуальних функцій.

class baseClass

{ public:

virtual void Show() { cout << "Базовий клас" << endl; } // віртуальна функція

};

class firstD : public baseClass

{ public:

void Show() { cout<<"Перший похідний клас" <<endl; } // перевизначення Show() для firstD

};

class secondD : public baseClass

{ public:

void Show() { cout<<"Другий похідний клас" <<endl; } //перевизначення Show() для secondD

};

int main()

{ baseClass ObjB; // створення об'єкта базового типу

baseClass \*bp; // створення покажчика на об'єкт базового типу

firstD ObjF; // створення об'єкта похідного типу

secondD ObjS; // створення об'єкта похідного типу

bp = &ObjB; // присвоєння покажчику адреси об'єкта базового класу

bp->Show(); // доступ до функції Show() класу baseClass

bp = &ObjF; // присвоєння покажчику адреси об'єкта похідного класу

bp->Show(); // доступ до функції Show() класу firstD

bp = &ObjS; // присвоєння покажчику адреси об'єкта похідного класу

bp->Show(); // доступ до функції Show() класу secondD

getch();

}

Внаслідок виконання ця програма відображає на екрані такі результати:

Базовий клас

Перший похідний клас

Другий похідний клас.

Розглянемо детально код цієї програми, щоб зрозуміти, як вона працює.

У класі baseClass функція Show() оголошена віртуальною. Це означає, що її можна перевизначити у похідному класі (у класі, виведеному з baseClass). І вона дійсно перевизначається в обох похідних класах firstD і secondD. У основній функції main() оголошуються чотири змінні: ObjB – об'єкт типу baseClass, bp – покажчик на об'єкт класу baseClass, а також два об'єкти ObjF і ObjS двох похідних класів firstD і secondD відповідно. Потім покажчику bp присвоюється адреса об'єкта ObjB і викликається функція Show(). Оскільки функцію Show() оголошено віртуальною, то C++ у процесі виконання програми визначає, до якої саме версії функції Show() тут потрібно звернутися. Причому рішення приймається шляхом аналізу типу об'єкта, який адресується покажчиком bp. У цьому випадку покажчик bp вказує на об'єкт типу baseClass, тому спочатку виконується та версія функції Show(), яку оголошено у базовому класі baseClass.

Потім покажчику bp присвоюється адреса об'єкта ObjF. Оскільки, за допомогою покажчика на базовий клас можна звертатися до об'єкта будь-якого його похідного класу, то, коли функція Show() викликається вдруге, C++ знову з'ясовує тип об'єкта, який адресує покажчик bp, і, виходячи з цього типу, визначає, яку версію функції Show() потрібно викликати. Оскільки покажчик bp тепер вказує на об'єкт типу firstD, то виконується версія функції Show(), яку визначено у похідному класі firstD. Аналогічно, після присвоєння покажчику bp адреси об'єкта ObjS, викликається версія функції Show(), яку оголошено у похідному класі secondD.

Віртуальну функцію можна викликати і звичайним способом (не через покажчик), використовуючи оператор "крапка" і задаючи ім'я об'єкта, який викликається. Це означає, що у наведеному вище прикладі було б синтаксично коректно звернутися до функції Show() наступним чином:

ObjF.Show();

Проте під час виклику віртуальної функції у такий спосіб її поліморфні атрибути ігноруються - *динамічний поліморфізм досягається тільки під час звернення до віртуальної функції через покажчик на базовий клас*.

Функція, оголошена віртуальною, залишається віртуальною і у всіх спадкоємцях. Однак іноді в одному або декількох похідних класах перевизначення віртуальної функції може бути відсутнім. У цьому випадку механізм підключення віртуальної функції зберігається, а, отже, викликається функція базового класу, найближчого до розглянутого. Відсутність аспекту віртуальної функції не порушує механізму пізнього зв'язування і, якщо у спадкоємців такого класу з'явиться аспект віртуальної функції, він буде викликаний без будь-яких проблем.

Віртуальна функція обов'язково повинна бути компонентом деякого класу. Клас, який містить хоча б одну віртуальну функцію, називається *поліморфним*, і, відповідно, об'єкт такого класу теж є *поліморфним*.

