Лекція 5

Тема: Наслідування. Конструктори, деструктори й успадкування

У зв'язку зі спадкуванням виникають два питання, що стосуються конструкторів і деструкторів. По-перше, коли викликаються конструктори й деструктори базового й похідного класів? По-друге, як передаються параметри конструкторів базового класу? Відповіді на ці питання втримуються в наступному розділі.

Коли викликаються конструктори й деструктори

Базовий і похідний клас можуть містити декілька конструкторів і деструктор. Отже, дуже важливо правильно розуміти, у якому порядку вони викликаються при створенні й знищенні об'єктів похідного класу. Для початку розглянемо наступний приклад.

```
#include <iostream>
using namespace std;
class base {
public:
	base() { cout << "Створюється об'єкт класу base\n"; }
	~base() { cout << "Знищується об'єкт класу base\n"; }
};
class derived : public base {
public:
	derived() { cout << "Створюється об'єкт класу derived\n"; }
	~derived() { cout << "Знищується об'єкт класу derived\n"; }
};

int main() {
	derived ob;
	// Крім створення й знищення об'єкта, нічого не відбувається
	return 0;
}
```

Як зазначено в коментарі до функції main(), програма просто створює, а потім знищує об'єкт **ob** класу **derived**. У ході виконання програма виводить на екран наступні повідомлення.

Створення об'єкта класу base
Створення об'єкта класу derived
Знищення об'єкта класу derived
Знищення об'єкта класу base

Як бачимо, спочатку викликається конструктор базового класу, а потім - похідного. Після цього, оскільки об'єкт **ob** негайно знищується, викликається деструктор класу **derived**, а за ним - деструктор класу **base**.

Результати цього експерименту можна узагальнити. При створенні об'єкта похідного класу спочатку викликається конструктор базового класу, а потім - похідного. При знищенні об'єкта похідного класу спочатку викликається деструктор похідного класу, а потім - базового. Інакше кажучи, конструктори викликаються в ієрархічному порядку, а деструктори - у зворотному.

Це цілком природно. Оскільки базовий клас не має ніякої інформації про похідні класи, ініціалізація його об'єктів повинна виконуватися до ініціалізації будь-якого об'єкта похідного класу. Отже, конструктор базового класу повинен викликатися першим.

Цілком очевидно, що деструктори повинні викликатися у зворотному порядку. Оскільки похідний клас успадковує властивості базового, знищення об'єкта базового класу викличе знищення об'єкта похідного класу. Отже, деструктор похідного класу повинен викликатися до повного знищення об'єкта.

При ієрархічному спадкуванні (коли похідний клас стає базовим для свого спадкоємця) застосовується наступне правило: конструктори викликаються в ієрархічному порядку, а деструктори - у зворотному. Розглянемо приклад.

```
#include <iostream>
using namespace std;
class base {
public:
	base() { cout << "Створення об'єкта класу base\n"; }
```

```
        ~base() { cout << "Знищення об'єкта класу base\n"; }
};
class derived1 : public base {
public:
        derived1() { cout << "Створення об'єкта класу derived1\n"; }
        ~derived1() { cout << "Знищення об'єкта класу derived1\n"; }
};
class derived2 : public derived1 {
public:
        derived2() { cout << "Створення об'єкта класу derived2\n"; }
        ~derived2() { cout << "Знищення об'єкта класу derived2\n"; }
};

int main() {
        derived2 ob;
        // Створюємо й знищуємо об'єкт ob
        return 0;
}
```

У результаті на екран виводяться наступні рядки.
Створення об'єкта класу base
Створення об'єкта класу derived1
Створення об'єкта класу derived2
Знищення об'єкта класу derived2
Знищення об'єкта класу derived1
Знищення об'єкта класу base

