Лекція 6

Тема: Віртуальні функції

Мова C++ забезпечує як статичний, так і динамічний поліморфізм. Як вказувалося в попередніх главах, статичний поліморфізм досягається за допомогою перевантаження функцій і операторів. Динамічний поліморфізм реалізується на основі спадкування й віртуальних функцій. Саме ця тема перебуває в центрі уваги даної глави.

Віртуальні функції

Віртуальна функція (virtual function) — це функція-член, оголошена в базовому класі й перевизначена в похідному. Щоб створити віртуальну функцію, слід указати ключове слово **virtual** перед її оголошенням у базовому класі. Похідний клас перевизначає цю функцію, пристосовуючи її для своїх потреб. По суті, віртуальна функція реалізує принцип “один інтерфейс, кілька методів”, що лежить в основі поліморфізму. Віртуальна функція в базовому класі визначає вид інтерфейсу, тобто спосіб виклику цієї функції. Кожне перевизначення віртуальної функції в похідному класі реалізує операції, властиві лише даному класу. Інакше кажучи, перевизначення віртуальної функції створює конкретний метод (specific method).

На віртуальні функції накладаються наступні обмеження.

1. Вони не можуть бути статичними.
2. Вони не можуть бути дружніми.
3. Конструктори не можуть бути віртуальними.

Оскільки об'єкти похідного класу, по суті, являють собою модифіковані об'єкти базового класу, на них у мові C++ можна посилатися за допомогою вказівника на об'єкти базового класу. Якщо вони містять віртуальні функції, їх вибір буде виконаний на етапі виконання програми. Ієрархія класів, зв'язаних визначеною віртуальною функцією, називається поліморфічним кластером.

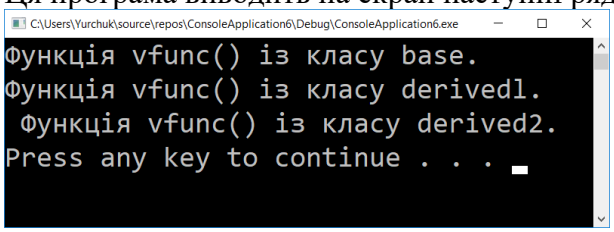
При звичайному виклику віртуальні функції нічим не відрізняються від інших функцій-членів. Особливі властивості віртуальних функцій проявляються при їхньому виклику за допомогою вказівників. Вказівники на об'єкти базового класу можна використовувати для посилання на об'єкти похідних класів. Якщо покажчик на об'єкт базового класу встановлюється на об'єкт похідного класу, що містить віртуальну функцію, вибір необхідної функції ґрунтується на типі об'єкта, на який посилається покажчик, причому цей вибір здійснюється в ході виконання програми. Таким чином, якщо покажчик посилається на об'єкти різних типів, то будуть викликані різні віртуальні функції. Це ставиться й до посилань на об'єкти базового класу.

Розглянемо для початку наступний приклад.

```
class base {
public:
	virtual void vfunc() {
		cout << "Функція vfunc() із класу base.\n";
	}
};
class derived1 : public base {
public:
	void vfunc() {
		cout << "Функція vfunc() із класу derivedl.\n";
	}
};
class derived2 : public base {
public:
	void vfunc() {
		cout << " Функція vfunc() із класу derived2.\n";
		int main()
		{
			base *p, b;
			derived1 d1; derived2 d2;
			// Покажчик на об'єкт базового класу,
			p = &b;
			p->vfunc(); // Виклик функції vfunc() із класу base.
					// Покажчик на об'єкт класу derived1.
			p = &d1;
			p->vfunc(); // Виклик функції vfunc() із класу derivedl.
					// Покажчик на об'єкт класу derived2.
```

```
        p = &d2;
        p->vfunc(); // Виклик функції vfunc() із класу derived2.
        return 0;
}
```

Ця програма виводить на екран наступні рядки.



Як показує ця програма, усередині класу **base** оголошена віртуальна функція **vfunc().** Зверніть увагу на ключове слово **virtual** в оголошенні функції. При перевизначенні функції **vfunc**() у класах **derived1** і **derived2** ключове слово **virtual** не потрібно. (Однак його використання не є помилкою, просто воно не обов'язково.)

У даній програмі класи **derived1** і **derived2** є похідними від класу **base**. Усередині кожного із цих класів функція **vfunc**() перевизначається заново відповідно до нового призначення. У програмі **main**() оголошено чотири змінні.

Ім'я	Тип
p	Покажчик на базовий клас
b	Об'єкт базового класу
d1	Об'єкт класу derivedl
d2	Об'єкт класу derived2

Крім того, вказівнику **p** привласнюється адреса об'єкта **b**, а функція **vfunc**() викликається за допомогою вказівника **p**. Оскільки вказівник **p** посилається на об'єкт класу **base**, виконується варіант функції **vfunc()** з базового класу. Потім вказівнику **p** присвоюється адреса об'єкта **d1**, і функція **vfunc**() знову викликається з його допомогою. Цього разу вказівник **p** посилається на об'єкт класу **derived1**. Отже, викликається функція **derived1::vfunc()**. У наприкінці вказівнику **p** присвоюється адреса об'єкта **d2**, тому вираз **p->vfunc()** приводить до виклику функції **vfunc**() із класу **derived2**. Принципово важливо, що варіант викликуваної функції визначається типом об'єкта, на який посилається покажчик **p**. Крім того, вибір відбувається в ході виконання програми, що забезпечує основу динамічного поліморфізму.

