Лекція 8 Віртуальні функції і поліморфізм

Одним з трьох основних принципів об'єктно-орієнтованого програмування є поліморфізм. Стосовно до C++ поліморфізм є термін, який використовується для опису процесу, в якому різні реалізації функції можуть бути доступні за допомогою одного і того ж імені. З цієї причини поліморфізм іноді характеризується фразою «один інтерфейс, багато методів». Це означає, що до всіх функцій-членів загального класу можна отримати доступ одним і тим же способом, незважаючи на можливі відмінності в конкретних діях, пов'язаних з кожною окремою операцією.

У C++ поліморфізм підтримується як під час виконання, так в період компіляції програми. Перевантаження операторів і функцій – це приклади поліморфізму, що відноситься до часу компіляції. Але, не дивлячись на могутність механізму перевантаження операторів і функцій, він не в змозі вирішити всі задачі, які виникають в реальних застосуваннях об'єктно-орієнтованої мови програмування. Тому в C++ також реалізований поліморфізм періоду виконання на основі використання похідних класів і віртуальних функцій.

Вказівники на похідні типи

Фундаментом для динамічного поліморфізму служить вказівник на базовий клас. Вказівники на базові і похідні класи пов'язані такими відносинами, які не властиві вказівникам інших типів. Вказівник одного типу, як правило, не може вказувати на об'єкт іншого типу. Однак вказівники на базові класи і об'єкти похідних класів – виключення з цього правила. У C++ вказівник на базовий клас також можна використовувати для посилання на об'єкт будь-якого класу, успадкованого від базового. Припустимо, що у нас є базовий клас Animal і класи Cat, God, Frog і Cow, які успадковані від класу Animal. (Рис. 1.) У C++ вказівник, що був оголошений як вказівник на базовий клас Animal, може бути також вказівником на похідні класи Cat, God, Frog і Cow.

```
Animal *A[4]; // вказівники на базовий клас
Cat B; Dog C; Frog D; Cow E;
A[0] = &B; // вказівник на базовий клас посилається на
//об'єкт класу Cat
A[1] = &C; // вказівник на базовий клас посилається на
//об'єкт класу Dog
A[2] = &D; // вказівник на базовий клас посилається на
//об'єкт класу Frog
A[3] = &E; // вказівник на базовий клас посилається на
//об'єкт класу Cow
```

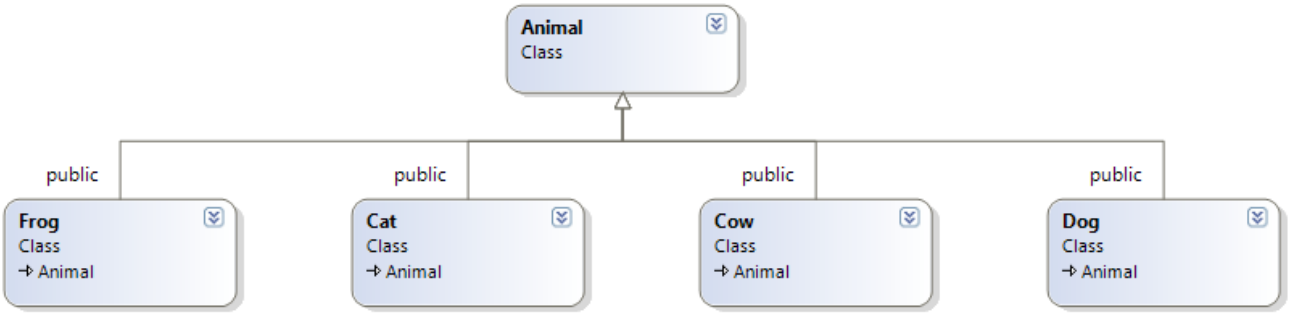


Рис. 1. Класи Cat, God, Frog і Cow успадковані від класу Animal.

