Лекція 3

1. Дружні функції

За допомогою ключового слова **friend** можна надати звичайній функції доступ до закритих членів класу. Дружня функція має доступ до всіх закритих і захищених членів класу. Для того щоб оголосити дружню функцію, слід помістити її прототип усередині класу, указавши перед нею ключове слово **friend**. Розглянемо приклад.

```
#include <iostream>
#include "windows.h"
using namespace std;
class myclass {
	int a, b;
public:
	friend int sum(myclass x);
	void set_ab(int i, int j);
};
void myclass::set_ab(int i,int j) {
	a = i;
	b = j;
}
// Увага: функція sum() не є функцією- членом ніякого класу,
int sum(myclass x) {
	/* Тому що функція sum() є дружньою повідношенню до класу myclass, вона має прямий
доступдо змінних a й b. */
	return x.a + x.b;
}
int main() {
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);
	myclass n;
	n.set_ab(3, 4);
	cout << sum(n);
	cout << endl;
	system("pause");
	return 0;
    }
```

У даному прикладі функція **sum()** не є членом класу **myclass**. Однак вона має повний доступ до закритих членів. Крім того, функцію **sum()** можна викликати без допомоги оператора ".". Оскільки вона не є членом класу, їй не потрібно вказувати ім'я об'єкта.

Хоча дружні функції мають рівні права зі членами класу, у деяких ситуаціях вони бувають особливо корисні. По-перше, дружні функції дозволяють перевантажувати деякі види операторів. По-друге, вони полегшують створення деяких функцій уведення-виведення. По-третє, дружні функції корисні в ситуаціях, коли кілька класів можуть містити члени, тісно пов'язані з іншими частинами програми. Розглянемо останнє твердження докладніше.

Уявіть собі два класи, кожний з яких виводить на екран повідомлення про помилку. Інша частина програми повинна перевіряти, відображається чи в цей момент на екрані повідомлення про помилку, щоб не ушкодити його при виведенні своєї інформації. Цю проблему можна розв'язати, помістивши в кожний клас функцію-член, що повертає індикатор активного повідомлення.

Однак для перевірки цієї умови потрібно виконати додаткову роботу, тобто викликати дві функції, а не одну. Якщо умову потрібно перевірити швидко, додаткові витрати часу можуть виявитися неприйнятними. Однак, якщо оголосити функцію, дружню стосовно кожного із класів, можна перевірити стан обох об'єктів за допомогою однієї функції. У таких ситуаціях дружні функції дозволяють створювати більш ефективний код. Проілюструємо це твердження наступною програмою.

```
#include <iostream>
#include "windows.h"
using namespace std;
const int IDLE = 0;
const int INUSE = 1;
class C2;  // неповне оголошення
class C1 {
	int status;   // Рівно IDLE, якщо екран вільний,
				  // і INUSE, якщо екран зайнятий.
				  // . . .
public:
	void set_status(int state);
	friend int idle(C1 a, C2 b);
};
class C2 {
	int status;   // Рівно IDLE, якщо екран зайнятий,
				  //і INUSE, якщо екран вільний.
				  // . . .
public:
	void set_Status(int state);
	friend int idle(C1 a, C2 b);
};
void C1::set_status(int state) { status = state; }
void C2::set_Status(int state) { status = state; }
int idle(C1 a, C2 b)
{
	if(a.status || b.status) return 0; else return 1;
}
int main()
{
	C1 x;
	C2 y;
	x.set_status(IDLE); y.set_Status(IDLE);
	if(idle(x, y)) cout << "Екран вільний.\n"; else cout << "Екран зайнятий.\n";
	x.set_status(INUSE);
	if(idle(x, y)) cout << "Екран вільний.\n"; else cout << "Екран зайнятий.\n";
	return 0;
    }
```

Зверніть увагу на те, що в цій програмі використовується неповне оголошення (forward declaration) класу C2. Це необхідно, оскільки оголошення функції idle() в класі C1 посилається на клас C2, який ще не оголошений. Щоб здійснити неповне оголошення класу, можна просто застосувати спосіб, продемонстрований у програмі.