Виклик віртуального метода здійснюється за допомогою так званої *таблиці віртуальних методів* (*Virtual Method Table*, VMT) - спеціальної структури в пам'яті, яка автоматично створюється під час компіляції, а потім використовується під час виконання програми. Дана таблиця генерується для кожного класу, що має хоча б один віртуальний метод, і містить адреси віртуальних методів цього класу.

Кожен об'єкт класу, що має віртуальні методи, як прихований атрибут (зазвичай, як перший член), містить покажчик на VMT-таблицю свого класу, який називають  *вказівником віртуальної таблиці* (*Virtual Table Pointer, vpointer*) або *віртуальним табличним курсором*. Цей *vpointer*-вказівник автоматично вставляється у початок конструктора класу на етапі компіляції і додається до об'єкта при його (об'єкта) ініціалізації. Саме при ініціалізації встановлюється взаємозв'язок між об'єктом і його VMT-таблицею.

При виклику віртуального метода об'єктом, місцезнаходження коду реалізації даного методу визначається за VMT-таблицею класу даного об'єкта. Механізм виклику віртуального метода наступний:

1. з об'єкта береться *vpointer*-покажчик на відповідну VMT-таблицю;
2. з VMT-таблиці, за фіксованим зсувом, вибирається адреса віртуального метода (покажчик на реалізацію метода), що використовується для даного класу;
3. здійснюється безпосередній виклик метода.

Використання віртуальних методів і перевизначених методів – це два принципово різні процеси.

Передусім, різні аспекти (версії) перевизначеного метода мають різну сигнатуру, тоді як прототипи у різних аспектах віртуального метода мають точно збігатися не тільки за іменем, а й по сигнатурі. Якщо прототипи різних аспектів віртуального метода будуть різними, то такі методи просто вважатимуться перевизначеними, а їх "віртуальна суть" втратиться.

Окрім цього, віртуальний метод повинен бути членом класу, для якого він визначається, а не його "другом". Водночас віртуальний метод може бути "другом" іншого класу.

Перевага застосування віртуальних методів полягає в тому, що при цьому використовується саме механізм пізнього зв'язування, який допускає обробку об'єктів, тип яких невідомий під час компіляції.

### **Успадкування віртуальних функцій**

Якщо функція оголошується віртуальною, то вона залишається такою незалежно від того, через скільки рівнів похідних класів вона може пройти. Наприклад, якби похідний клас secondD був виведений з похідного класу firstD, а не з базового класу baseClass, як це показано в наведеному вище прикладі, то функція Show(), як і раніше, залишалася б віртуальною, і механізм вибору відповідної версії теж працював би коректно. Тобто, наведений нижче приклад є коректним.

class secondD : public firstD // цей клас виведено з класу firstD, а не з baseClass

{ public:

void Show() { cout<<"Другий похідний клас" <<endl; } //перевизначення Show() для secondD

};

Якщо похідний клас не перевизначає віртуальну функцію, то використовується функція, яку було визначено в базовому класі. Наприклад,

class baseClass

{ public:

virtual void Show() { cout << "Базовий клас" << endl; }

};

class firstD : public baseClass

{ public:

void Show() { cout << "Перший похідний клас" << endl; }

};

class secondD : public baseClass

{ // функція Show() тут взагалі не визначена

};

int main()

{ baseClass ObjB; // створення об'єкта базового типу

baseClass \*bp; // створення покажчика на об'єкт базового типу

firstD ObjF; // створення об'єкта похідного типу

secondD ObjS; // створення об'єкта похідного типу

bp = &ObjB; // присвоєння покажчику адреси об'єкта базового класу

bp->Show(); // доступ до функції Show() класу baseClass

bp = &ObjF; // присвоєння покажчику адреси об'єкта похідного класу

bp->Show(); // доступ до функції Show() класу firstD

bp = &ObjS; // присвоєння покажчику адреси об'єкта похідного класу

bp->Show(); // звернення до Show() класу baseClass, оскільки у secondD вона не перевизначена

getch();

}

Тепер у процесі виконання цієї програми на екран виводиться таке:

Базовий клас

Перший похідний клас

Базовий клас.

