Лекція 4

Тема: Наслідування

ПЛАН

Принципи інкапсуляції, приховання інформації й абстракції даних забезпечують захист об'єктів від несанкціонованого втручання. Однак це лише одна зі сторін об'єктно-орієнтованого програмування. Вона дозволяє перейти від індивідуального будівництва програм до великоблочного, але в той же час сковує можливості програміста. Дійсно, одержавши готовий модуль із дружелюбним і зрозумілим інтерфейсом, ви можете лише пристосувати його до своїх потреб. Якщо ж вам знадобиться розширити його функціональні можливості, прийдеться знову звернутися до розроблювача з проханням модифікувати чи переписати модуль самому. Ані те, ані інше не занадто вигідно. Набагато ефективніше надати програмісту можливість використовувати готовий модуль як базу для власних розробок, не переробляючи його код. Цей підхід називається повторним використанням коду.

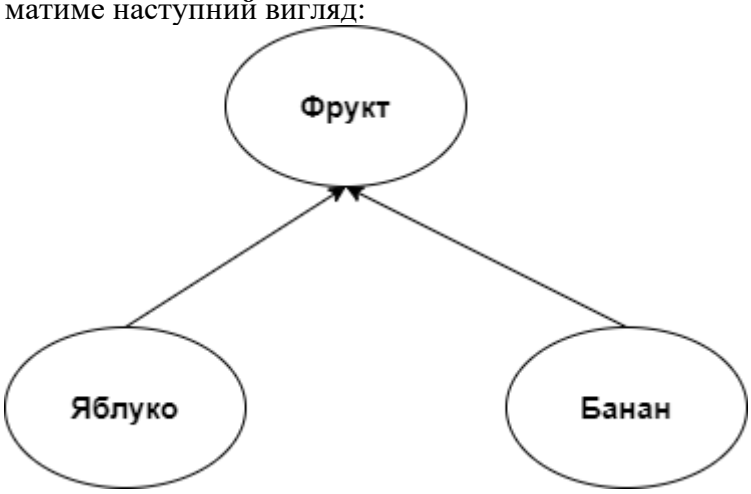
Крім того, досить часто в багатьох додатках інформація уже організована у виді визначеної структури, чи ієрархії. Систему, що має бути змоделюваною, іноді можна подати у виді об'єктів, зв'язаних особливими відношеннями, які можна було б назвати подібністю. Таку подібність можна знайти навіть у математичних задачах, де коло об'єктів сильно обмежене. Наприклад, вектор можна вважати матрицею, що складається з одного стовпця і декількох рядків, а число — вектором, що складається з одного рядка й одного стовпця. При розв’язанні математичних задач це не має ніякого значення, однак у процесі програмування цю обставину можна використовувати для підвищення ефективності програми.

На відміну від композиції об’єктів, яка включає в себе створення нових об’єктів шляхом об’єднання інших об’єктів, спадкування включає в себе створення нових об’єктів шляхом безпосереднього збереження властивостей і поведінки інших об’єктів, а потім їх розширення або навпаки — конкретизації. Подібно композиції об’єктів, спадкування відбувається всюди в реальному житті. Наприклад, при народженні ви успадкували гени від своїх батьків, і вам передалися певні фізичні властивості від кожного з них (схильність до хвороб, видам діяльності тощо), але потім ви додали свою особистість до всього отриманого. Технологічні продукти (комп’ютери, смартфони тощо) успадковують функціонал від своїх попередників, при цьому додаючи щось своє (нове/унікальне) і зберігаючи сумісність. Наприклад, процесор Intel Pentium успадкував багато функціональних властивостей від процесора Intel 486, який, в свою чергу, успадкував свій функціонал від більш ранніх процесорів. Мова C++ багато успадкувала від мови програмування Сі, на якій вона базується, а мова програмування Сі успадкувала багато властивостей від інших мов програмування, які були до неї.