Віртуальну функцію можна викликати звичайним способом, використовуючи ім'я об'єкта й оператор однак поліморфізм досягається тільки при звертанні до неї через покажчик. Наприклад фрагмент, що випливає, програми є зовсім правильним.

d2.vfunc(); // Викликається функція **vfunc**() із класу **derived2**.

Незважаючи на те що такий виклик віртуальної функції помилкою не є, ніяких переваг він не надає.

На перший погляд, перевантаження віртуальної функції в похідному класі мало відрізняється від звичайного перевантаження функцій. Однак це не так, і термін перевантаження не застосуємо до перевизначення віртуальних функцій з кількох причин. Найбільш важлива відмінність полягає в тому, що прототип перевантаженої віртуальної функції повинен точно збігатися із прототипом, визначеним у базовому класі. Цим віртуальні функції відрізняються від перевантажених, які відрізняються типами й кількістю параметрів. (Фактично при перевантаженні функцій типи й кількість їх параметрів повинні відрізнятися! Саме ці відмінності дозволяють компіляторові вибирати правильний варіант перевантаженої функції.) При перевантаженні віртуальної функції всі аспекти їх прототипів повинні бути однаковими. Якщо не дотримувати цього правила, компілятор буде вважати ці функції просто перевантаженими, а їх віртуальна природа буде загублена. Друге важливе обмеження полягає в тому, що віртуальні функції не можуть бути статичними членами класів. Крім того, вони не можуть бути дружніми функціями. І, нарешті, конструктори не можуть бути віртуальними, хоча на деструктори це обмеження не поширюється.

Через перераховані обмеження для перевизначення віртуальної функції в похідному класі використовується термін заміщення (overriding).

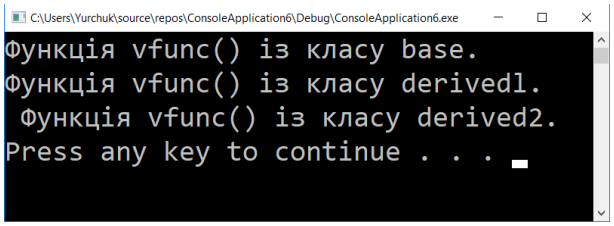
Виклик віртуальної функції за допомогою посилання на об'єкт базового класу

У попередньому прикладі віртуальна функція викликалася за допомогою вказівника на об'єкт базового класу, однак поліморфна природа віртуальних функцій зберігається й при їхньому виклику за допомогою посилання на об'єкт базового класу. Псилання є неявним вказівником. Таким чином, посилання на об'єкт базового класу можна використовувати для звертання до об'єкта базового або будь-якого похідного класу. Якщо віртуальна функція викликається за допомогою посилання на об'єкт базового класу, її варіант залежить від типу об'єкта, на який установлено посилання в момент виклику.

У більшості випадків віртуальна функція за допомогою посилання викликається при передачі параметрів. Розглянемо ще один варіант попередньої програми.

/* У цій програмі для виклику віртуальної функції застосовується посилання на об'єкт базового класу. */

```
#include <iostream>
using namespace std;
class base {
public:
	virtual void vfunc() {
		cout << "Функція vfunc() із класу base.\n";
	}
};
	class derived1 : public base {
	public:
		void vfunc() {
			cout << " Функція vfunc() із класу derived1.\n";
		}
	};
	class derived2 : public base {
	public:
		void vfunc() {
			cout <<  " Функція vfunc () із класу derived2.\n";
		}
	};
	// Використовується параметр, що є посиланням на об'єкт
	// базового класу,
	void f(base &r)
	{
		r.vfunc();
	}
	int main()
	{
		base b; derived1 d1; derived2 d2;
		f(b);  //     Функції      f()    передається   об'єкт класу  base,
		f(d1); //     Функції      f()    передається   об'єкт класу  derived1.
		f(d2); //     Функції      f()    передається   об'єкт класу  derived2.
		system("pause");
		return 0;
	}
```



Ця програма виводить на екран ті ж повідомлення, що й раніше. У даному прикладі функція **f** () одержує в якості параметра посилання на об'єкт класу **base**. У функції **main**() ця функція викликається за допомогою об'єктів класів **base**, **derived1** і **derived2**. Конкретний варіант функції **vfunc**() вибирається усередині функції **f**() залежно від типу її параметра.

Для простоти в інших прикладах цієї глави віртуальні функції викликаються за допомогою вказівників, причому результат нічим не відрізняється від виклику за допомогою посилань.