Крім того, необхідно розуміти, що хоча «базовий» вказівник можна використовувати для доступу до об'єктів будь-якого похідного типу, зворотне твердження не вірно. Іншими словами, використовуючи вказівник на похідний клас, не можна отримати доступ до об'єкта базового типу. На рис. 2. зображено вікно повідомлень про помилки під час компіляції програми.

```
Cat *P;
Animal A;
P=&A;
```

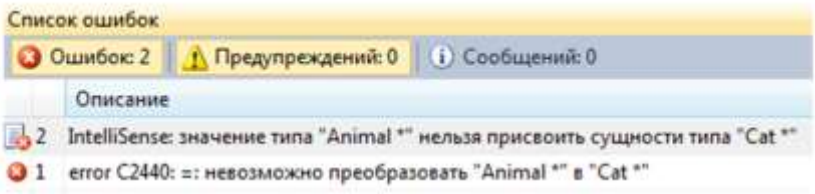


Рис. 2. Помилки під час компіляції програми.

Той факт, що вказівник на базовий тип можна використовувати для посилання на будь-який об'єкт, успадкованій від базового, надзвичайно важливий і принциповий для C++. Як буде показано нижче, ця гнучкість є ключовим моментом для способу реалізації динамічного поліморфізму в C++.

Подібно вказівниками, *посилання* на базовий клас також можна використовувати для доступу до об'єкта похідного типу. Ця можливість особливо часто застосовується при передачі аргументів функцій. Параметр, який має тип посилання на базовий клас, може приймати об'єкти базового класу, а також об'єкти будь-якого іншого типу, успадкованого від нього.

Віртуальні функції

Динамічний поліморфізм можливий завдяки поєднанню двох засобів: спадкування і віртуальних функцій.

Віртуальна функція – це функція, яка оголошується в базовому класі з використанням ключового слова virtual і перевизначається в одному або декількох похідних класах. Таким чином, кожен похідний клас може мати власну версію віртуальної функції.

Коли віртуальна функція викликається через вказівник (або посилання) на базовий клас, C++ визначає, яку саме версію віртуальної функції потрібно викликати, по типу об'єкта, що адресується цим вказівником. Причому слід мати на увазі, що це рішення приймається під час виконання програми. Отже, при посиланні на різні об'єкти будуть викликатися і різні версії віртуальної функції. Саме за типом об'єкта, що адресується, (а не по типу самого вказівника) визначається, яка версія віртуальної функції буде виконана. Таким чином, якщо базовий клас містить віртуальну функцію і якщо з цього базового класу успадковані два (або більше) інших класу, то при адресації різних типів об'єктів через вказівник на базовий клас будуть виконуватися і різні версії віртуальної функції. Аналогічний механізм працює і при використанні посилання на базовий клас.

Щоб оголосити функцію віртуальної, досить випередити її оголошення ключовим словом virtual.

Функція оголошується віртуальною в базовому класі за допомогою ключового слова virtual. При перевизначені віртуальної функції в похідному класі ключове слово virtual повторювати не потрібно (хоча це не буде помилкою).

Атрибут virtual передається «у спадок».

Якщо функція оголошується як віртуальна, вона залишається такою незалежно від того, через скільки рівнів похідних класів вона може пройти.

Якщо віртуальна функція перевизначається в похідному класі, її називають перевизначеною.

Прототипи віртуальної функції і її перевизначень повинні бути абсолютно однаковими. Якщо прототипи будуть різними, то така функція буде просто вважатися перевантаженою, і її «віртуальна сутність» втратиться.

І ще одне важливе зауваження: *конструктор не успадковується*. Конструктор базового класу створює об’єкт, який «живе» в об'єкті похідного класу. Тому деструкторам дозволяється бути віртуальними, а конструкторам – ні!

```
class Cat : public
{
public:
    virtual ~Cat()
    {
    }
```

```
class Cat : public Animal
{
public:
    virtual Cat()
    {
    }
```

Вище показано два фрагмента коду, зліва правильний, а справа - з помилкою.