Розглянемо приклад з яблуками і бананами. Хоча яблуко і банан — це різні фрукти, але у них обох є одна загальна властивість: вони обидва є фруктами. І оскільки яблука і банани — це фрукти, то логічно що все, що вірно для фруктів, вірно і для яблук з бананами. Наприклад, всі фрукти мають свою назву, колір і розмір. Яблука і банани також мають свої назви, колір і розмір. Ми можемо сказати, що яблука і банани успадкували (отримали) всі властивості фруктів, тому що вони самі є фруктами. Ми також знаємо, що фрукти піддаються процесу дозрівання, завдяки якому вони стають їстівними. Оскільки яблука і банани є фруктами, то,

відповідно, вони також піддаються процесу дозрівання, в результаті чого стають їстівними.

Якщо зобразити відносини між яблуками, бананами і фруктами на діаграмі, то це матиме наступний вигляд:



Тут ми бачимо ієрархію.

Ієрархії

Ієрархія — це діаграма зі зв'язками об'єктів. Більшість ієрархій або демонструють прогресію з плином часу (386 > 486 > Pentium), або класифікують речі таким чином, щоб вони переходили від загального до конкретного (Фрукти > Яблука > Макінтош). Ще зі шкільної біології царство, тип, клас, ряд, родина, рід і вид є визначенням ієрархії (рух від загального до конкретного або навпаки).

Ось ще один приклад ієрархії: квадрат є прямокутником, який є чотирикутником, який є фігурою. Правильний трикутник — це трикутник, який також є фігурою:

Тут кожен елемент успадковує властивості і поведінку елементу над ним.

Успадкування - один з наріжних каменів об'єктно-орієнтованого програмування, тому

що воно дозволяє створювати ієрархічні класифікації. Використовуючи успадкування, можна створювати загальні класи, що визначають властивості, характерні для всієї сукупності родинних класів. Ці класи можуть успадковувати властивості один в одного, додаючи до них свої власні унікальні характеристики.

Згідно зі стандартною термінологією мови C++ клас, що лежить в основі ієрархії, називається базовим (base class), а клас, що успадковує властивості базового класу, - похідним (derived class). Похідні класи, у свою чергу, можуть бути базовими стосовно інших класів. У мові C++ передбачений потужний і гнучкий механізм успадкування. Успадкування буває одиночним і множинним. При одиночному успадкуванні в кожного похідного класу є лише один базовий клас, а при множинному — декілька.

1. Керування доступом до членів базового класу

При спадкуванні члени базового класу стають членами похідного класу. Керуючись оголошенням похідного класу, компілятор ніби збирає його з різних частин — спочатку він бере усі властивості базового класу, а потім додає до них нові функціональні можливості похідного. Як правило, для спадкування використовується наступна синтаксична конструкція.

class *ім'я-похідного- класу :рівень_доступу ім'я-базового- класу {*
/ / тіло класу
};

Параметр *рівень_доступу* визначає статус членів базового класу в похідному класі. У якості цього параметра використовуються специфікатори public, private або protected. Якщо рівень доступу не зазначений, то для похідного класу за замовчуванням використовується специфікатор private, а для похідної структури - public. Розглянемо варіанти, що виникають у цих ситуаціях.

Тип наслідування	Базовий клас	Наслідуваний клас
public:	public	public
	protected	protected
	private	private
protected:	public	protected
	protected	protected
	private	private
private:	public	private
	protected	private
	private	private

Якщо рівень доступу до членів базового класу задається специфікатором **public**, то всі відкриті й захищені члени базового класу стають відкритими й захищеними членами похідного класу. При цьому закриті члени базового класу не міняють свого статусу й залишаються недоступними членам похідного. Як демонструє наступна програма, об'єкти класу **derived** можуть безпосередньо посилатися на відкриті члени класу base.

```
#include <iostream>
using namespace std;
class base {
	int i, j;
public:
	void set(int a, int b) { i = a; j = b; }
	void show() { cout << i << " " << j << "\n"; }
};
class derived : public base {
	int k; public:
		derived(int x) { k = x;  }
		void showk() { cout << k << "\n"; }
};
```

```
int main() {
	derived ob(3);
	ob.set(1, 2); // Звертання до члена класу base
	ob.show(); // Звертання до члена класу base
	ob.showk(); // Звертання до члена класу derived
	return 0;
}
```