Це правило застосовне й до множинного спадкування. Розглянемо наступну програму.

```
#include <iostream>
using namespace std;
class base1 {
public:
        base1() { cout << "Створення об'єкта класу base1\n"; }
        ~base1() { cout << "Знищення об'єкта класу base1\n"; }
};
class base2 {
public:
        base2() { cout << " Створення об'єкта класу base2\n"; }
        ~base2() { cout << " Створення об'єкта класу base2\n"; }
};
class derived : public base1, public base2(
public:
        derived() { cout << " Створення об'єкта класу derived\n"; }
        ~derived)) { cout << " Знищення об'єкта класу derived\n"; }
};
int main() {
        derived ob;
        // Створення й знищення об'єкта ob
        return 0;
}
```

Ця програма видає на екран наступні повідомлення.
Створення об'єкта класу base1
Створення об'єкта класу base2
Створення об'єкта класу derived
Знищення об'єкта класу derived
Знищення об'єкта класу base2
Знищення об'єкта класу base1

Як бачимо, і в цьому випадку конструктори викликаються в ієрархічному порядку, з ліва на право, як зазначено в списку спадкування класу **derived**. Деструктори викликаються у зворотному порядку, з права на ліво. Допустимо, що ім'я **base2** зазначене в списку спадкування класу derived перед іменем **base1**.

```
class  derived : public base2, public base1 {
```

Тоді результати роботи програми виглядали б так.
Створення об'єкта класу base2
Створення об'єкта класу basel
Створення об'єкта класу derived
Знищення об'єкта класу derived
Знищення об'єкта класу basel
Знищення об'єкта класу base2

Передача параметрів конструкторові базового класу

Дотепер ми розглядали конструктори, що не мають аргументів. Якщо конструктор похідного класу повинен одержувати кілька параметрів, слід просто використовувати стандартну синтаксичну форму конструктора з параметрами. Для цього застосовується розширена форма оголошення конструктора похідного класу, яка дозволяє передавати аргументи декільком конструкторам одного або декількох базових класів. Загальна форма цієї синтаксичної конструкції така.

конструктор _ похідного-класу(список_аргументів) :base1(список_аргументів) ,
base2(список_аргументів) ,
...
basen(список_аргументів)
{
// Тіло конструктора похідного класу
}

Тут параметри **base1-basen** є іменами базових класів. Зверніть увагу на те, що оголошення конструктора похідного класу відділяється двокрапкою від специфікацій базових класів, які, у свою чергу, розділяються комами. Розглянемо наступну програму.

```
#include <iostream>
using namespace std;
class base {
protected: int i;
public:
	base(int x) {
		i = x;
		cout << "Створення об'єкта класу base\n";
	}
	~base() { cout << "Знищення об'єкта класу base\n"; }
};
class derived : public base {
	int j; public:
		// Клас derived використовує змінну x;
		// змінна y передається базовому класу.
		derived(int x, int y) : base(y)
		{
			j = x; cout << "Створення об'єкта класу derived\n";
		}

		~derived() { cout << "Знищення об'єкта класу derived\n"; }
		void show() { cout << i << " " << j << "\n"; }
};
int main() {
	derived ob(3, 4);
		ob.show(); // Виводить на екран числа 4 3
	return 0;
}
```

Тут конструктор класу **derived** має два параметри: **x** и **y**. Однак у самому конструкторові використовується лише змінна **x**, а змінна **y** передається конструкторові базового класу. Як правило, у конструкторові похідного класу повинні оголошуватися всі параметри, необхідні базовому класу Для цього вони вказуються після двокрапки в списку аргументів конструктора базового класу.

Розглянемо приклад множинного спадкування.

```
#include <iostream>
using namespace std;
class base1 {
protected: int i;
public:
	base1(int x) {
		i = x;
		cout << "Створення об'єкта класу basel\n";
	}
	~base1() { cout << "Знищення об'єкта класу basel\n"; }
};
class base2 {
protected: int k;
public:
	base2(int x) {
		k = x;
		cout << "Створення об'єкта класу base2\n";
	}
	~base2() { cout << "Знищення об'єкта класу base2\n"; }
};
class derived : public base1, public base2 {
	int j; public:
		derived(int x, int y, int z) : base1(y), base2(z)
		{
			j = x; cout << "Створення об'єкта класу derived\n";
		}
		~derived() { cout << "Знищення об'єкта класу derived\n"; }
		void show() { cout << i << " " << j << " " << k << "\n"; }
};
int main() {
	derived ob(3, 4, 5);
	ob.show(); // Виводить на екран числа 4 3 5
	return 0;
}
```