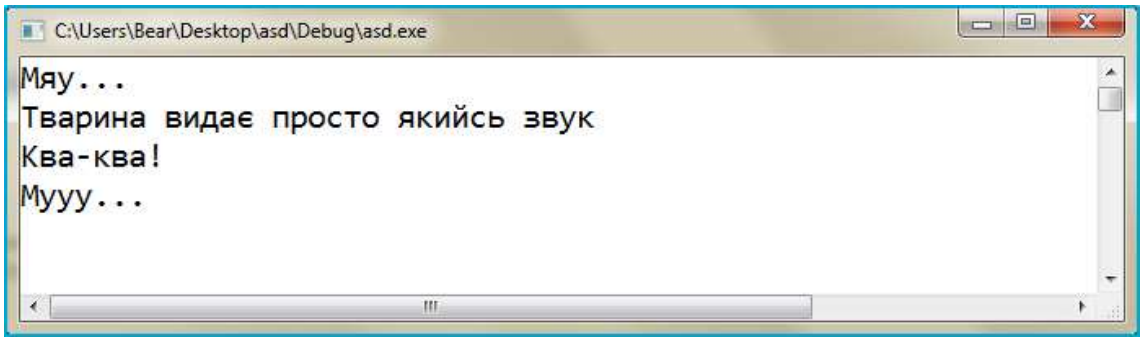
Клас, який включає віртуальну функцію, називається поліморфним класом. Цей термін також застосовується до класу, який успадковує базовий клас, що містить віртуальну функцію.

Розглянемо наступну коротку програму, в якій демонструється використання віртуальних функцій.

```
class Animal
{
public:
 virtual void GetSound(){
 cout<<"Тварина видає просто якийсь звук "<<endl;
 }
};

class Cat : public Animal
{
public:


 void GetSound() // Перевизначення функції GetSound()
                  //для класу Cat
 {
      cout<<"Мяу..."<<endl;
 }
};
class Dog : public Animal
{
public:
 /*void GetSound() // Перевизначення функції GetSound()
                    //для класу Dog закоментоване
 {
      cout<<"Гав!"<<endl;
 }*/
};
class Frog : public Animal
{
public:
 void GetSound() // Перевизначення функції GetSound()
                  //для класу Frog
 {
      cout<<"Ква-ква!"<<endl;
 }
};

class Cow : public Animal
{
public:
 void GetSound() // Перевизначення функції GetSound()
                        //для класу Cow
 {
      cout<<"Муууу..."<<endl;
 }
};
```

```
int main()
{
setlocale(0,"Ukr");
Animal *A[4]; // вказівники на базовий клас
Cat B; Dog C; Frog D; Cow E;
A[0] = &B; // вказівник на базовий клас посилається на
//об'єкт класу Cat
A[1] = &C; // вказівник на базовий клас посилається на
//об'єкт класу Dog
A[2] = &D; // вказівник на базовий клас посилається на
//об'єкт класу Frog
A[3] = &E; // вказівник на базовий клас посилається на
//об'єкт класу Cow
for(int i=0; i<4; i++)
 A[i]->GetSound(); // поліморфний виклик!!!
}
```

При виконанні програма генерує ось такі результати:



Якщо похідний клас не перевизначає віртуальну функцію, то використовується функція, що визначена в базовому класі.

Як бачимо, в похідному класі Dog функція `GetSound()` не перевизначається и тоді викликається функція базового класу.

Поліморфізм забезпечує можливість деякому узагальненому класу визначати функції, які будуть використовувати всі похідні від нього класи, причому похідний клас може визначити власну реалізацію деяких або всіх цих функцій. Іноді ця ідея виражається в такий спосіб: базовий клас диктує загальний інтерфейс, який буде мати будь-який об'єкт, успадкований від цього класу, але дозволяє при цьому похідному класу визначити метод, який використовується для реалізації цього інтерфейсу. Ось чому для опису поліморфізму часто використовується фраза «один інтерфейс, багато методів».

Чому ж так важливий загальний інтерфейс з безліччю реалізацій? Відповідь знову повертає нас до основної причини виникнення об'єктно-орієнтованого програмування: такий інтерфейс дозволяє програмісту впоратись з дедалі більшою складністю програм. Наприклад, якщо коректно розробити програму, то можна бути впевненим в тому, що до всіх об'єктів, успадкованим від базового класу, можна буде отримати доступ єдиним (загальним для всіх) способом, не дивлячись на те, що конкретні дії у одного похідного класу можуть відрізнятися від дій у іншого.