Якщо властивості базового класу успадковуються за допомогою специфікатора доступу **private**, усі відкриті й захищені члени базового класу стають закритими членами похідного класу. Наприклад програма нище навіть не буде скомпільована, тому що обидві функції set() і show() тепер є закритими членами класу derived.

```
//  Ця програма  не  буде  скомпільована,
#include  <iostream>
using namespace  std;
class base {
	int   i, j; public:
		void  set(int  a, int  b) { i = a;   j = b; }
		void  show() { cout << i << "    " << j << "\n"; }
};
//   Відкриті  члени  класу base  є
//  закритими членами класу  derived,
class  derived : private  base {
	int  k;
public:
	derived(int x) { k = x; }
	void  showk()  {  cout << k << "\n";
} };
int main() {
	derived ob(3);
	ob.set(1, 2);   //   Помилка,   доступ  до  функції  set () заборонений.
	ob.show();   //   Помилка,   доступ  до  функції  show()   заборонений.
	return  0;
}
```

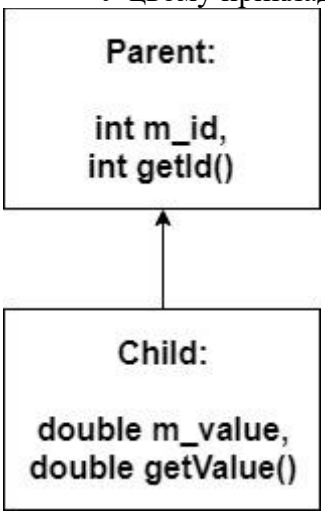
При закритому успадкуванні всі відкриті й захищені члени базового класу стають закритими членами похідного класу. Це значить, що вони залишаються доступними членам похідного класу, але недоступні іншим елементам програми, що не є членами базового або похідного класів.

Ось 2 класи, які допомагатимуть нам ілюструвати важливі моменти:

```
class Parent
{
public:
    int m_id;

    Parent(int id = 0)
        : m_id(id)
    {
    }

    int getId() const { return m_id; }
};

class Child : public Parent
{
public:
    double m_value;

    Child(double value = 0.0)
        : m_value(value)
    {
    }

    double getValue() const { return m_value; }
```

```
};
```

У цьому прикладі клас Child є дочірнім, а клас Parent — батьківським:



Оскільки Child успадковує змінні-члени і методи класу Parent, то ви можете припустити, що члени класу Parent копіюються в клас Child, але це не так. Замість цього розглядайте Child як клас, який складається з двох частин: перша — Parent, друга — Child:

Ви вже бачили безліч прикладів того, що відбувається при створенні об’єктів звичайного (НЕ дочірнього) класу:

```
int main()
{
    Parent parent;

    return 0;
}
```

Parent — це не дочірній клас, тому що він не наслідує властивості будь-яких інших класів. Мова C++ виділяє пам’ять для Parent, потім викликається **конструктор за замовчуванням** класу Parent для виконання ініціалізації.

Тепер розглянемо, що відбувається при створенні об’єктів дочірнього класу:

```
int main()
{
    Child child;

    return 0;
}
```

На перший погляд все начебто так само, як і в попередньому прикладі, але “під капотом” це не зовсім так. Як ми вже говорили, клас Child складається з двох частин: частина Parent і частина Child. Коли C++ створює об’єкти дочірніх класів, то він робить це поетапно. Спочатку створюється найвищий клас ієрархії (той, який батько). Потім створюється дочірній клас, який йде наступним по порядку, і так до тих пір, поки не буде створено останній клас (той, який знаходиться в самому низу ієрархії).

Тому, при створенні об’єкта класу Child, спочатку створюється частина Parent класу Child (з використанням конструктора за замовчуванням класу Parent) і після того, як з частиною Parent завершено, створюється друга частина Child (з використанням конструктора за замовчуванням класу Child). У нашому прикладі більше немає дочірніх класів, тому на цьому все і завершується.