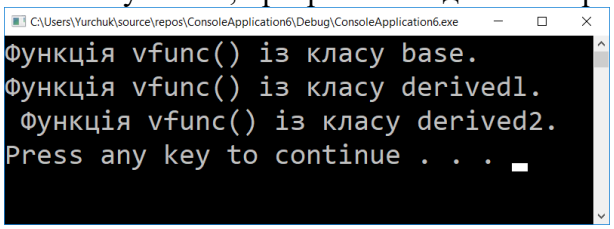
Атрибут virtual успадковується

При успадкуванні віртуальної функції її віртуальна природа також успадковується. Це значить, що якщо похідний клас, що успадкував віртуальну функцію від базового класу, стає базовим стосовно іншого похідного класу, віртуальна функція може як і раніше заміщатися. Інакше кажучи, не має значення,

скільки раз успадковувалася віртуальна функція, вона однаково залишається віртуальної. Розглянемо наступну програму.

```
#include <iostream>
using namespace std;
class base {
public:
	virtual void vfunc() {
		cout << "Функція vfunc() із класу base.\n";
	}
};
class derived1 : public base {
public:
	void vfunc() {
		cout << " Функція vfunc() із класу derived1.\n";
	}
};
		/* Клас derived2 успадковує віртуальну функцію vfunc()  від класу derived1. */
		class derived2 : public derived1 {
		public:
			// Функція vfunc() залишається віртуальної,
			void vfunc() {
				cout <<" Функція vfunc() із класу derived2.\n";
			}
		};
		int main()
		{
			base *p, b;
			derived1 d1;
			derived2 d2;
			// Покажчик на об'єкт класу base.
			p = &b;
			p->vfunc(); // Виклик функції vfunc() із класу base.
					// Покажчик на об'єкт класу derivedl
				p = &d1;
			p->vfunc(); // Виклик функції vfunc() із класу derived1.
					// Покажчик на клас derived2
			p = &d2;
			p->vfunc(); // Виклик функції vfunc() із класу derived2.
			return 0;
		}
```

Як і очікувалося, програма виводить на екран наступні повідомлення.



У цьому випадку клас **derived2** є спадкоємцем класу **derived1**, а не класу **base**, але функція **vfunc**() залишається віртуальної.

Віртуальні функції є ієрархічними

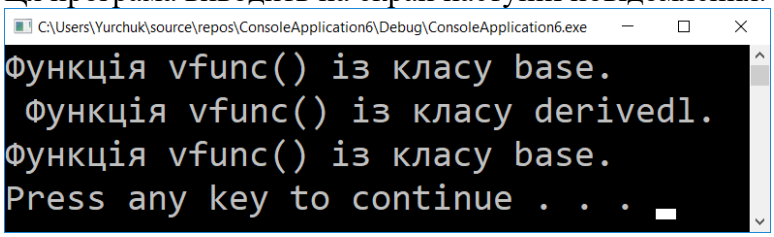
Як відомо, якщо функція оголошена віртуальною в базовому класі, її можна замістити в похідному класі. Однак віртуальну функцію не обов'язково заміщати. У цьому випадку викликається функція, визначена в базовому класі. Розглянемо як приклад програму, у якій клас **derived2** не перевантажує функцію **vfunc**().

```
#include <iostream>
using namespace std;
class base {
public:
	virtual void vfunc() {
		cout << "Функція vfunc() із класу base.\n";
	}
};
```

```
class derived1 : public base {
public:
	void vfunc() {
		cout << " Функція vfunc() із класу derivedl.\n";
	}
};
class derived2 : public base {
public:
	// Функція vfunc() не заміщається в класі derived2,
	// використовується версія із класу base.
};
int main()
{
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);

	base *p, b;
	derived1 d1; derived2 d2;
	// Вказівник на об'єкт класу base.
p= &b;
	p->vfunc(); // Виклик функції vfunc() із класу base.
			// Покажчик на об'єкт класу derived1.
	p = &d1;
	p->vfunc(); // Виклик функції vfunc() із класу derived1.
			// Покажчик на об'єкт класу derived2.
	p = &d2;
	p->vfunc(); // Виклик функції vfunc() із класу base,
	system("pause");
		return 0;
}
```

Ця програма виводить на екран наступні повідомлення.



Оскільки функція **vfunc**() не заміщається в класі **derived2**, через покажчик на об'єкт класу **derived2** викликається її версія із класу **base**.

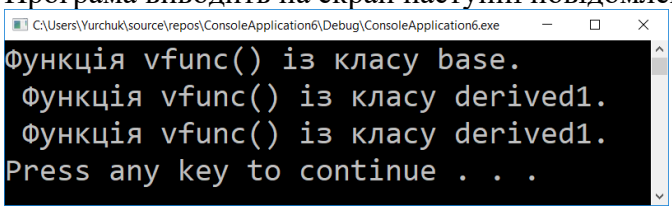
Попередня програма ілюструє окремий випадок більш універсального правила. Успадкування в мові C++ організоване по ієрархічному принципу, тому віртуальні функції також повинні бути ієрархічними. Це значить, що якщо віртуальна функція не заміщається, викликається її попередня перевизначена версія. Наприклад, у наступній програмі клас **derived2** є спадкоємцем класу **derived1**, який, у свою чергу, є похідним від класу **base** Однак функція **vfunc**() у класі **derived2** не заміщається. Отже, найближча до класу **derived2** версія функції **vfunc**() визначена в класі **derived1**. Таким чином, виклик функції **vfunc**() за допомогою об'єкта класу **derived2** ставиться До Функції **derivedl**::**vfunc** ().

```
#include <iostream>
using namespace std;
class base {
public:
	virtual void vfunc() {
		cout << "Функція vfunc() із класу base.\n";
	}
};
class derived1 : public base {
public:
	void vfunc()
 {
	cout << " Функція vfunc() із класу derived1.\n";
}
};
class derived2 : public derived1 {
public:
	/* Функція vfunc() не заміщається в класі derived2.
```

```
	Оскільки клас derived2 є спадкоємцем класу derivedl, викликається функція vfunc() із класу
derivedl.
	*/
};
int main()
{
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);

	base *p, b;
	derived1 d1;
	derived2 d2;
	// Покажчик на об'єкт класу base
	p = &b;
	p->vfunc(); // Виклик функції vfunc() із класу base.
			// Покажчик на об'єкт класу derived1.
	p = &d1;
	p->vfunc(); // Функція vfunc() із класу derived1.
			// Покажчик на об'єкт класу derived2
	p = &d2;
	p->vfunc(); // Функція vfunc() із класу derived1.
	system("pause");
	return 0;
}
```