Дружня функція може бути членом іншого класу. Розглянемо варіант нашої програми, включивши функцію idle() в клас C1.

```
#include <iostream>
#include "windows.h"
using namespace std;
const int IDLE = 0;
const int INUSE = 1;
class C2;  // Неповне оголошення
```

```
class C1 {
	int status;   // Рівно IDLE, якщо екран зайнятий,
                                // і INUSE, якщо вільний.
                                // ...
public:
	void set_status(int state);
	int idle(C2 b);
};// Тепер функція idle() - член класу C1.
	class C2 {
		int status;   // Рівно IDLE, якщо екран зайнятий,
                                      //і INUSE, якщо вільний.
                                      // . . .
	public:
		void set_status(int state);
		friend int C1::idle(C2 b);
	};
	void C1::set_status(int state) status = state;
	void C2::set_status(int state) status = state;
	// Функція idle() є членом класу C1 і другом класу C2
	int C1::idle(C2 b)
	{
		if(status || b.status) return 0;
else return 1;
	}
	int main() {
		C1 x;
		C2 y;
		x.set_status(IDLE);
		y.set_status(IDLE);
		if(x.idle(y)) cout << "Екран вільний.\n"; else cout << "Екран зайнятий.\n";
		x.set_status(INUSE);
		if(x.idle(y)) cout << "Екран вільний.\n"; else cout << "Екран зайнятий.\n";
		return 0;
	}
```

Оскільки функція idle() є членом класу C1, вона має прямий доступ до змінної **status**, що належить об'єкту класу C1. Таким чином, будь-який об'єкт класу C2 необхідно передавати у функцію idle ().

Дружні функції мають два важливі обмеження. По-перше, похідний клас не успадковує дружні функції. По-друге, дружні функції не можуть містити специфікатор зберігання, тобто в її оголошенні не можна використовувати ключові слова **static** або **extern**.

2. Дружні класи

Один клас може бути дружнім стосовно іншого. У цьому випадку дружній клас і всі його функції- члени мають доступ до закритих членів, визначених в іншому класі. Розглянемо приклад.

```
// Застосування дружніх класів
#include <iostream>

using namespace std;
class Twovalues {
	int a;
	int b;
public:
	void init(int i, int j) { a = i; b = j; }
	friend class Min;
};
class Min {
```

```
public:
	int min(Twovalues x);
};
int Min::min(Twovalues x) {
	return x.a < x.b ? x.a : x.b;
}
int main() {
	Twovalues ob;
	ob.init(10, 20);
	Min m;
	cout << m.min(ob);
	return 0;
}
```

У даному прикладі клас min має доступ до закритих змінних a й b, оголошеним у класі Twovalues.

Слід запам'ятати: якщо один клас є дружнім стосовно іншого, він просто одержує доступ до елементів, визначених у цьому класі, але не успадковує їх, тобто члени класу не стають членами дружнього класу.

Дружні класи рідко використовуються в практичних додатках. Вони необхідні лише в особливих випадках.

3. Функції, що підставляються (самостійно)

Мова C++ має важливу властивість: у ньому існують функції, що підставляються (inline functions), які широко використовуються в класах.

У мові C++ можна написати коротку функцію, яка не викликається, а підставляється у відповідне місце програми. Цей процес нагадує функціональну макропідстановку. Щоб замінити виклик функції підстановкою, перед її визначенням слід указати слово **inline**. Наприклад, у наступній програмі функція шах () не викликається, а підставляється.

```
#include <iostream>
using namespace std;
inline int max(int a, int b) {
return a>b ? a : b; }
int main() {
cout << max(10, 20);
cout << " " << max(99, 88);
return 0; }
```

З погляду компілятора ця програма виглядає так

```
#include <iostream>
using namespace std;
int main () {
cout << (10>20 ? 10 : 20);
cout << " " << (99>88 ? 99 : 88);
return 0; }
```

Функції, що підставляються, дозволяють створювати дуже ефективні програми Оскільки класи звичайно містять трохи інтерфейсних функцій, які найчастіше викликаються для доступу до його закритих членів, необхідно, щоб ці функції виконувалися якнайшвидше . Як відомо, кожний виклик функції сполучений з додатковими витратами на передачу й повернення керування. Звичайно при виклику функції її аргументи заносяться в стек, а вміст регістрів копіюється в оперативну пам'ять, щоб після повернення керування можна було відновити первісний стан програми. На ці операції затрачається додатковий час. Однак, якщо замість виклику тіло функції просто підставляється в програму, нічого цього не потрібно. На жаль, прискорення роботи програми досягається за рахунок збільшення розміру коду, оскільки тіло функції, що підставляється, дублюється певну кількість раз. Із

цієї причини функції, що підставляються, повинні бути дуже маленькими. Крім того, що підставляються слід робити тільки ті функції, швидкодію яких дійсно суттєво впливає на ефективність програми.