Підкреслимо, що аргументи конструктора базового класу передаються за допомогою аргументів конструктора похідного класу. Отже, навіть якщо конструктор похідного класу не має власних аргументів, його оголошення повинне містити аргументи конструкторів базових класів. У цьому випадку аргументи, передані конструкторові похідного класу, просто переправляються конструкторам базових класів. Наприклад, у розглянутій нижче програмі конструктор класу **derived** не має власних аргументів, а конструктори класу **base1** і **base2**, навпаки, мають по одному параметру.

```
#include <iostream>
using namespace std;
class base1 {
protected: int i;
public:
	base1(int x) { i = x; cout << "Створення об'єкта класу basel\n"; }
	~base1() { cout << "Знищення об'єкта класу basel\n"; }
};
class base2 {
protected: int k;
public:
	base2(int x) { k = x; cout << "Створення об'єкта класу base2\n"; }
	~base2() { cout << "Знищення об'єкта класу base2\n"; }
};
class derived : public base1, public base2 {
public:
	/* Конструктор класу derived не має параметрів, у його оголошенні вказуються параметри
конструкторів
	базових класів. */
	derived(int x, int y) : base1(x), base2(y)
	{
		cout << "Створення об'єкта класу derived\n";
	}
	~derived() { cout << "Знищення об'єкта класу derived\n"; }
```

```
        void show() { cout << i << " " << k << "\n"; }
};
int main() {
        derived ob(3, 4);
        ob.show(); // Виводить на екран числа 3 4
        return 0;
}
```

Конструктор похідного класу може довільно використовувати всі параметри, зазначені в його оголошенні, навіть якщо вони передаються конструкторам базового класу. Інакше кажучи, передача параметрів конструкторам базових класів не виключає їхній використання усередині похідного класу. Таким чином, фрагмент програми, наведений нижче, є абсолютно правильним.

```
class derived: public base { int j;
public:         // Клас derived використовує обоє параметра x і y,
                // а потім передає їхньому конструкторові базового класу.
derived(int x, int y): base(x, y) { j = x*y; cout << "Створення об'єкта класу derived\n"; }
```

Передаючи параметри конструкторам базових класів, слід мати на увазі, що в якості аргументу можуть використовуватися будь-які припустимі вирази, наприклад, виклики функцій або змінні. Це повністю узгодиться із принципом динамічної ініціалізації об'єктів, передбаченої в мові C++.

Надання доступу

Якщо до базового класу застосовується механізм закритого спадкування, усі його відкриті й захищені члени стають закритими членами похідного класу. Однак у деяких ситуаціях можна відновити вихідний статус одного або декількох успадкованих членів, які раніше були відкритими або захищеними. Наприклад, може виникнути необхідність зберегти відкритий доступ до деяких членів базового класу, незважаючи на те що вони успадковуються закритим похідним класом. Стандарт мови C++ передбачає для цього два шляхи. По-перше, можна застосувати оператор **using**. Цей спосіб більш кращий. Оператор **using** призначений для підтримки просторів іме. По-друге, можна використовувати оголошення рівня доступу (access declaration) у похідному класі. Цей спосіб також підтримується стандартом мови C++, однак уважається небажаним. Це значить, що в нових програмах його слід уникати. Однак, оскільки в побуті залишається велика кількість старих програм, розглянемо цей приклад докладніше.
Оголошення рівня доступу виглядає в такий спосіб.

```
базовий_клас::член;
```

Це оголошення розміщається усередині похідного класу після заголовка відповідного розділу. Зверніть увагу на те, що тип змінної в оголошенні рівня доступу вказувати не слід.

```
class base {
public:
        int j; // Відкритий член класу base
};
// Закритий спадкоємець класу base,
class derived : private base {
public:
        // Місце для оголошення рівня доступу.
        base::j; // Тепер змінна j знову відкрита.
};
```

Оскільки клас **derived** є закритим спадкоємцем класу **base**, відкрита змінна-член **j** із класу **base** стає закритою змінною-членом класу **derived**. Однак у розділ **public** класу **derived** ми помістили оголошення рівня доступу **base::j**; Змінна **j** знову стала відкритою.