Програма виводить на екран наступні повідомлення.



Чисто віртуальні функції

Отже, якщо віртуальна функція не заміщається в похідному класі, викликається її версія з базового класу. Однак у багатьох випадках неможливо створити розумну версію віртуальної функції в базовому класі. Наприклад, базовий клас може не мати достатній обсяг інформації для створення віртуальної функції. Крім того, у деяких ситуаціях необхідно гарантувати, що віртуальна функція буде заміщена у всіх похідних класах. Для цих ситуацій у мові C++ передбачені чисто віртуальні функції.

Чисто віртуальна функція (pure virtual function) — це віртуальна функція, що не має визначення в базовому класі. Для оголошення чисто віртуальної функції використовується наступна синтаксична конструкція.

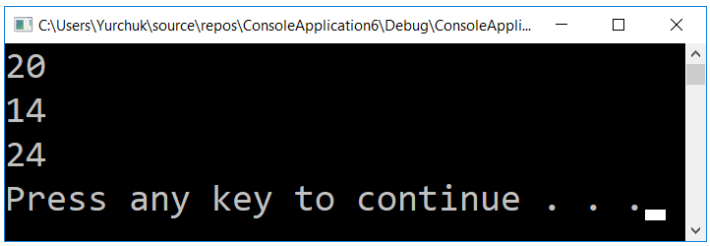
virtual тип ім'я_функції (список_параметрів) = 0;

Чисто віртуальні функції повинні перевантажуєтся в кожному похідному класі, а якщо ні, то виникне помилка компіляції.

Наступна програма містить простий приклад чисто віртуальної функції. Базовий клас **number** містить ціле число **val**, функцію **setval**() і чисто віртуальну функцію **show**(). Похідні класи **hextype**, **dectype** і **octtype** є спадкоємцями класу **number** і перевизначають функцію **show**() так, що вона виводить значення **val** у відповідній системі числення (шістнадцятковій, десяткової або вісімковій).

```
#include <iostream>
using namespace std;
class number {
protected: int val;
public:
	void setval(int i) { val = i; }
	// Функція show() є чисто віртуальною,
	virtual void show() = 0;
};
class hextype : public number {
public:
	void show() {
		cout << hex << val << "\n";
	}
```

```
};
class dectype : public number {
public:
	void show() {
		cout << val << "\n";
	}
};
class octtype : public number {
public:
	void show() {
		cout << oct << val <<" \n";
	}
};
int main()
{
	dectype d; hextype h; octtype o;
	d.setval(20);
	d.show(); // Виводить десяткове число 20.
	h.setval(20);
	h.show(); // Виводить шістнадцяткове число 14.
	o.setval(20);
	o.show(); // Виводить вісімкове число 24.
	system("pause");
	return 0;
    }
```



Незважаючи на простоту цього прикладу, він досить яскраво ілюструє ситуацію, коли в базовому класі неможливо дати осмислене визначення віртуальної функції. У цьому випадку клас **number** просто забезпечує однаковий інтерфейс для використання похідних типів. Функцію **show**() неможливо визначити в класі **number**, оскільки в ньому не задана основа системи числення. Однак чисто віртуальна функція **show**() гарантує, що в кожному похідному класі вона буде відповідним чином перевизначена.

Слід мати у виді, що всі похідні класи зобов'язані перевизначати чисто віртуальну функцію. Якщо цього не зробити, виникне помилка компіляції.

Абстрактні класи

Клас, що містить хоча б одну чисто віртуальну функцію, називається абстрактним (abstract class). Оскільки абстрактний клас містить одну або кілька функцій, що не мають визначення (тобто чисто віртуальні функції), його об'єкти створити неможливо. Отже, абстрактні класи можна використовувати лише як основу для похідних класів.

Незважаючи на те що об'єкти абстрактного класу не існують, можна створити покажчики й посилання на абстрактний клас. Це дозволяє застосовувати абстрактні класи для підтримки динамічного поліморфізму й вибирати відповідну віртуальну функцію залежно від типу покажчика або посилання.

Застосування віртуальних функцій

В основу об'єктно-орієнтованого програмування покладений принцип “один інтерфейс, кілька методів’’. Він дозволяє визначати базовий клас операцій з однаковим інтерфейсом, а їх конкретизацію надавати похідним класам.