Як і специфікатор **register**, ключове слово **inline** є лише рекомендацією, а не наказом компіляторові. У деяких випадках компілятор може його проігнорувати. Крім того, деякі компілятори обмежують категорії функцій, які можуть бути такими, що підставляються. Зокрема , як правило, компілятори не дозволяють підставляти рекурсивні функції. У кожному конкретному випадку інформацію про обмеження на застосування функцій, що підставляються, слід шукати в документації, що супроводжує компілятор. Урахуйте, якщо функцію не можна підставити, вона буде викликатися.

Функції, що підставляються, можуть бути членами класу. Наприклад наступна програма уважається цілком коректною.

```
#include <iostream>
using namespace std;
class myclass {
int a, b;
public:
void init(int i, int j);
void show(); };
// Створити функцію, що підставляється.
inline void myclass::init(int i, int j) {
a = i;
b = j; }
// Створити іншу функцію, що підставляється.
inline void myclass::show()
{
cout << a << " " << b << "\n"; }
int main() {
myclass x;
x.init(10, 20); x.show();
return 0; }
```

4. Визначення функцій, що підставляються, усередині класу

Коротку функцію можна визначити безпосередньо усередині оголошення класу. Якщо функція визначена усередині оголошення класу, вона автоматично перетворюється в, ту що підставляється (якщо це не суперечить обмеженням компілятора). Указувати при цьому ключове слово inline в загальному не обов'язково, хоча помилкою це не вважається. Наприклад попередню програму можна переписати, помістивши визначення функцій init() і show() в оголошення класу myclass.

```
#include <iostream>
using namespace std;
class myclass {
int a, b;
public:
// Автоматична підстановка
void init(int i, int j ) { a=i; b=j; }
void showo { cout << a << " " << b << "\n"; } };
int main() {
myclass x;
x.init(10, 20); x.show();
return 0; }
```

Зверніть увагу на спосіб запису тіла функції усередині оголошення класу myclass. Оскільки функція, що підставляються звичайно коротка, таке приймання цілком типове. Однак зовсім необов'язково записувати функції саме так Наприклад, оголошення функції myclass можна переписати інакше.

```
#include <iostream>
```

```
using namespace std;
class myclass {
int a, b; public:
// Автоматична підстановка
void init(int i, int j)
{
a = i; b = j; }
void show() {
cout << a << " " << b << "\n"; } };
```

З технічної точки зору підстановка функції show() не має змісту, оскільки час, затрачуване на введення-виведення, набагато перевищує час, необхідний для виклику функції. Однак переважна більшість програмістів воліє поміщати всі короткі функції-члени усередині оголошення класу. (Украй рідко в професійних програмах можна зустріти короткі функції-члени, визначені поза оголошенням класу.

5. Статичні змінні-члени

Якщо перед оголошенням змінної-члена поставити ключове слово static, компілятор створить тільки один екземпляр цієї змінної, який буде використовуватися всіма об'єктами даного класу. На відміну від звичайних змінних-членів, статичні змінні-члени не компілюються для кожного об'єкта окремо. Незалежно від кількості об'єктів класу, статична змінна завжди існує в одному екземплярі. Таким чином, усі об'єкти даного класу використовують ту саму змінну. Усі статичні змінні ініціалізуються нулем ще до створення першого об'єкта класу.

Оголошення статичної змінної-члена в класі не означає її визначення (інакше кажучи, пам'ять для неї не виділяється). Щоб розмістити статичну змінну в пам'яті, слід визначити її поза класом, тобто глобально. Для цього в повторному оголошенні статичної змінної перед її іменем вказується ім'я класу, якому вона належить, і оператор дозволу області видимості. (Нагадаємо, що оголошення класу являє собою всього лише логічну схему, а не фізичну сутність.)