Розробнику похідного класу не потрібно заново винаходити елементи, вже існуючі в базовому класі. Більш того, відділення інтерфейсу від реалізації дозволяє створювати бібліотеки класів, написанням яких можуть займатися сторонні організації. Коректно реалізовані бібліотеки повинні надавати загальний інтерфейс, який програміст може використовувати для виведення класів відповідно до своїх конкретних потреб. Таким чином можна визначити поліморфізм як один інтерфейс для безлічі реалізацій (Рис. 3)

Рис. 3. Поліморфізм – це один інтерфейс для безлічі реалізацій.

Коли потрібен віртуальний деструктор

Розглянемо приклад із застосуванням динамічної пам’яті. Конструктор базового класу (Parent) виділяє динамічну пам’ять, а деструктор її звільняє. Від класу Parent успадковується похідний клас Child, його конструктор теж виділяє динамічну пам’ять, а деструктор її звільняє. Так саме робить конструктор і деструктор класу SubChild, похідний від класу Child. Відносини між цими класами показані на рис. 4.

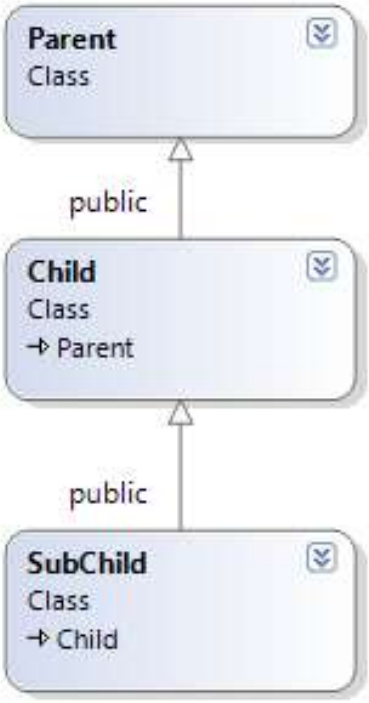


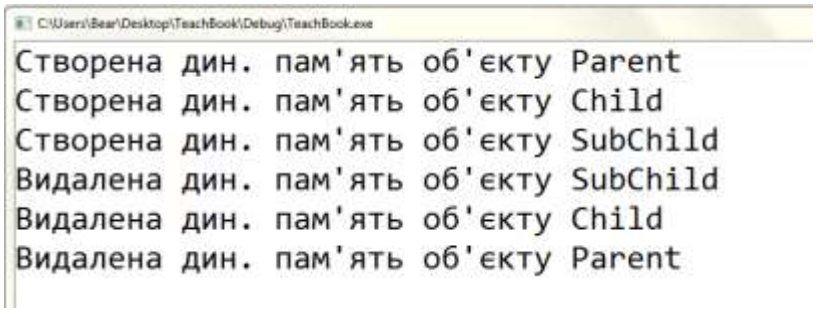
Рис. 4. Відносини між класами з віртуальними деструкторами.

У наведеному тексті програми «віртуальність» деструктора базового класу успадковується деструкторами похідних класів:

```
class Parent
{
public:
 Parent()
 {
      cout << "Створена дин. пам'ять об'єкту Parent" << endl;
 }
 virtual ~Parent()
 {
      cout << "Видалена дин. пам'ять об'єкту Parent" << endl;
 }
```

```
};
class Child : public Parent
{
public:
 Child()
 {
      cout << "Створена дин. пам'ять об'єкту Child" << endl;
 }
 ~Child()
 {
      cout << "Видалена дин. пам'ять об'єкту Child" << endl;
 }
};
class  SubChild : public Child
{
public:
 SubChild()
 {
      cout << "Створена дин. пам'ять об'єкту SubChild" << endl;
 }
 ~SubChild()
 {
      cout << "Видалена дин. пам'ять об'єкту SubChild" << endl;
 }
};

int main() {
 setlocale(0, "");
 Parent *P2 = new SubChild();
 delete P2;
}
```