Як підтверджують отримані результати виконання цієї програми, оскільки функція Show() не перевизначена у похідному класі secondD, то під час її виклику за допомогою посилання bp->Show() виконується та версія функції Show(), яку було визначено у базовому класі baseClass.

Розглянемо ще один приклад.

class baseClass

{ public:

virtual void Show() { cout << "Базовий клас" << endl; }

};

class firstD : public baseClass

{ public:

void Show() { cout << "Перший похідний клас" << endl; }

};

class secondD : public **firstD**

{ // функція Show() тут взагалі не визначена

};

int main()

{ baseClass ObjB; // створення об'єкта базового типу

baseClass \*bp; // створення покажчика на об'єкт базового типу

firstD ObjF; // створення об'єкта похідного типу

secondD ObjS; // створення об'єкта похідного типу

bp = &ObjB; // присвоєння покажчику адреси об'єкта базового класу

bp->Show(); // доступ до функції Show() класу baseClass

bp = &ObjF; // присвоєння покажчику адреси об'єкта похідного класу

bp->Show(); // доступ до функції Show() класу firstD

bp = &ObjS; // присвоєння покажчику адреси об'єкта похідного класу

bp->Show(); // звернення до функції Show() класу firstD, оскільки у secondD її не перевизначено

getch();

}

Внаслідок виконання ця програма відображає на екрані такі результати:

Базовий клас

Перший похідний клас

Перший похідний клас.

Як бачите, похідний клас secondD тепер використовує версію функції Show(), яку визначено у похідному класі firstD, оскільки вона знаходиться найближче в ієрархічному ланцюжку класів.

Таким чином, динамічний поліморфізм забезпечується тим, що в класі-нащадку змінюють реалізацію методу класу-предка з обов'язковим збереженням сигнатури методу. Це забезпечує збереження незмінного інтерфейсу класу-предка і дозволяє здійснити зв'язування імені методу в коді з різними класами - з об'єкта якого класу здійснюється виклик, з того класу і береться метод з такою назвою.

### **Потреба у застосуванні віртуальних функцій**

Як уже наголошувалося, віртуальні функції у поєднанні з похідними типами дають змогу C++ підтримувати динамічний поліморфізм. Для ООП динамічний поліморфізм є важливим з однієї причини: він дає змогу деякому узагальненому класу визначати функції, які використовуватимуть усі похідні від нього класи, причому похідний клас може визначити власну реалізацію усіх або деяких із цих функцій. Іноді ця ідея виражається так: базовий клас диктує загальний інтерфейс, який матиме будь-який об'єкт, виведений з цього класу, але при цьому дає змогу похідному класу визначити метод, який використовується для реалізації цього інтерфейсу. Ось чому для опису поліморфізму часто використовують фразу – один інтерфейс, багато методів.

Для успішного застосування поліморфізму необхідно розуміти, що базовий і похідний класи утворюють ієрархію, розвиток якої спрямований від більшого до меншого ступеня узагальнення (тобто від базового класу до похідного). У разі коректної реалізації, базовий клас забезпечує всі елементи, які похідний клас може використовувати безпосередньо. Він також визначає функції, які похідний клас повинен реалізувати самостійно. Це дає похідному класу гнучкість у визначенні власних методів, але водночас зобов'язує використовувати загальний інтерфейс.

Іншими словами, оскільки формат інтерфейсу визначається базовим класом, то будь-який похідний клас повинен розділяти цей загальний інтерфейс. Таким чином, застосування віртуальних функцій дає змогу базовому класу визначати узагальнений інтерфейс, який використовуватиметься всіма похідними класами.

Може виникнути запитання: чому такий важливий загальний інтерфейс з множиною реалізацій? Відповідь знову повертає нас до основної спонукальної причини виникнення ООП: такий інтерфейс дає змогу програмісту справлятися із наростаючою складністю програм.