Щоб розібратися в механізмі використання статичних змінних-членів, розглянемо наступну програму.

```
#include <iostream>
using namespace std;
class shared {
	static int a;
	int b;
public:
	void set(int i, int j) {a = i; b = j;
}
void show(); };
int shared::a; // Визначаємо змінну a.
void shared::show() {
	cout << "Це статична змінна a: " << a;
cout << "\n це звичайна змінна b: " << b;
cout << "\n"; }
int main() {
	shared x, y;
	x.set(1, 1); // Присвоюємо змінній a значення 1
	x.show();
	y.set(2, 2); // Присвоюємо змінній b значення 2
	y.show();
	x.show(); /* Тут одночасно змінюється значення змінних-членів об'єктів x і b,
оскільки змінна a використовується обома об'єктами. */
return 0;
```

```
}
```

Результат роботи цієї програми такий.

Це статична змінна a: 1
Це звичайна змінна b: 1
Це статична змінна a: 2
Це звичайна змінна b: 2
Це статична змінна a: 1
Це звичайна змінна a: 1

Зверніть увагу на те, що цілочисельна змінна a оголошена як усередині класу shared, так і поза ним. Ми вже вказували, що це необхідно, оскільки оголошення змінної a усередині класу shared не супроводжується виділенням пам'яті.

Статична змінна-член виникає до створення першого об'єкта класу. Наприклад, у наступній короткій програмі змінна a одночасно є й відкритої, і статичною. Отже, функція main() має до неї прямий доступ. Оскільки змінна a виникає раніше об'єктів класу share, їй можна привласнити значення на самому початку програми. Як показано в програмі, при створенні об'єкта x значення змінної a не змінюється. Із цієї причини обидва оператора виведення відображають на екрані те саме значення - 99.

```
#include <iostream>
using namespace std;
class shared {
public:
	static int a;
};
int shared::a; // Визначення змінної a.
int main() {
	// Ініціалізація перед створенням об'єкта.
	shared::a = 99;
	cout << "Початкове значення змінної a: " << shared::a;
	cout << "\n";
	shared x;
	cout << "Значення змінної x.a: " << x.a;
	return 0;
}
```

Зверніть увагу на те, що при звертанні до змінної a необхідно вказувати ім'я класу, якому вона належить, і застосовувати оператор дозволу області видимості. Як правило, щоб звернутися до статичної змінної-члену незалежно від об'єкта, необхідно завжди вказувати ім'я класу, у якому вона оголошена.

Статичні змінні-члени дозволяють управляти доступом до ресурсів, які спільно використовуються всіма об'єктами класу. Наприклад, можна створити кілька об'єктів, що записують дані на жорсткий диск. Однак очевидно, що в кожний конкретний момент часу запис може робити лише один об'єкт. У таких ситуаціях слід оголосити статичну змінну, що служить індикатором. За значенням цієї змінної можна визначити, вільний файл або зайнятий. Кожний з об'єктів повинен аналізувати це значення перед початком запису. Розглянемо приклад, що ілюструє цю ситуацію.

```
#include <iostream>
using namespace std;
```

```
class cl {
	static int resource;
public:
	int get_resource();
	void free_resource() { resource = 0; }
};
int cl::resource; // Визначаємо ресурс.                                    .
int cl::get_resource() {
	if (resource) return 0; // Ресурс зайнятий,
	else {
		resource = 1;
		return 1;  // Ресурс наданий об'єкту.
	}
}
int main() {
	cl ob1, ob2;
	if (ob1.get_resource()) cout << "Об'єкт ob1 має ресурс\n";
	if (!ob2.get_resource()) cout << "Об'єкту ob2 доступ заборонений\n";
	ob1.free_resource(); // Звільняємо ресурс.
	if (ob2.get_resource())
		cout << "Об'єкт ob2 може використовувати ресурс\n";
	return 0;
    }
```

За допомогою статичної змінної можна також визначити кількість існуючих об'єктів конкретного класу. Розглянемо приклад.

```
#include <iostream>
using namespace std;
class Counter {
public:
	static int count;
	Counter() { count++; }
	~Counter() { count--; }
};
int Counter::count;
void f();
int main(void) {
	Counter o1;
	cout << "Існуючі об'єкти: ";
	cout << Counter::count << "\n";
	Counter o2;
	cout << "Існуючі об'єкти: ";
	cout << Counter::count << "\n";
	f();
	cout << "Існуючі об'єкти: ";
	cout << Counter::count << "\n";
	return 0;
}
void f() {
	Counter temp;
	cout << "Існуючі об'єкти: ";
	cout << Counter::count << "\n";
	// Після повернення з функції f() змінна temp знищується
    }
```

Ця програма виводить на екран наступні рядки.