Цей процес насправді легко проілюструвати:

```
#include <iostream>

class Parent
```

```
{
public:
    int m_id;

    Parent(int id = 0)
        : m_id(id)
    {
        std::cout << "Parent\n";
    }

    int getId() const { return m_id; }
};

class Child : public Parent
{
public:
    double m_value;

    Child(double value = 0.0)
        : m_value(value)
    {
        std::cout << "Child\n";
    }

    double getValue() const { return m_value; }
};

int main()
{
    std::cout << "Instantiating Parent:\n";
    Parent dParent;

    std::cout << "Instantiating Child:\n";
    Child dChild;

    return 0;
}
```

Результат виконання програми:

```
Instantiating Parent:
Parent
Instantiating Child:
Parent
Child
```

Як ви можете бачити, при створенні Child спочатку створюється частина Parent класу Child. У цьому є сенс, тому що (логічно) дитина не може існувати без батьків. Це також сприяє безпеці та ефективності виконання коду: дочірній клас часто використовує змінні-члени і методи батька, але батьківський клас нічого не знає про свій дочірній клас. Початкова ініціалізація батьківського класу гарантує, що його змінні-члени і методи будуть проініціалізовані до моменту використання їх дочірнім класом.

Порядок побудови класів в “ланцюжку” спадкувань

Часто трапляються ситуації, коли одні класи успадковують властивості інших класів, які, в свою чергу, успадковують властивості своїх попередніх (батьківських) класів. Наприклад:

```
class A
{
public:
    A()
    {
```

```
        std::cout << "A\n";
    }
};

class B : public A
{
public:
    B()
    {
        std::cout << "B\n";
    }
};

class C : public B
{
public:
    C()
    {
        std::cout << "C\n";
    }
};

class D : public C
{
public:
    D()
    {
        std::cout << "D\n";
    }
};
```

Пам’ятайте, що в C++ завжди йде побудова з «першого» або «топового» класу ієрархії. Потім C++ переходить до наступного класу ієрархії і виконує його побудову. Цей процес послідовний.

Проілюструємо порядок побудови класів у вищенаведеному “ланцюжку” спадкування:

```
int main()
{
    std::cout << "Constructing A: \n";
    A cA;

    std::cout << "Constructing B: \n";
    B cB;

    std::cout << "Constructing C: \n";
    C cC;

    std::cout << "Constructing D: \n";
    D cD;
}
```

Результат:

```
Constructing A:
A
Constructing B:
A
B
Constructing C:
A
B
C
Constructing D:
A
B
```

```
C
D
```

Таким чином мова C++ виконує побудову дочірніх класів поетапно, починаючи з верхнього класу ієрархії і закінчуючи нижнім класом ієрархії. У міру побудови кожного класу для виконання ініціалізації викликається відповідний конструктор відповідного класу.

2. Успадкування й захищені члени

Специфікатор **protected** підвищує гнучкість механізму спадкування. Якщо член класу оголошений захищеним (**protected**), то поза класом він недоступний. Із цього погляду захищений член класу нічим не відрізняється від закритого. Єдине виключення із цього правила стосується спадкування. У цій ситуації захищений член класу суттєво відрізняється від закритого.

Як вказувалося в попередньому розділі, закритий член базового класу не доступний іншим елементам програми, включаючи похідний клас. Однак захищені члени базового класу поводяться інакше. При відкритому спадкуванні захищені члени базового класу стають захищеними членами похідного класу й, отже, доступні іншим членам похідного класу. Іншими словами, захищені члени класу стосовно свого класу є закритими й. у той же час, можуть успадковуватися похідним класом. Розглянемо приклад.

```
#include  <iostream>
using namespace  std;
class  base {
protected:
       int   i, j;    //   Закриті стосовно     класу  base,
                                  //   але доступні класу derived,
public:
       void  set(int  a, int  b) { i = a;    j = b; }
       void  show() { cout << i << "     " << j << "\n"; }
};
class  derived : public base {
       int   k;
public:
       // Клас derived має доступ до членів i й j із класу base
       void setk() { k = i*j; }
       void showk() { cout << k << "\n"; }
};
int main() {
       derived ob;
       ob.set(2, 3); // Усе в порядку, цей член доступний класу derived
       ob.show(); // Усе в порядку, цей член доступний класу derived
       ob.setk();
       ob.showk();
       return 0;
}
```

У даному прикладі, оскільки клас **derived** успадковує властивості класу base за допомогою відкритого спадкування, а змінні i й j оголошені захищеними, функція seek() із класу derived має до них доступ. Якби змінні i й j були оголошені в класі **base** закритими, то клас **derived** не мав би до них доступу, і програму не можна було скомпілювати.