Цей спосіб можна застосовувати для відновлення статусу відкритих і захищених членів. Однак з його допомогою не можна підняти або понизити рівень доступу до члена класу. Наприклад, член, оголошений закритим у базовому класі, не можна зробити відкритим у похідному. (Якби це було можливо, механізм інкапсуляції був би повністю зруйнований!)
Наступна програма ілюструє оголошення рівня доступу. Зверніть увагу на те, як відновлюється відкритий

статус членів **j**, **seti()** і **geti().**

```
#include <iostream> using namespace  std;
class base {
	int i; // Закритий член класу base
public:
	int j, k;
	void seti(int x) { i = x; }
	int geti() { return i; }
};
// Закритий спадкоємець класу base,
class derived : private base {
public:
	/* Наступні три оператори відновлюють
	відкритий статус членів j, seti() і geti(). */
	base::j; // Змінна j знову відкрита, а змінна k  ні.
	base::seti; // Функція seti() знову відкрита.
	base::geti; // Функція geti() знову відкрита.
			// base::i; // Невірно, рівень доступу піднімати не можна.
	int a; // Відкритий член. };
	int main() {
		derived ob;
		//ob.i =10; // Не можна, тому що змінна i
		// є закритим членом класу derived.
		ob.j = 20; // Можна, тому що змінна j відкрита в класі derived,
				//ob.k = 30; // Не можна, оскільки змінна k є
				// закритим членом класу derived.
		ob.a = 40; // Можна, тому що змінна a є відкритим
				// членом класу derived, ob.seti(10);
		cout << ob.geti() << " " << ob.j << " " << ob.a;
		return 0;
	}
```

Цей спосіб дозволяє відновлювати рівень доступу до деяких відкритих або захищеним членам класу, залишаючи похідний клас закритим.

Незважаючи на те що мова C++ допускає оголошення рівня доступу, вони вважаються небажаними. Це означає, що при створенні нових програм замість їх слід застосовувати оператор **using**, що забезпечує той же результат.