Існуючі об'єкти: 1
Існуючі об'єкти: 2
Існуючі об'єкти: 3
Існуючі об'єкти: 2

Як бачимо, статична змінна-член count збільшується при кожному створенні об'єкта й зменшується при кожному знищенні. Отже, з її допомогою можна

відстежити кількість існуючих об'єктів класу Counter.

Статичні змінні-члени не заміняють собою глобальні. Однак глобальні змінні не відповідають духу об'єктно-орієнтованого програмування, оскільки вони порушують принципи інкапсуляції.

6. Статичні функції-члени

Функції-члени також можуть бути статичними, але на них поширюється кілька обмежень Вони мають прямий доступ тільки до інших статичних членів класу. (Зрозуміло, глобальні функції й дані також доступні статичним функціям-членам.) Статична функція-член не має покажчика **this**. Та сама функція не може одночасно мати статичну й нестатичну версії. Статичні функції не можуть бути віртуальними. І на закінчення, статичні функції не можна повідомляти за допомогою ключових слів **const** або **volatile**.

Розглянемо злегка змінену версію програми, що надає той самий ресурс різним об'єктам. Зверніть увагу на те, що функцію get_resource () можна викликати як незалежно від об'єкта, використовуючи ім'я класу й оператор дозволу області видимості, так і через об'єкт.

```
#include <iostream>
using namespace std;
class c1 {
	static int resource;
public:
	static int get_resource();
	void free_resource() { resource = 0; }
};
int c1::resource;// Визначення ресурсу.
int c1::get_resource()
{
	if (resource) return 0; // Ресурс занят.
	else {
		resource = 1;
		return 1;  // Ресурс наданий даному об'єкту.
	}
}
int main() {
	c1 ob1, ob2;
	/* Функція get_resource() є статичної,тому її можна викликати незалежно від об'єкта.
*/
	if (c1::get_resource()) cout << "Об'єкт ob1 має ресурс\n";
	if (!c1::get_resource()) cout << "Об'єкту ob2 доступ заборонений\n";
	ob1.free_resource();
	if (ob2.get_resource()) // can still call using object syntax
		cout << "Тепер об'єкт ob2 може використовувати ресурс\n";
	return 0;
    }
```

Статичні функції-члени мають обмежене коло застосувань, однак вони бувають корисні для попередньої ініціалізації закритих статичних змінних-членів до створення реальних об'єктів. Наприклад наступна програма абсолютно вірна.

```
#include <iostream>
using namespace std;
class static_type {
	static int i;
public:
```

```
        static void init(int x) { i = x; }
        void show() { cout << i; }
};
int static_type::i; // визначаємо змінну i
int main() {
        // Ініціалізуєм статичні дані перед створенням об'єкта
static_type::init(100);
        static_type x;
        x.show(); // Відображаємо число 100
        return 0;
    }
```

7. Оператор дозволу області видимості

Як відомо, оператор "::" зв'язує між собою імена класу і його члена, повідомляючи компіляторові, якому класу належить даний член. Однак оператор дозволу області видимості має ще одне призначення: він відкриває доступ до змінної, ім'я якої "замасковане" усередині вкладеної області видимості. Як приклад розглянемо наступний фрагмент програми.

```
int i;  // Глобальна змінна i
void f() {
        int i; // Локальна змінна i
        i = 10; // Використовуємо локальну змінну i
}
```

Як зазначено в коментарі, присвоювання i=10, ставиться до локальної змінної i. А що робити, якщо функції f () потрібний доступ до глобальної змінної i? Тоді слід застосувати оператор дозволу області видимості

```
int i;  // Глобальна змінна i.
void f() {
        int i; // Локальна змінна i.
        ::i = 10; // Звертання до глобальної змінної i
    }
```

8. Вкладені класи

Один клас можна визначити усередині іншого. У такий спосіб створюються вкладені класи. Оскільки оголошення класу фактично визначає його область видимості, вкладені класи можуть існувати тільки усередині області видимості зовнішнього класу. Взагалі вкладені класи використовуються рідко. Оскільки в мові C++ існує гнучкий і потужний механізм наслідування, необхідності створювати вкладені класи практично немає.