Якщо похідний клас є базовим стосовно іншого похідного класу, то будь-який захищений член вихідного базового класу, відкрито наслідуваний першим похідним класом, також може успадковуватися другим похідним класом як захищений член. Наприклад програма, що описана нище, цілком коректна, і клас derived2 дійсно має доступ до змінних i й j.

```
#include  <iostream>
using namespace  std;
class base
{
protected: int i, j;
```

```
public:
       void set(int a, int b) { i = a; j = b; }
       void show() { cout << i << " " << j << "\n"; }
};
// Змінні і й j успадковуються як захищені,
class derivedl : public base {
       int k;
public:
       void setk() { k = i*j; } // доступно
       void showk() { cout << k << "\n"; }
};
class derived2 : public derivedl {
       int m;
public:
       void setm() { m = i - j; } // Допускається
       void showm() { cout << m << "\n"; }
};
int main() {
       derivedl ob1;
       derived2 ob2;
       ob1.set(2, 3); ob1.show(); ob1.setk(); ob1.showk();
       ob2.set(3, 4); ob2.show(); ob2.setk(); ob2.setm(); ob2.showk(); ob2.showm();
       return 0;
}
```

Однак, якби до класу base застосовувався механізм закритого спадкування, те всі його члени стали б закритими членами класу derivedl і були недоступні класу derived2. (У той же час змінні і й j були б як і раніше доступні класу derivedl.) Ця ситуація ілюструється наступною програмою (вона містить помилку й не компілюється).

```
// Ця програма містить помилку,
#include  <iostream>
using namespace  std;
class  base {
protected: int   i, j;
public:
       void set(int a, int b) { i = a; j = b; }
       void show() { cout << i << " " << j << "\n"; }
};
// Тепер усі члени класу base є закритими членами
// класу derivedl.
class derivedl : private base {
       int k;
public:
       // Це робити можна, оскільки змінні і й j
       // є закритими членами класу derivedl.
       void setk() { k = i*j; } // OK
       void showk() { cout << k << "\n"; }
};
// Доступ до членів i, j , set() і show() не успадковується,
class derived2 : public derivedl {
       int m;
public:
       // Неправильно, тому що змінні і й j є закритими
       // членами класу derivedl
       void setm() { m = i - j; } // Помилка
       void showm() { cout << m << "\n"; }
};
int main() {
       derivedl obl;
       derived2 ob2;
       ob1.set(1, 2); // Помилка, викликати функцію set() не можна,
       obi.show(); // Помилка, викликати функцію show () не можна.
       ob2.set(3, 4); // Помилка, викликати функцію set() не можна.
       ob2.show(); // Помилка, викликати функцію show() не можна.
       return 0;
```

```
}
```

Якби до класу base застосовувалося закрите спадкування, клас derivedl як і раніше мав би доступ до його відкритих і захищених членів. Однак ці привілеї іншим спадкоємцям не передаються.

3. Захищене успадкування

До базового класу можна застосовувати механізм захищеного спадкування. При цьому всі відкриті й захищені члени базового класу стають захищеними членами похідного класу. Розглянемо приклад.

```
#include <iostream>
using namespace std;
class base {
protected:
	int i, j; // Закриті члени класу base,
			// доступні класу derived.
public:
	void setij(int a, int b) { i = a; j = b; }
	void showij() { cout << i << " " << j << "\n"; }
};
// Клас, отриманий за допомогою захищеного спадкування,
class derived : protected base {
	int k; public:
		// Клас derived має доступ до членів i, j і setij() із класу base.
		void setk() { setij(10, 12); k = i*j; }
		// Звідси можна викликати функцію showij().
		void showall() { cout << k << " "; showij(); }
};
int main() {
	derived ob;
	//  ob.setij(2, 3); // Невірно, функція setij() є
	//                          закритим членом класу derived.
	ob.setk(); // Вірно, викликається відкритий член класу derived,
	ob.showall(); // Вірно, викликається відкритий член класу derived.
			// ob.showij(); // Невірно, функція showij() є
			// захищеним членом класу derived
	return 0;
}
```

Як випливає из коментарів, незважаючи на те що функції setij() і showij() є відкритими членами класу base, у класі derived, утвореному за допомогою захищеного спадкування, вони стають захищеними. Це значить, що у функції main () вони не доступні.