Наприклад, якщо коректно розробити програму, то можна упевнитися в тому, що до всіх об'єктів, виведених з базового класу, можна буде отримати доступ єдиним (загальним для всіх) способом, незважаючи на те, що конкретні дії одного похідного класу можуть відрізнятися від дій іншого. Це означає, що програмісту доведеться пам'ятати тільки один інтерфейс, а не велику їх кількість. Окрім цього, похідний клас має можливість використовувати будь-які або всі функції, надані базовим класом. Іншими словами, розробнику похідного класу не потрібно наново винаходити елементи, які вже є в базовому класі.

Понад це, від'єднання інтерфейсу від реалізації дає змогу створювати бібліотеки класів, написанням яких можуть займатися сторонні організації. Коректно реалізовані бібліотеки повинні надавати загальний інтерфейс, який програміст може використовувати для виведення похідних класів відповідно до своїх конкретних потреб. Наприклад, як бібліотека базових класів MFC (*Microsoft Foundation Classes*), так і новіша бібліотека класів .*NET* *Framework* *Windows Forms* підтримують Windows-програмування. Використання цих класів дає змогу розробляти програми, які можуть успадкувати багато функцій, потрібних будь-якій Windows-програмі. Знадобиться тільки додати в неї засоби, унікальні для програми. Це – велика допомога при програмуванні складних ієрархічних систем.

### **Приклад застосування віртуальних функцій**

Щоб отримати уявлення про потужність механізму "один інтерфейс, багато методів", розглянемо таку програму. Вона створює базовий клас figure, призначений для зберігання розмірів різних двовимірних об'єктів і обчислення їх площ. Функція Set() є стандартним методом класу, оскільки вона підходить для всіх похідних класів. Проте функція Show() оголошена як віртуальна, оскільки методики обчислення площі різних об'єктів будуть різними. Програма використовує базовий клас figure для виведення двох спеціальних класів rectangle і triangle.

class figure

{ protected:

double x, y;

public:

void Set(double \_x, double \_y) { x = \_x; y = \_y; }

virtual void Show()

{ cout << "У цьому класі виразу для обчислення площі не визначено" << endl;

}

};

class triangle : public figure

{ public:

void Show()

{ cout << "Трикутник з висотою " << x << " i основою " << y;

cout << " має площу "<< x \* 0.5 \* y << " кв. од." << endl;

}

};

class rectangle : public figure

{ public:

void Show()

{ cout << "Прямокутник розмірами " << x << " x " << y;

cout << " має площу " << x \* y << " кв. од." << endl;

}

};

int main()

{ figure \*p; // створення покажчика на об'єкт базового типу

triangle ObjT; // створення об'єктів похідних типів

rectangle ObjR; // створення об'єкта похідного типу

p = &ObjT; // присвоєння покажчику адреси об'єкта похідного класу

p->Set(10.3, 5.5);

p->Show();

p = &ObjR; // присвоєння покажчику адреси об'єкта похідного класу

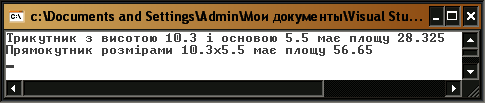
p->Set(10.3, 5.5);

p->Show();

getch();

}

Ось як виглядають результати виконання цієї програми:



У цій програмі під час роботи з класами rectangle і triangle використано однаковий інтерфейс, хоча у кожному з них реалізовано власні методики обчислення площі відповідних об'єктів.

Використовуючи оголошення базового класу figure, можна також вивести похідний клас circle для обчислення площі круга за заданим значенням радіуса. Для цього достатньо створити новий похідний тип, який би обчислював площу круга.

Потужність віртуальних функцій опирається на той факт, що програміст може легко вивести новий тип, який розділятиме загальний інтерфейс з іншими "спорідненими" об'єктами. Ось, наприклад, як це можна зробити в нашому випадку:

class circle : public figure

{ public:

void Show()

{ cout << "Круг з радіусом " << x << " має площу " << 3.14 \* x \* x << endl;

}

};

Перш ніж використати клас circle, розглянемо уважно визначення функції Show(). Зверніть увагу на те, що в ній використовується тільки одне значення змінної x, яка повинна містити радіус круга. Проте, згідно з визначенням функції Set(), у базовому класі figure їй передається два значення, а не одне. Оскільки похідному класу circle не потрібне друге значення, то є два способи вирішити цю проблему.