У загальному випадкові вкладені класи виглядають наступним чином:

```
class OuterClass {
        ...
                class NestedClass {
                ...
        }
};
```

Приклад:

```
#include <iostream>
using namespace std;
class A {
```

```
        A(int a, int b) : _a(a), _b(b) {}
        int _a, _b;
public:
        class B {
        public:
                void foo() {
                        A obj_A(12, 21);
                        cout << obj_A._a << "\t" << obj_A._b << endl;
                }
        };
};
int main() {
        A::B obj_B;
        obj_B.foo();
        cin.get();
        return 0;
}
```

Статичні вкладені класи, як і поля та методи класів, асоціюють із зовнішніми класами, у яких вони оголошені. Як і статичні методи, статичні класи не можуть отримувати прямий доступ до полів та методів екземплярів своїх зовнішніх класів: вони можуть використовувати їх тільки через посилання на об'єкти.

Внутрішні класи, як і поля та методи екземплярів класів, асоціюють із екземплярами зовнішніх класів. Внутрішні класи мають прямий доступ до полів та методів об'єктів своїх зовнішніх класів. Оскільки внутрішній клас асоційований із екземпляром класу, він не може бути статичним.

Вкладені класи використовують у наступних випадках:

- коли це — спосіб логічного групування класів, які використовують тільки разом. Якщо клас використовується іншим і тільки одним класом, то логічно вбудувати його у нього;

- збільшення рівня інкапсуляції. Нехай існують два класи верхнього рівня A та B. Нехай клас B вимагає доступу до поля класу A, яке оголошено як private. У такому випадкові доцільним є оголосити клас B вкладеним у A. Оскільки B буде членом A, то отримає доступ до членів A. Крім цього, B буде схованим від решти класів;

- спрощення підтримки коду та збільшення його читабельності. Є сенс малі класи вбудовувати у класи верхнього рівня, таким чином вони будуть розташовані ближче до місця їхнього використання.

Існують два спеціальні види внутрішніх класів: локальні класи (*local classses*) та анонімні класи (*anonymous classes*).

9. Локальні класи

Локальний клас – підвид внутрішнього класу. Локальними класами називають класи, оголошені усередині будь-якого блоку, позначеного фігурними дужками {}, який може складатися з нуля чи більше операторів. Зазвичай локальні класи оголошують усередині тіла методу. Оголошують локальні класи усередині довільного блоку, наприклад: тіла методу, оператора *for*, оператора розгалуження *if* тощо.

Клас можна визначити усередині функції. Розглянемо наступний приклад.

```
#include <iostream>
using namespace std;
void f();
int main() {
	f();
	// Клас myclass звідси не видний.
	return 0;
}
void f() {
	class myclass {
		int i;
	public:
		void put_i(int n) { i = n; }
		int get_i() { return  i; }
	}   ob;
	ob.put_i(10);
	cout << ob.get_i();
    }
```

На локальні класи накладає кілька обмежень. По-перше, усі функції-члени повинні визначатися усередині оголошення локального класу. По-друге, локальні класи не можуть звертатися до локальних змінних, оголошеним всередині функції, за винятком статичних локальних змінних (**static**) і зовнішніх змінних (**extern**). Однак вони мають доступ до імен типів і обчисленням, визначених у зовнішній функції. По-третє, усередині локальних класів не можна повідомляти статичні змінні. Через ці обмеження локальні класи застосовуються досить рідко.

10.Передача об'єктів функціям

Об'єкти можна передавати функціям, як звичайні змінні. Для цього застосовується звичайний механізм передачі параметрів за значенням. Незважаючи на зовнішню простоту, цей процес може привести до несподіваних наслідків, що стосуються конструкторів і деструкторів. Спробуємо розібратися в цьому за допомогою наступної програми.

```
// Передача об'єкта функції.
#include <iostream>
using namespace std;
class myclass {
	int i;
public:
	myclass(int n);
	~myclass();
	void set_i(int n) { i = n; }
	int get_i() { return i; }
};
myclass::myclass(int n) {
	i = n;
	cout << "Створення  " << i << "\n";
}
myclass::~myclass() {
	cout << "Знищення " " i << "\n";
}
void f(myclass ob);
int main() {
	myclass o(1);
	f(o);
	cout << "Це змінна i у функції main: ";
	cout << o.get_i() << "\n";
```

```
        return 0;
}
void f(myclass ob) {
        ob.set_i(2);
        cout << "Це локальна  змінна   i:    " << ob.get_i();
        cout << "\n";
    }
```

У результаті роботи цієї програми на екрані з'являться наступні рядки.