Віртуальні функції, абстрактні класи й динамічний поліморфізм являють собою один з найбільш потужних і гнучких механізмів реалізації принципу “один інтерфейс, кілька методів”. Використовуючи цей механізм, можна створювати ієрархії класів, організовані за принципом “ від загального — до частки” (від базового класу — до похідних). По цьому методу в базовому класі слід визначати всі універсальні властивості й інтерфейси. Якщо деяку операцію можна реалізувати тільки в похідному класі, використовується віртуальна функція. По суті, у базовому класу описуються лише самі загальні властивості, а в похідних класах вони конкретизуються.

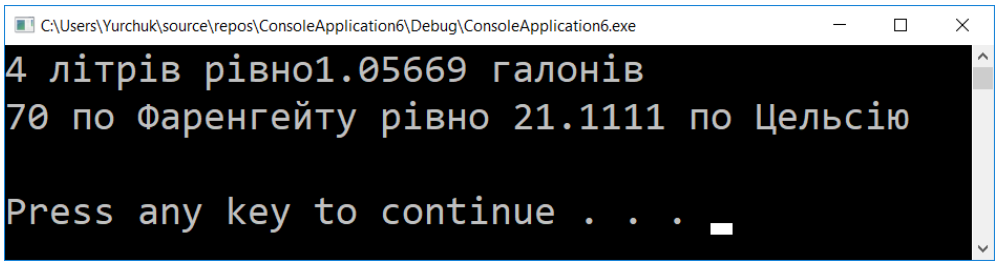
Проілюструємо сказане наступним прикладом, у якім створюється ієрархія класів, що виконують перетворення одиниці виміру з однієї системи вимірювання в іншу (наприклад, літрів — у галони). У

базовому класі **convert** оголошено дві змінні — **val1** і **val2**, у яких зберігаються вихідне й перетворене значення відповідно. Крім того, у ньому визначені функції **getinit**() і **getconv** () які повертають значення. Ці члени класу **convert** фіксовані й можуть використовуватися у всіх похідних класах. Однак функція **compute**(), що виконує фактичне перетворення, є чисто віртуальною й повинна визначатися у всіх класах, похідних від класу **convert**. Конкретний зміст функції **compute** () залежить від типу виконуваного перетворення.

```
#include "windows.h"

#include <iostream>
using namespace std;
class convert {
protected:
	double val1; // Початкове значення,
	double val2; // Перетворене значення,
public:
	convert(double i) {
		val1 = i;
	}
	double getconv() { return val2; }
	double getinit() { return val1; }
	virtual void compute() = 0;
};
// Перетворення літрів у галони,
class l_to_g : public convert {
public:
	l_to_g(double i) : convert(i) { }
	void compute() {
		val2 = val1 / 3.7854;
	}
};
// Перетворення шкали Фаренгейта в шкалу Цельсія,
class f_to_c : public convert {
public:
	f_to_c(double i) : convert(i) { }
	void compute() {
		val2 = (val1 - 32) / 1.8;
	}
};
int main()
{
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);

	convert *p;   // Покажчик на базовий клас.
	l_to_g lgob(4); f_to_c fcob(70);
	// Застосування віртуальної функції,
	p = &lgob;
	cout << p->getinit() << " літрів рівно"; p->compute();
	cout << p->getconv() << " галонів\n";    // l_to_g
	p = &fcob;
	cout << p->getinit() << " по Фаренгейту рівно "; p->compute();
		cout << p->getconv() << " по Цельсію\n"; // f_to_c
		cout << endl;
		system("pause");
		return 0;
    }
```



У цій програмі створюються класи **l_to_g** і **f_to_c**, похідні від класу **convert**. Ці класи виконують перетворення обсягу, обмірюваного в літрах, в обсяг, виражений у галонах, а також переводять шкалу

температур по Фаренгейту в шкалу по Цельсію відповідно. Кожний похідний клас заміщає функцію **compute** () по- своєму, виконуючи необхідне перетворення. Однак, незважаючи на відмінність фактичних перетворень (тобто методів) у класах **l_to_g** і **f_to_c**, їхній інтерфейс однаковий.

Віртуальні функції дозволяють дуже легко реагувати на нову ситуацію. Припустимо, що в попередній програмі необхідно передбачити перетворення футів у метри, включивши його в клас **convert**.

```
// Перетворення футів у метри
class f_to_m : public convert {
public:
       f_to_m(double i) : convert(i) { }
       void compute() {
              val2 = vail / 3.28;
       }
    };
```

Абстрактні класи й віртуальні функції дозволяють створювати бібліотеки класів (class library), що носять узагальнений характер. Будь-який програміст може створити клас, похідний від бібліотечного, додавши свої власні функції. При цьому буде збережений однаковий інтерфейс, певний базовим класом Таким чином, бібліотечні класи можна адаптувати до нових ситуацій.

На закінчення відзначимо, що базовий клас **convert** є прикладом абстрактного класу. Віртуальна функція **compute** () не визначається в класі **convert**, оскільки в ньому немає інформації про виконуване перетворення. Функція **compute**() наповнюється конкретним змістом лише в похідних класах.