Перший (і одночасно найгірший) полягає у тому, щоб, працюючи з об'єктом класу circle, викликати функцію Set(), передаючи їй як другий параметр фіктивне значення. Основний недолік цього методу – відсутність чіткості у задаванні параметрів і потреба пам'ятати про спеціальні винятки, які порушують дію механізму – один інтерфейс, багато методів.

Вдаліший спосіб вирішення цієї проблеми полягає у наданні параметру функції Set() значення, яке діє за замовчуванням. У цьому випадку під час виклику функції Set() для круга потрібно задавати тільки радіус, для трикутника або прямокутника - обидва значення. Нижче показано програму, у якій реалізовано цей підхід.

class figure

{ protected:

double x, y;

public:

void Set(double \_x, double \_y**=0**) { x = \_x; y = \_y; }

virtual void Show()

{ cout << "У цьому класі виразу для обчислення площі не визначено" << endl;

}

};

class triangle : public figure

{ public:

void Show()

{ cout << "Трикутник з висотою " << x << " i основою " << y;

cout << " має площу " << x \* 0.5 \* y << " кв. од." << endl;

}

};

class rectangle : public figure

{ public:

void Show()

{ cout << "Прямокутник розмірами " << x << " x " << y;

cout << " має площу " << x \* y << " кв. од." << endl;

}

};

class circle : public figure

{ public:

void Show()

{ cout << "Круг з радiусом " << x;

cout << " має площу " << 3.14159 \* x \* x ; cout << " кв. од." << endl;

}

};

int main()

{ figure \*p; // створення покажчика на об'єкт базового типу

triangle ObjT; // створення об'єктів похідних типів

rectangle ObjR; // створення об'єкта похідного типу

circle ObjC; // створення об'єкта похідного типу

p = &ObjT; // присвоєння покажчику адреси об'єкта похідного класу

p->Set(10.3, 5.5);

p->Show();

p = &ObjR; // присвоєння покажчику адреси об'єкта похідного класу

p->Set(10.3, 5.5);

p->Show();

p = &ObjC; // присвоєння покажчику адреси об'єкта похідного класу

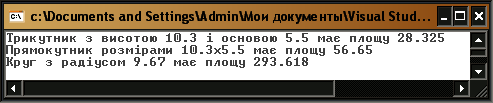
p->Set(9.67);

p->Show();

getch();

}

Внаслідок виконання ця програма відображає на екрані такі результати:



### **Суто віртуальні функції та абстрактні класи**

Якщо віртуальна функція, яка не перевизначена у похідному класі, викликається об'єктом цього похідного класу, то використовується та її версія, яку було визначено в базовому класі. Але у багатьох випадках взагалі немає сенсу давати визначення віртуальної функції в базовому класі. Наприклад, в базовому класі figure (з попереднього прикладу) визначення функції Show() – це просто заглушка. Вона не обчислює і не відображає площу жодного з об'єктів.

Розглянемо клас triangle. Він абсолютно зайвий, якщо у ньому не визначити функцію Show(). У цьому випадку варто створити метод, який би гарантував, що похідний клас дійсно містить усі необхідні функції. У C++ для вирішення цього питання і передбачено *суто віртуальні функції*.

***Суто віртуальна функція*** – це віртуальна функція, яку оголошено в базовому класі, але вона не має у ньому ніякого визначення. Тому будь-який похідний тип повинен визначити власну версію цієї функції, адже у нього просто немає ніякої можливості використовувати версію з базового класу (через її відсутність).