Розглянемо ще один приклад множинного відкритого успадкування.

```
#include <iostream>
using namespace std;
class TInner
{
public:
	int i;
	TInner(int n) :i(n) { cout<<"Ctor TInner\n"; }
	TInner(TInner& x) { *this = x; cout<<"Copy ctor TInner\n"; }
	~TInner() { cout << "Dtor TInner\n"; }
};
class TBase
{
public:
	double a;
	TInner b;
	TBase() :b(10), a(5.0) { cout << "Ctor TBase\n"; }
	~TBase() { cout << "Dtor TBase\n"; }
	TBase(TBase& x) :b(10) { *this = x; cout << "Copy ctor TBase\n"; }
	void printBase() { cout << "TBase::TInner::i = "<< b.i<<" a = "<<a<<endl; }
};
class TDerived : public TBase
{
public:
	char c;
	TDerived() :c('Z') { cout << "Ctor TDerived\n"; }
	TDerived(TDerived& x) { *this = x; cout << "Copy ctor TDerived\n"; }
	~TDerived() { cout << "Dtor TDerived\n"; }
	void printDerived() { cout << "TDerived::TInner::i = " << b.i << " c = " << c << endl;
```

```
        }
};
class TDerived2 : public TDerived
{
public:
        float f;
        TDerived2() :f(10.0) { cout << "Ctor TDerived2\n"; }
        ~TDerived2() { cout << "Dtor TDerived2\n"; }
        void printDerived2() { cout << "TDerived2::TInner::i = " << b.i << " f = " << f << endl;
        }
};
int main()
{
        TBase first;
        TDerived second;
        TDerived2 third;
        first.printBase();
        second.printDerived();
        third.printDerived2();
        cout << "Sizeof(TInner) = " << sizeof(TInner)<<endl;
        cout << "Sizeof(TBase) = " << sizeof(TBase)<<endl;
        cout << "Sizeof(TDerived) = " << sizeof(TDerived)<<endl;
        cout << "Sizeof(Tderived2) = " << sizeof(TDerived2)<<endl;


        cout << endl;

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

У цій програмі описано чотири класи: TInner, TBase, TDerived, TDerived2. Клас TBase є контейнерним; він містить об'єкт класу TInner. Крім того, клас TBase є базовим стосовно класу TDerived. У свою чергу, клас TDerived є базовим стосовно класу TDerived2.

У функції main() створюється три об'єкти — first, second і third — класів TBase, TDerived і TDerived2 відповідно. Потім викликаються функції, що виводять на екран вміст цих об'єктів.

Проаналізуємо результати роботи цієї програми (номери не є частиною виводу — вони додані для зручності). Програма виконувалася під керуванням 32-розрядної операційної системи.

1. Ctor TInner
2. Ctor TBase
3. Ctor TInner
4. Ctor TBase
5. Ctor TDerived
6. Ctor TInner
7. Ctor TBase
8. Ctor TDerived
9. Ctor TDerived2
10. TBase::TInner::i = 10 a = 5
11. TDerived::TInner::i = 10 c = Z
12. TDerived2::TInner::i = 10 f = 10
13. Sizeof(TInner) = 4
14. Sizeof(TBase) = 16
15. Sizeof(TDerived) = 24
16. Sizeof(Tderived2) = 32
17.
18. Press any key to continue . . .
19. Dtor TDerived2
20. Dtor TDerived
21. Dtor TBase
22. Dtor TInner
23. Dtor TDerived
24. Dtor TBase
25. Dtor TInner
26. Dtor TBase
27. Dtor TInner

У рядку 1 ми бачимо повідомлення конструктора класу `TInner`, що генерується при створенні об'єкта `first`. Повідомлення про створення об'єкта `first` міститься в другому рядку.

Рядки 3–5 містять повідомлення від конструкторів, що створюють об'єкт `second` — спочатку найбільше глибоко вкладений об'єкт класу `TInner`, потім об'єкт контейнерного класу `TBase` і, нарешті, об'єкт похідного класу `TDerived`.

Аналогічно в рядках 6–9 наведені повідомлення від конструкторів `TInner`, `TBase`, `TDerived` і `TDerived2`, що послідовно викликаються при створенні об'єкта `third`.

Рядки 10–12 являють собою результат роботи функцій-членів `printBase`(), `printDerived`() і `PrintDerived2`().

Рядки 13–16 демонструють розміри класів. Як бачимо, хоча члени класу не копіюються (інакше ми одержали би повідомлення від конструктора копіювання), розміри класу враховують розміри успадкованих полів. Крім того, необхідно помітити, що ці розміри обчислюються з урахуванням вирівнювання, тому, наприклад, розмір класу `TBase` дорівнює 16 байт, хоча розмір поля типу `double` дорівнює 9 байт, а розмір класу `TInner` — 4 байт.

Рядки 17-18 результат виконання команди `cout << endl; system("pause");`

У рядках 19–27 містяться повідомлення від деструкторів. Вони наочно показують, що деструктори об'єктів викликаються в зворотному порядку.

І головне: класи `TDerived` і `TDerived2` створені за допомогою відкритого копіювання. Це означає, що всі поля їх базових класів зберігають свій рівень доступу.

Відзначимо, що в нашій програмі всі члени класів поміщені у відкритих розділах. Як правило, усе навпаки — дані необхідно ховати. Якби ми підкорилися цій вимозі і помістили змінні a і b у закритий розділ класу `TBase`, вони були б сховані від класу `TDerived`. Інакше кажучи, хоча закриті члени класу `TBase` успадковуються класом `TDerived`, функції-члени похідного класу не мають прямого доступу до закритих полів — інкапсуляція продовжує діяти! Об'єкти класу `TDerived` одержують поля класу `TBase` "в упакуванні", і звертатися до них можуть лише через функції відкритого інтерфейсу. Таким чином, закриті члени базового класу не рівноцінні закритим членам похідного класу. Вони доступні лише членам базового класу. Що ж у такому випадку означає вираження члени базового класу автоматично стають членами похідного класу? Повною мірою це відноситься лише до відкритих і захищених об'єктів, інакше успадкування вступило б у протиріччя з принципом приховання інформації.

До речі, відкрите успадкування дозволяє обійтися без створення об'єктів класів `TBase` і `TDerived`. Звернутися до функцій-членів базових класів можна безпосередньо через об'єкт похідного класу `TDerived2`.