Порівняння раннього й пізнього зв'язування

Раннє зв'язування означає події, що відбуваються на етапі компіляції. По суті, раннє зв'язування означає, що на етапі компіляції відома вся інформація, що дозволяє вибрати викликувану функцію. (Інакше кажучи, об'єкт і виклик функції зв'язуються один з одним на етапі компіляції.) Прикладами раннього зв'язування є звичайні виклики функції (включаючи стандартні бібліотечні функції), виклики перевантажених функцій і операторів. Основна перевага раннього зв'язування — ефективність. Оскільки вся інформація про викликувану функцію на етапі компіляції вже відома, сам виклик функції відбувається дуже швидко.

Пізніше зв'язування є антиподом раннього. У мові C++ пізніше зв'язування означає, що фактичний вибір викликуваної функції здійснюється тільки в ході виконання програми. Основним засобом пізнього зв'язування є віртуальні функції. Як відомо, вибір викликуваної віртуальної функції залежить від типу покажчика або посилання, за допомогою яких вона викликається. Оскільки в більшості випадків на етапі компіляції ця інформація відсутня, зв'язування об'єкта й виклику функції відкладається до виконання програми. Основна перевага пізнього зв'язування — гнучкість. На відміну від раннього, пізніше зв'язування дозволяє створювати програму, що реагує на події, що відбуваються в ході її виконання, без використання великої кількості коду, що передбачає всілякі варіанти. Однак пізніше зв'язування може сповільнити роботу програми.

Механізм віртуальних функцій

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.

Розглянемо наступну програму.

```
#include <iostream>
using namespace std;
class TBase
{
public:
      char a;
      TBase() :a('X') { cout<<"Ctor TBase\n"; }
      ~TBase() { cout << "Dtor TBase\n"; }
      //    virtual void print() {cout <<"TBase::a = "<<a<<endl;}
```

```
};
class TDerived : public TBase
{
public:
      char b;
      TDerived() :b('Y') { cout << "Ctor TDerived\n"; }
      ~TDerived() { cout << "Dtor TDerived\n"; }
      void print() { cout << "TDerived::b = "<< b<<endl; }
};
class TDerived2 : public TDerived
{
public:
      char c;
      TDerived2() :c('Z') { cout << "Ctor TDerived2\n"; }
      ~TDerived2() { cout << "Dtor TDerived2\n"; }
      void print() { cout << "TDerived2::c " <<c<<endl; }
};
int main()
{
      cout << "Sizeof(char) = "<< sizeof(char) << endl;
      cout << "Sizeof(TBase) = "<< sizeof(TBase) << endl;
      cout << "Sizeof(TDerived) = "<< sizeof(TDerived) << endl;
      cout << "Sizeof(TDerived2) = "<< sizeof(TDerived2) << endl;

      system("pause");
      return 0;
   }
```

```
Sizeof(char) = 1
Sizeof(TBase) = 1
Sizeof(TDerived) = 2
Sizeof(TDerived2) = 3
```

Отже, розмір типу char дорівнює одному байту. Клас TBase містить одне поле типу char, отже, його розмір також дорівнює одному байту. Клас TDerived успадковує поле типу char, додаючи його до власного члена b. Таким чином, розмір збільшується вдвічі. Аналогічна ситуація виникає в класі TDerived2, що містить три поля типу char — одне своє і два успадкованих.

Тепер знімемо коментар з наступної рядка.

```
//    virtual void print() {cout <<"TBase::a = "<<a<<endl;}
```

Результати роботи програми стануть зовсім іншими.

```
Sizeof(char) = 1
Sizeof(TBase) = 8
Sizeof(TDerived) = 12
Sizeof(TDerived2) = 16
```

Розмір вказівника на таблицю віртуальних функцій дорівнює 4 байт, отже, справжній розмір класу TBase повинний бути дорівнює 5 байт. Однак за рахунок вирівнювання машинного слова він збільшується до 8 байт.

Отже, розмір класу TDerived повинний бути рівним 8+4=12 байт. Як бачимо, це відповідає дійсності. Аналогічно, sizeof(TDerived2)= sizeof(TDerived)+sizeof(вказівник)= 12+4=16 байт. Таким чином, ця програма підтверджує загальновідомий факт, що кожен об'єкт поліморфічного кластера містить вказівник на таблицю віртуальних функцій. Виключення перевизначених версій з похідних класів нічого не змінює — віртуальні функції утворять ієрархію й успадковуються всіма похідними класами.

Віртуальний деструктор

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

```
#include <iostream>
```

```
using namespace std;
class TBase
{
public:
	char* a;
	TBase(const char* s) { a = _strdup(s); cout<<"Ctor TBase\n"; }
	~TBase() { delete a; cout << "Dtor TBase\n"; }
	virtual void print() { cout << "TBase::a = "<<a<<endl; }
};
class TDerived : public TBase
{
public:
	char* b;
	TDerived(const char* s1, const char* s2) :TBase(s2)
	{
		b = _strdup(s2); cout << "Ctor TDerived\n";
	}
	~TDerived() { delete b; cout << "Dtor TDerived\n"; }
	void print() { cout << "TDerived::b = "<< b<<endl; }
};
class TDerived2 : public TDerived
{
public:
	char* c;
	TDerived2(const char* s1, const char* s2, const char* s3) :TDerived(s2, s3)
	{
		c = _strdup(s3); cout << "Ctor TDerived2\n";
	}
	~TDerived2() { delete c; cout << "Dtor TDerived2\n"; }
	void print() { cout << "TDerived2::c = " <<c<<endl; }
};
int main()
{
	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.

Помітимо, що 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 екран наступні рядки.

Ctor TBase
Ctor TDerived
Ctor TDerived2
Dtor TBase

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.

Поставимо перед деструкторами класів TBase і TDerived (вони обоє є базовими для класів TDerived і TDerived2 відповідно) ключове слово virtual.