Щоб оголосити суто віртуальну функцію, використовують такий загальний формат:

virtual тип\_результата ім'я\_функції (перелік\_параметрів) = 0;

У цьому записі позначення “= 0” є ознакою того, що функція оголошується як суто віртуальна. Наприклад, в наступній версії визначення базового класу figure функція Show() вже представлена як суто віртуальна:

class figure

{ protected:

double x, y;

public:

void Set(double \_x, double \_y=0) { x = \_x; y = \_y; }

virtual void Show() = 0; // суто віртуальна функція

};

Оголосивши функцію суто віртуальною, програміст створює умови, при яких похідний клас просто вимушений мати визначення власної її реалізації. Без цього компілятор видасть повідомлення про помилку. Наприклад, якщо спробувати скомпілювати нижче наведену модифіковану версію програми обчислення площ геометричних фігур, у якій з класу circle видалено визначення функції Show(), то буде отримана помилка.

class figure

{ protected:

double x, y;

public:

void Set(double \_x, double \_y) { x = \_x; y = \_y; }

virtual void Show() = 0; // суто віртуальна функція

};

class triangle : public figure

{ public:

void Show()

{ cout << "Трикутник з висотою " << x << " i основою " << y;

cout << " має площу " << x \* 0.5 \* y << " кв. од." << endl; }

};

class rectangle : public figure

{ public:

void Show()

{ cout << "Прямокутник розмірами " << x << " x " << y;

cout << " має площу " << x \* y << " кв. од." << endl; }

};

class circle : public figure

{ // Вiдсутнiсть визначення функцiї Show() викличе повiдомлення про помилку

};

int main()

{ figure \*p; // створення покажчика на об'єкт базового типу

triangle ObjT; // створення об'єктів похідних типів

rectangle ObjR; // створення об'єкта похідного типу

circle ObjC; // створення об'єкта похідного типу

p = &ObjT; // присвоєння покажчику адреси об'єкта похідного класу

p->Set(10.3, 5.5);

p->Show();

p = &ObjR; // присвоєння покажчику адреси об'єкта похідного класу

p->Set(10.3, 5.5);

p->Show();

getch();

}

Клас, у якому є хоча б одна чиста віртуальна функція, називається *абстрактним*. Призначення абстрактного класу - забезпечити інтерфейс з іншими класами.

Абстрактний клас характеризує одна важлива особливість: у такого класу не може бути об'єктів. Абстрактний клас можна використовувати тільки як базовий, з якого виводитимуться інші похідні класи. Причина того, що абстрактний клас не можна використовувати для побудови об'єктів, полягає у тому, що його одна або декілька функцій не мають визначення.

Але навіть якщо базовий клас є абстрактним, то його все одно можна використовувати для оголошення покажчиків і посилань, які необхідні для підтримки динамічного поліморфізму.

### **Статичне (раннє) та динамічне (пізнє) зв'язування**

У C++, як і у інших об’єктно-орієнтованих мовах програмування, терміни раннє *зв'язування* (*early binding*) і *пізнє зв'язування* (*late* *binding*) пов'язують з подіями, які відбуваються при компілюванні та у період виконання програми відповідно.

З принципових переваг раннього зв'язування можна назвати ефективність: воно працює швидше від пізнього і часто вимагає менших витрат пам'яті. Його основний недолік – відсутність гнучкості.

Пізнє зв'язування означає, що точне рішення про виклик функції буде ухвалено у процесі виконання програми. Пізнє зв'язування у C++ досягається за рахунок застосування віртуальних функцій і похідних клсів.

Перевага пізнього зв'язування полягає у тому, що воно забезпечує великий ступінь гнучкості коду. Його можна застосовувати для підтримки загального інтерфейсу і давати змогу при цьому різним об'єктам, які використовують цей інтерфейс, визначати їх власні реалізації. Окрім того, пізнє зв'язування може допомогти програмісту у створенні бібліотек класів, що характеризуються багатократним використанням і можливістю розширюватися. До його недоліків можна віднести, хоч і незначне, але все таки зменшення швидкості виконання програм (через побудову ТVM -віртуальних таблиць, визначення віртуальних функцій).

Відповідь на запитання: чому віддати перевагу – ранньому або пізньому зв'язуванню, залежить від призначення програми. Пізнє зв'язування – це один з найпотужніших засобів мови програмування C++. Проте за цю потужність доводиться розплачуватися втратами у швидкості виконання програм. Тому пізнє зв'язування найкраще використовувати тільки у разі, коли воно істотно покращує структуру і керованість програмою.