```
TDerived2 obj;
obj.printBase();
obj.printDerived();
obj.printDerived2();
```

Перевизначення методів батьківського класу

Виклик методів батьківського класу

При виклику методу через об'єкт дочірнього класу, компілятор спочатку дивиться, чи існує цей метод в дочірньому класі. Якщо ні, то він починає просуватися по "ланцюжку" спадкування вгору і перевіряє, чи був цей метод визначений в будь-якому з батьківських класів. Компілятор використовуватиме перше знайдене визначення. Наприклад:

```
#include <iostream>

class Parent
{
protected:
    int m_value;

public:
    Parent(int value)
        : m_value(value)
    {
    }

    void identify() { std::cout << "I am a Parent!\n"; }
};
```

```
class Child : public Parent
{
public:
    Child(int value)
        : Parent(value)
    {
    }
};

int main()
{
    Parent parent(6);
    parent.identify();

    Child child(8);
    child.identify();

    return 0;
}
```

Результат виконання програми:

```
I am a Parent!
I am a Parent!
```

При виклику child.identify(), компілятор дивиться, чи визначений метод identify() в класі Child. Ні, тому компілятор переходить до класу Parent. У класі Parent є визначення методу identify(), тому компілятор використовує саме це визначення.

Перевизначення методів батьківського класу

Однак, якби ми визначили метод identify() в класі Child, то використовувалося б саме це визначення. Це означає, що ми можемо змусити батьківські методи працювати по-іншому з нашими дочірніми класами, просто перевизначаючи їх в дочірніх класах!

Вищенаведений приклад стане кращим, якщо child.identify() виводитиме I am a Child!. Давайте змінимо метод identify() в класі Child так, щоб він повертав правильну відповідь.

Перевизначення батьківського методу в дочірньому класі відбувається, як звичайне визначення методу:

```
class Child : public Parent
{
public:
    Child(int value)
        : Parent(value)
    {
    }

    int getValue() { return m_value; }

    // Ось наш змінюваний метод батьківського класу
    void identify() { std::cout << "I am a Child!\n"; }
};
```

Ось той же код функції main(), що і у вищенаведеному прикладі, але вже з внесеними змінами в клас Child:

```
int main()
{
    Parent parent(6);
    parent.identify();

    Child child(8);
    child.identify();

    return 0;
}
```

Результат:

```
I am a Parent!
I am a Child!
```

Зверніть увагу, коли ми перевизначаємо батьківський метод в дочірньому класі, то дочірній метод не наслідує специфікатор доступу батьківського методу з тим же ім'ям. Використовується той специфікатор доступу, який вказаний в дочірньому класі. Таким чином, метод, визначений як private в батьківському класі, може бути перевизначений як public в дочірньому класі, або навпаки!