```
#include <iostream>
using namespace std;
class TBase
{
public:
	char* a;
	TBase(const char* s) { a = _strdup(s); cout<<"Ctor TBase\n"; }
	virtual ~TBase() { delete a; cout << "Dtor TBase\n"; }
	virtual void print() { cout << "TBase::a = "<<a<<endl; }
};
class TDerived : public TBase
{
public:
	char* b;
	TDerived(const char* s1, const char* s2) :TBase(s2)
	{
		b = _strdup(s2); cout << "Ctor TDerived\n";
	}
	virtual ~TDerived() { delete b; cout << "Dtor TDerived\n"; }
	void print() { cout << "TDerived::b = "<< b<<endl; }
};
class TDerived2 : public TDerived
{
public:
	char* c;
	TDerived2(const char* s1, const char* s2, const char* s3) :TDerived(s2, s3)
	{
		c = _strdup(s3); cout << "Ctor TDerived2\n";
	}
	~TDerived2() { delete c; cout << "Dtor TDerived2\n"; }
	void print() { cout << "TDerived2::c = " <<c<<endl; }
};
int main()
{
	TBase* pObj = new TDerived2("TBase", "TDerived", "TDerived2");
	delete pObj; // Помилка!
	system("pause");
	return 0;
}
```

Це приведе до наступного результатам.

```
Ctor TBase
Ctor TDerived
Ctor TDerived2
Dtor TDerived2
Dtor TDerived
Dtor TBase
```

Тепер пам'ять звільнена корректно.

Прийоми програмування

Дотепер ми вивчали, як слід і як не слід робити, працюючи з ієрархією класів. Зокрема, не можна створювати віртуальні конструктори. По визначенню віртуальний конструктор — це оксиморон, інакше кажучи, внутрішньо суперечлива сутність. Віртуальність функції виявляється, коли, знаходячись у межах ієрархії класів, необхідно послатися на існуючий об'єкт, тип якого заздалегідь невідомий. У той же час конструктор викликається, коли об'єкта ще не існує (власне, саме для його створення він і викликається!). Це явне протиріччя.

Однак це протиріччя можна зняти, використовувавши узагальнене рішення. Ми будемо так називати синтаксичну конструкцію, що імітує поводження забороненої сутності, будучи зовсім коректною із синтаксичної точки зору. Інакше кажучи, узагальнене рішення — це спосіб обдурити занадто строгий компілятор.

Отже, чого ми хочемо від віртуального конструктора? Щоб він створював об'єкти різного типу — у залежності від одержуваних аргументів. Спробуємо його реалізувати.

```
#include <iostream>
using namespace std;
class TBase {
public:
	char *a;
```

```
	virtual ~TBase() {}
	virtual TBase* virtual_copy() const
	{
		cout<<"TBase virtual copy\n";
		return new TBase(*this);
	}
	virtual TBase* virtual_ctor() const
	{
		cout<<"TBase virtual ctor \n";
		return new TBase();
	}
};
class TDerived : public TBase {
public:
	char* b;
	TDerived* virtual_copy() const
	{
		cout<<"TDerived virtual copy\n";
		return new TDerived(*this);
	};
	TDerived* virtual_ctor() const
	{
		cout<<"TDerived virtual ctor\n";
		return new TDerived();
	}
};
int main()
{
	TBase objBase;
	TDerived objDerived;
	TBase* pBase = &objBase;
	TBase* s2 = pBase->virtual_ctor();
	TBase* s1 = pBase->virtual_copy();
	pBase = &objDerived;
	TDerived* s3 = (TDerived*)pBase->virtual_ctor();
	TDerived* s4 = (TDerived*)pBase->virtual_copy();
	delete s1;
	delete s2;
	delete s3;
	delete s4;

	system("pause");
	return 0;
}
```

Зрозуміло, конструктор і конструктор копіювання віртуальними не стали. Просто віртуальні функції, що заміщаються в похідному класі, звертаються до відповідного конструктора, а звертатися до цих функцій можна за допомогою вказівника на базовий клас.

У результаті виконання програми на екрані з'являться наступні рядки.