4. Множинне успадкування

Похідний клас може одночасно успадковувати властивості декількох базових класів. Наприклад, у програмі, наведеної нижче, клас derived успадковує властивості класів basel і base2.

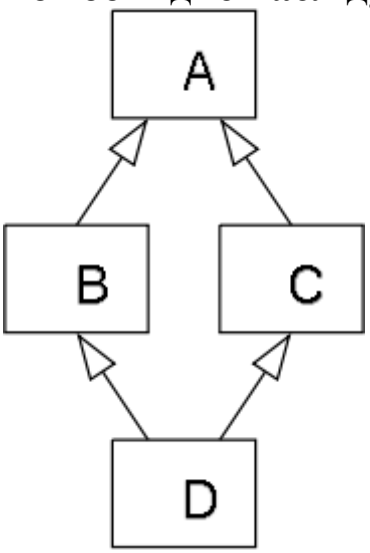
// Приклад множинного спадкування.

```
#include <iostream> using namespace  std;
class basel {
protected:
	int x;
public:
	void showx() { cout << x << "\n"; }
};

class base2 {
protected:
	int y;
```

```
public:
       void showy() (cout " в " "\n";
} },
// Множинне спадкування.
class derived : public basel, public base2 {
public:
       void set(int i, int j) { x = i; y = j; }
};
int main() {
       derived ob;
       ob.set(10, 20); // Ця функція належить класу derived,
       ob.showx(); // Ця функція належить класу basel.
       ob.showy(); // Ця функція належить класу base2.
       return 0;
}
```

Як бачимо, при множинному успадкуванні імена базових класів перелічуються в списку й розділяються комами, причому перед кожним іменем базового класу вказується свій специфікатор доступу.

Ромбовидне наслідування (diamond inheritance)



При множинному успадкуванні може виникнути неоднозначність. Розглянемо, наприклад неправильну програму, що випливає.

```
#include <iostream>
using namespace  std;
class base {
public:
       int  i;
};
// Клас derived1 є спадкоємцем класу base.
class derived1 : public base {
public:
       int j;
};
// Клас derived2 є спадкоємцем класу base.
class derived2 : public base {
public:
       int k;
};
/* Клас derived3 є спадкоємцем класів derivedl і derived2. Отже, у класі derived3 існують
дві копії класу base! */
class derived3 : public derived1, public derived2 {
public: int sum;
};
int main() {
```

```
        derived3 ob;
        ob.i = 10; // Неоднозначність, яка змінна і мається на увазі???
        ob.j = 20; ob.k = 30;
        // Тут змінна і також визначена неоднозначно,
        ob.sum = ob.i + ob.j + ob.k;
        // Тут змінна і також визначена неоднозначно,
        cout << ob.i << " ";
        cout << ob.j << " " << ob.k << " ";
        cout << ob.sum;
        return 0;
}
```

Як зазначено в коментарях, класи **derived1** і **derived2** є спадкоємцями класу **base**. Однак клас **derived 3** є похідним від обох класів **derived2** і **derived3**. (Таке спадкування називається діамантовим) Отже, в об'єкті класу **derived3** утримуються дві копії об'єкта класу **base**. Таким чином, вираз

ob.i = 20;

у якому відбувається звертання до змінної і, стає неоднозначним, оскільки невідомо, з об'єкта якого класу випливає взяти цю змінну: **derived1** або **derived2**. Володіючи двома копіями об'єкта класу **base**, об'єкт класу **ob** містить два екземпляри змінної **ob.i**! Як бачимо, цей оператор у принципі неоднозначний.