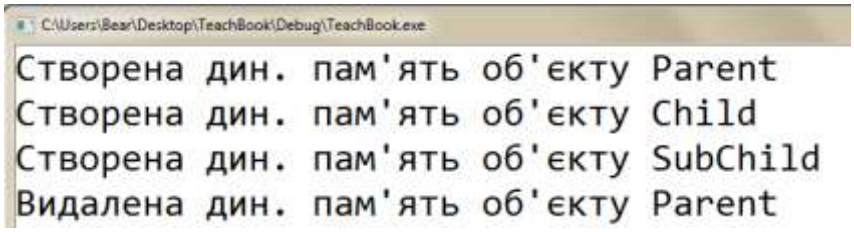
І результат отримуємо такий:



А тепер приберемо ключове слово virtual для деструктора класу Parent:

```
virtual ~Parent()
{
    cout << "Видалена дин. пам'ять об'єкту Parent" << endl;
}
```

Після цього програма видає такий результат:



Тобто динамічна пам’ять звільняється не повністю. Це випадок, коли деструктор обов'язково має бути віртуальним.

Суто віртуальні функції та абстрактні класи

У багатьох випадках взагалі немає сенсу давати визначення віртуальної функції в базовому класі. Наприклад, в базовому класі Animal (з попереднього прикладу) визначення функції GetSound () – це просто вивід тексту без особливого сенсу, тому, що неможливо собі уявити, який звук видає просто тварина.

Конкретний звук завжди видає конкретна тварина, тому функцію GetSound() обов'язково треба перевизначити для всіх похідних класів.

Суто віртуальна функція – це віртуальна функція, яка не має визначення в базовому класі.

Суто віртуальна функція – це функція, яка оголошена в базовому класі, але не має в 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, використовується наступний загальний формат:

```
virtual тип ім’я_функції (список_параметрів) = 0;
```

Тут під елементом тип мається на увазі тип значення, що повертається функцією, а елемент ім’я_функції- її ім'я. Позначення = 0 є ознакою того, що функція тут оголошується як суто віртуальна.

Оголосивши функцію суто віртуальною, програміст створює умови, при яких похідний клас просто змушений мати визначення власної її реалізації. Без цього компілятор видасть повідомлення про помилку.

```
class Animal
{
public:
 virtual void GetSound()=0; // Суто віртуальна функція
};
class Cat : public Animal
{
public:
 void GetSound() // Перевизначення функції GetSound() для класу
Cat
 {
       cout<<"Мяу..."<<endl;
 }
};
class Dog : public Animal
{
public:
 /*void GetSound() // Перевизначення функції GetSound()
                     //для класу Cat закоментоване
 {
       cout<<"Гав!"<<endl;
 }*/
};
```

Наприклад, при компіляції програми, фрагмент якої представлений вище, з'являється ось таке вікно повідомлень (Рис. 5):

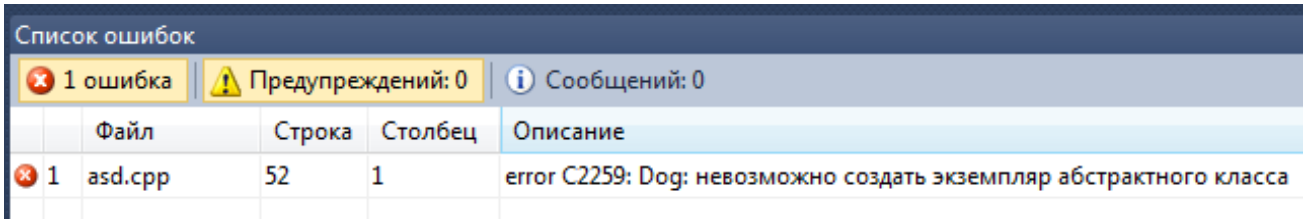


Рис. 5. Повідомлення під час компіляції програми.

Якщо клас має хоча б одну суто віртуальну функцію, його називають абстрактним. Абстрактний клас характеризується однією важливою особливістю: у такого класу не може бути об'єктів. Абстрактний клас можна використовувати тільки в якості базового, з якого будуть успадковуватися інші класи. Причина того, що абстрактний клас не можна використовувати

для створення об'єктів, лежить, безумовно, в тому, що хоча б одна з його функцій не має визначення. Але навіть якщо базовий клас є абстрактним, його все одно можна використовувати для оголошення вказівників і посилань, які необхідні для підтримки динамічного поліморфізму.