```
TBase virtual ctor
TBase virtual copy
TDerived virtual ctor
TDerived virtual copy
```

В ідіомі віртуального конструктора використовуються ослаблені обмеження на типи значень, що повертаються віртуальними функціями. Інакше кажучи, тепер прототипи віртуальних функцій у базовому і похідному класах не зобов'язані збігатися. На жаль, компілятор Microsoft C++ 6.0 не підтримує цю можливість. Для того щоб реалізувати ідіому віртуального конструктора, слід скористатися більш сучасними компіляторами.

Модифікатори override і final

Для вирішення певних проблем в **спадкуванні** в C++11 додали два спеціальних модифікатори: override і final. Зверніть увагу, ці модифікатори не є **ключовими словами** — це звичайні модифікатори, які мають особливе значення в певному контексті.

Хоча final використовується не часто, override ж є фантастичним доповненням, яке ви повинні використовувати регулярно. На цьому уроці ми розглянемо обидва цих модифікатори, а також один виняток з правил, коли тип повернення перевизначення може не збігатися з типом повернення **віртуальної функції** батьківського класу.

Модифікатор override

Віртуальна функція дочірнього класу є перевизначенням, тільки якщо збігаються її сигнатура і тип повернення з сигнатурою і типом повернення віртуальної функції батьківського класу. А це, в свою чергу, може призвести до проблем, коли функція, яка повинна бути перевизначенням, насправді, ним не є.

Розглянемо наступний приклад:

```
#include <iostream>

class A
{
public:
	virtual const char* getName1(int x) { return "A"; }
	virtual const char* getName2(int x) { return "A"; }
};

class B : public A
{
public:
	virtual const char* getName1(short int x) { return "B"; }  // тип параметру – short int
	virtual const char* getName2(int x) const { return "B"; }  // метод є const
};

int main()
{
	B b;
	A& rParent = b;
	std::cout << rParent.getName1(1) << '\n';
	std::cout << rParent.getName2(2) << '\n';

	return 0;
}
```

Оскільки rParent — це **посилання** класу A на об’єкт b, то за допомогою віртуальних функцій ми маємо намір отримати доступ до B::getName1() і до B::getName2(). Однак, оскільки в B::getName1() інший тип параметру (short int замість int), то він не є перевизначенням методу A::getName1(). Більше того, оскільки B::getName2() є const, а A::getName2() — ні, то B::getName2() також не рахується перевизначенням A::getName2().

Відповідно, результат виконання програми:

A

A

Конкретно в цьому випадку, оскільки A і B просто виводять свої імена, досить легко побачити, що щось пішло не так, і перевизначення не викликаються. Однак в складнішій програмі, коли методи можуть і не повертати значення, які виводяться на екран, знайти помилку вже буде досить проблематично.

Для вирішення такого типу проблем і додали **модифікатор override** в C++11. Модифікатор override може використовуватися з будь-яким методом, який повинен бути перевизначенням. Досить просто вказати override в тому місці, де зазвичай вказується const (після дужок з параметрами). Якщо метод не перевизначає віртуальну функцію батьківського класу, то компілятор видасть помилку:

```
#include <iostream>

class A
{
public:
	virtual const char* getName1(int x) { return "A"; }
	virtual const char* getName2(int x) { return "A"; }
	virtual const char* getName3(int x) { return "A"; }
};

class B : public A
{
```

```
public:
	virtual const char* getName1(short int x) override { return "B"; }  // помилка компіляції, метод не є перевизначенням
	virtual const char* getName2(int x) const override { return "B"; }  // помилка компіляції, метод не є перевизначенням
	virtual const char* getName3(int x) override { return "B"; }  // все добре, метод є перевизначенням A::getName3(int)

};

int main()
{
	return 0;
    }
```

Тут ми отримаємо дві помилки: перша для B::getName1() і друга для B::getName2(), тому що жоден з цих методів не є перевизначенням віртуальних функцій класу A. Метод B::getName3() є перевизначенням, тому з ним ніяких проблем не виникає.

Використання модифікатора override ніяк не впливає на ефективність або продуктивність програми, але допомагає уникнути випадкових помилок. Отже, настійно рекомендується використовувати модифікатор override для кожного зі своїх перевизначень.

Правило: Використовуйте модифікатор override для кожного зі своїх перевизначень.

Модифікатор final

Можуть бути випадки, коли ви не хочете, щоб хтось міг перевизначати віртуальну функцію або наслідувати певний клас. Модифікатор final використовується саме для цього. Якщо користувач намагається перевизначити метод або наслідувати клас з **модифікатором final**, то компілятор видасть помилку.

Вказується final в тому ж місці, в якому і модифікатор override, наприклад:

```
class A
{
public:
	virtual const char* getName() { return "A"; }
};

class B : public A
{
public:
	// Помітили final в кінці? Це означає, що метод перевизначити вже не можна
	virtual const char* getName() override final { return "B"; } // все добре, перевизначення A::getName()
};

class C : public B
{
public:
	virtual const char* getName() override { return "C"; } // помилка компіляції: перевизначення методу B::getName(), який є final
    };
```

У цьому коді метод B::getName() перевизначає метод A::getName(). Але B::getName() має модифікатор final, це означає, що будь-які подальші перевизначення цього методу викликатимуть помилку компіляції. І дійсно, C::getName() вже не може перевизначити B::getName() — компілятор видасть помилку.

У разі, якщо ми хочемо заборонити наслідування певного класу, то модифікатор final вказується після імені класу:

```
class A
{
public:
	virtual const char* getName() { return "A"; }
};

class B final : public A // зверніть увагу на модифікатор final тут
{
public:
	virtual const char* getName() override { return "B"; }
```

```
};

class C : public B // помилка компіляції: заборонено наслідувати final-клас
{
public:
      virtual const char* getName() override { return "C"; }
    };
```

У цьому прикладі клас B оголошений як final. Таким чином, клас C не може наслідувати клас B — компілятор видасть помилку.