Цю програму можна виправити двома способами. По-перше, до змінної **i** можна застосувати оператор дозволу області видимості. Наприклад наступна програма працює коректно.

```
//У цій програмі для явного вибору змінної і
//  застосовується оператор дозволу області видимості.
#include <iostream>
using namespace std;
class base {
public:
        int   i;
};
// Клас derived1 є спадкоємцем класу base.
class derived1 : public base {
public:
        int j;
};
// Клас derived2 є спадкоємцем класу base.
class derived2 : public base {
public:
        int k;
};
/* Клас derived3 є спадкоємцем класів derived1 і derived2 одночасно. Отже, у кожному його
об'єкті втримуються дві копії об'єкта класу base! */
class derived3 : public derived1, public derived2 {
public: int sum;
};
int main() {
        derived3 ob;
        ob.derived1::i = 10;       // Неоднозначність усунута,
                                                // використовується змінна і із класу
derived1 .
        ob.j = 20; ob.k = 30;
        // Неоднозначність усунута

        ob.sum = ob.derived1::i + ob.j + ob.k;
        // Неоднозначність усунута
        cout << ob.derived1::i << " ";
        cout << ob.j << " " << ob.k << " ";
        cout << ob.sum;
        return 0;
}
```

Як бачимо, оператор дозволу області видимості "**::**" дозволяє явно вибрати варіант похідного класу. Однак цей розв'язок породжує нові проблеми. Що якщо насправді потрібна лише одна копія об'єкта класу base? чи Можна запобігти дублювання об'єктів класу **base** в об'єкті класу **derived3**? На обидва питання можна відповісти позитивно. Вирішення цих проблем засноване на застосуванні віртуальних базових класів (**virtual base classes**).
Якщо базовий клас має кілька спадкоємців, його дублювання можна запобігти. Для цього в оголошенні похідного класу перед іменем базового класу слід поставити ключове слово **virtual**. Наприклад попередню програму, можна виправити в такий спосіб.

```
//  Програма, що   використовує віртуальний базовий клас.
#include  <iostream>
using namespace std;
class base {
public:
       int   i;
};
// Клас derived1 є спадкоємцем віртуального класу base.
class derived1 : virtual public base {
public:
       int j;
};
// Клас derived2 є спадкоємцем віртуального класу base.
class derived2 : virtual public base {
public:
       int k;
};
/* Клас derived3 є спадкоємцем класів derived1 і derived2. Цього разу  його об'єкт містить
лише одну копію об'єкта базового класу. */
class derived3 : public derived1, public derived2 {
public: int sum;
};
int main() {
       derived3 ob;
       ob.i = 10; // Неоднозначність усунута
       ob.j = 20;

       ob.k = 30;
       // Неоднозначність усунута
       ob.sum = ob.i + ob.j + ob.k;
       // Неоднозначність усунута
       cout << ob.i << " ";
       cout << ob.j << " " << ob.k << " ";
       cout << ob.sum;
       return 0;
}
```

Як бачимо, перед іменем базового класу в специфікації похідного класу стоїть ключове слово **virtual**. Тепер обидва класи **derived1** і **derived2** є спадкоємцями віртуального базового класу **base**, і будь-які їхні спадкоємці будуть містити лише одну копію класу **base**. Отже, об'єкт класу **derived3** містить копію об'єкта класу **base**, і вираз **ob.i=10** стає зовсім правильним і однозначним.
Слід мати на увазі, що, хоча класи **derived1** і **derived2** оголосили клас **base** віртуальним, його об'єкти будуть як і раніше частиною об'єктів кожного із цих типів. Наприклад фрагмент, що випливає, є зовсім правильним.

//Визначаємо об'єкт класу derivedl
Derived1 myclass;
myclass.i = 88;

Відмінність між звичайним базовим класом і віртуальним проявляється, тільки коли об'єкт успадковує кілька об'єктів того самого базового класу. Якщо використовується віртуальний

базовий клас, то його об'єкт тільки один раз копіюється в кожний об'єкт похідного класу. А якщо ні, то виникає неоднозначність.