```
#include <iostream>

class Parent
{
private:
	void print()
	{
		std::cout << "Parent!";
	}
};

class Child : public Parent
{
public:
	void print()
	{
		std::cout << "Child!";
	}

};


int main()
{
	Child child;
	child.print(); // виклик child::print(), який є public
	return 0;

}
```

Розширення функціоналу батьківських методів

Можуть бути випадки, коли нам не потрібно повністю замінювати метод батьківського класу, але потрібно просто розширити його функціонал. Зверніть увагу, у вищенаведеному прикладі метод Child::identify() повністю перекриває Parent::identify()! Можливо, це не те, що нам потрібно. Ми можемо викликати метод батьківського класу з тим же ім'ям в методі дочірнього класу (для повторного використання коду), а потім додати свій код.

Щоб метод дочірнього класу викликав метод батьківського класу з тим же ім'ям, потрібно просто виконати звичайний виклик функції, але з доданням імені батьківського класу і оператора дозволу області видимості. У наступному прикладі ми виконаємо перевизначення identify() в класі Child, викликаючи спочатку Parent::identify(), а потім додаючи вже свій код:

```
class Child : public Parent
{
public:
    Child(int value)
        : Parent(value)
    {
    }

    int GetValue() { return m_value; }

    void identify()
    {
        Parent::identify(); // спочатку виконується виклик Parent::identify()
        std::cout << "I am a Child!\n"; // потім вже виводиться цей текст
    }
};
```

Разом з:

```
int main()
{
    Parent parent(6);
    parent.identify();

    Child child(8);
    child.identify();

    return 0;
}
```

Дає результат:

```
I am a Parent!
I am a Parent!
I am a Child!
```

При виконанні child.identify() виконується виклик Child::identify(). У Child::identify() ми спочатку викликаємо Parent::identify(), який виводить I am a Parent!. Коли Parent::identify() завершує своє виконання, Child::identify() продовжує своє виконання і виводить I am a Child!.

Все просто. Навіщо тоді потрібно використовувати оператор дозволу області видимості (::)? А потім, що, якби ми визначили Child::identify() наступним чином:

```
class Child : public Parent
{
public:
    Child(int value)
        : Parent(value)
    {
    }

    int GetValue() { return m_value; }

    void identify()
    {
        identify(); // немає оператора дозволу області видимості!
        std::cout << "I am a Child!";
    }
};
```

То виклик методу identify() без вказування оператора дозволу області видимості призвів би до виклику identify() в поточному класі, тобто Child::identify(). Потім знову виклик Child::identify(), і ура — у нас вийшов нескінченний цикл. Тому використання оператора дозволу області видимості є обов'язковою умовою при зміні методів батьківського класу.

Приховування методів батьківського класу

Мова C++ надає можливість змінити специфікатор доступу батьківського члена в дочірньому класі. Це робиться за допомогою **"using-оголошення"**. Наприклад, розглянемо наступний клас Parent:

```
#include <iostream>

class Parent
{
private:
    int m_value;

public:
    Parent(int value)
        : m_value(value)
    {
    }

protected:
    void printValue() { std::cout << m_value; }
};
```

Оскільки Parent::printValue() оголошений як **protected**, то він доступний тільки іншим членам Parent і своїм дочірнім класам. Для інших об’єктів доступ до нього закритий.

Визначимо клас Child, який змінює специфікатор доступу printValue() з protected на public:

```
class Child : public Parent
{
public:
    Child(int value)
        : Parent(value)
    {
    }

    // Parent::printValue є protected, тому доступ до нього не є відкритим для всіх об'єктів.
    // Але ми можемо це виправити за допомогою "using-оголошення"
    using Parent::printValue; // зверніть увагу, тут немає ніяких дужок
};
```

Це означає, що наступний код виконається без помилок:

```
int main()
{
    Child child(9);

    // Метод printValue() є public в класі Child, тому все добре
    child.printValue(); // виведеться 9
    return 0;
}
```

Тут є дві примітки:

По-перше, ви можете змінити специфікатори доступу тільки для тих членів батьківського класу, до яких є доступ у дочірнього класу. Ви не зможете змінити специфікатор доступу члена батьківського класу з private на protected або public, оскільки дочірній клас не має доступу до private-членів батьківського класу.

По-друге, починаючи з C++11 використання “using-оголошення” є кращим способом зміни специфікаторів доступу. Однак ви також можете використовувати “access-оголошення”. Це працює ідентично “using-оголошенню”, тільки без ключового слова using. Зараз цей спосіб вважається застарілим, але, переглядаючи старий код, ви можете побачити “access-оголошення”, тому про це варто знати.

Приховування батьківських методів в дочірньому класі

У мові C++ неможливо видалити або обмежити функціонал батьківського класу, крім як за допомогою безпосередньої зміни вихідного коду. Однак в дочірньому класі ви можете приховати функціонал, який існує в батьківському класі, так щоб до нього не можна було отримати доступ через об’єкти дочірнього класу. Це робиться шляхом зміни відповідних специфікаторів доступу.

Наприклад, ви можете зробити закритим відкритий член батьківського класу:

```
#include <iostream>

class Parent
{
public:
	int m_value;
};

class Child : public Parent
{
private:
	using Parent::m_value;

public:
	Child(int value)
		// Ми не можемо ініціалізувати m_value, оскільки це член класу Parent (Parent
повинен ініціалізувати m_value)
	{
		// Але ми можемо присвоїти значення
		m_value = value;
	}
};
```

```
int main()
{
	Child child(9);

	// Наступне не спрацює, оскільки m_value був перевизначений як private
	std::cout << child.m_value;

	return 0;
}
```

Це дозволяє інкапсулювати дані батьківського класу в дочірньому класі. В якості альтернативи можна використати спадкування типу private, що призведе до того, що всі успадковані public- і protected-члени класу Parent стануть private в класі Child.

Ви також можете закрити батьківські методи в дочірньому класі, використовуючи **ключове слово delete**:

```
#include <iostream>

class Parent
{
private:
	int m_value;

public:
	Parent(int value)
		: m_value(value)
	{
	}

	int getValue() { return m_value; }
};

class Child : public Parent
{
public:
	Child(int value)
		: Parent(value)
	{
	}


	int getValue() = delete; // робимо цей метод недоступним
};

int main()
{
	Child child(9);

	// Наступне не спрацює, тому що getValue() видалений
	std::cout << child.getValue();

	return 0;
}
```

Таким чином, компілятор скаржитиметься на нашу спробу виклику методу getValue() через об’єкт класу Child. Однак через об’єкт батьківського класу все працюватиме, тому що ми «видалили» getValue() тільки в дочірньому класі.

Перевантаження операторів з наслідуванням

Завдання перевизначити оператор для базового класу, щоб він працював і для похідного

```
class  base
{protected:
    int a;

public:
    base(int x=0):a(x) {}
```

```
    base operator +(const base &f)
     {return base(a+f.a);}
};
```

Перевантажуємо оператор плюс. І тепер створимо похідний клас

```
class first : public base
{public:
    first(int q=0):base(q){}
    void display(){cout<<a;}
};
```

На перший погляд виглядає все ок, спробуємо створити main()

```
int main()
{
    first w(4),w1(5), w2;
    w2=w+w1;
    w2.display();

    return 0;
}
```

І тут починаються проблеми,

```
w2=w+w1;
```

w1 + w2 обидва first неявно піднімаються до base Явище **upcasting** — це **перетворення від похідного класу до базового**.

результат тип base не може неявно "опуститися" до first

upcast vs. downcast:

Upcast (first → base) — завжди безпечно, дозволено неявно

Downcast (base → first) — небезпечно, дозволено лише через:

конструктор

явне приведення типу (static_cast<first>(...))

або динамічне приведення (dynamic_cast)

Найпростіше, це зробити конструктор, який би ініціалізовував конструктор похідного класу через конструктор базового класу.

Додати в first:

```
first(const base& b) : base(b) {}
```

Або використати static_cast:

```
first w3 = static_cast<first>(w1 + w2);
```

Кінцевий код.

```
#include <iostream>

using namespace std;

class  base
{protected:
    int a;

public:
    base(int x=0):a(x) {}

    base operator +(const base &f)
     {return base(a+f.a);}
```

```
};


class first : public base
{public:
    first(int q=0):base(q){}
    first (const base& q):base(q){}
   void display(){cout<<a;}
};

int main()
{
    first w(4),w1(5), w2;
    w2=w+w1;
    w2.display();

    return 0;
}
```