Створення 1
Це локальна змінна i: 2
Знищення 2
Це змінна i у функції main: 1
Знищення 1

Як показують ці результати, у ході роботи програми конструктор викликався лише одного разу, при створенні об'єкта **o** у функції main(), у той же час, деструктор був викликаний двічі. Розглянемо причини цієї ситуації.

Коли об'єкт передається у функцію, створюється його копія. Саме ця копія стає параметром функції. Це означає, що в програмі з'явився новий об'єкт. По завершенню роботи функції копія аргументу (тобто параметр) знищується. Це породжує два принципово важливі питання. По-перше, чи викликається конструктор при створенні копії аргументу? По-друге, чи викликається деструктор при знищенні параметра? Відповіді на ці питання вас здивують.

При створенні копії аргументу звичайний конструктор не викликається. Замість нього викликається конструктор копіювання. Саме він і створює нову копію аргументу. Конструктор копіювання можна визначати явно. Однак, якщо оголошення класу не містить явного визначення конструктора копіювання, як у нашому прикладі, викликається автоматичний конструктор копіювання, передбачений за замовчуванням. Він створює побітову (тобто ідентичну) копію об'єкта. Це легко зрозуміти. Адже звичайний конструктор не можна викликати для створення копії вже існуючого об'єкта, оскільки такий виклик замінив би поточний стан об'єкта первісним. У той же час при передачі об'єкта у функцію нам потрібний повний дублікат об'єкта, що відбиває його поточне, а не первісний стан.

По завершенню роботи функції копія об'єкта виходить із області видимості, тому викликається його деструктор. От чому в попередній програмі деструктор викликався двічі. Спочатку він викликається, коли параметр функції f() виходить зі своєї області видимості, а потім - коли при завершенні роботи програми знищується об'єкт o.

Підведемо підсумки. Якщо виникає необхідність створити копію об'єкта й передати її функції, звичайний конструктор не викликається. Замість нього викликається автоматичний конструктор копіювання, що створює побітовий дублікат об'єкта. Однак при знищенні копії об'єкта (як правило, у момент повернення з функції) викликається його деструктор.

Оскільки автоматичний конструктор копіювання створює точну копію оригіналу, він може стати джерелом проблем. Навіть якщо об'єкт передається функції за значенням, що теоретично дозволяє захистити аргумент від зміни, можливі побічні ефекти. Наприклад, якщо об'єкт, переданий як параметр, виділяє

динамічну пам'ять у момент свого створення й звільняє її при своєму уничтожении, його локальна копія усередині функції буде звільняти ту ж саму область пам'яті при виклику деструктора. Це приведе до ушкодження оригіналу. Для того щоб запобігти появі таких проблем, у класі необхідно явно визначити конструктор копіювання (див. главу 14).

11.Повернення об'єктів

Функція може повертати об'єкт викликаючому модулю. Як приклад проаналізуємо наступну програму.

```
// Повернення об'єктів з функцій.
#include <iostream>
using namespace std;
class myclass {
       int i; public:
              void set_i(int n) { i = n; }
              int get_i() { return i; }
};
myclass f();  // Повернення об'єкта класу myclass.
int main() {
       myclass o;
       o = f();
       cout << o.get_i() << "\n";
       return 0;
}
myclass f() {
       myclass x;
       x.set_i(1); return x;
    }
```

При поверненні з функції автоматично створюється тимчасовий об'єкт, у якому зберігається значення, що вертається. Саме цей об'єкт насправді вертається в викликаючий модуль. Після повернення цей об'єкт знищується. Це може привести до непередбачених наслідків. Наприклад, якщо деструктор об'єкта, що вертається, повинен звільняти динамічну пам'ять, вона буде звільнятися, навіть якщо об'єкт, якому привласнюється значення, що вертається, як і раніше посилається на неї. Уникнути цього можна за допомогою перевантаження оператора присвоювання і визначення конструктора копіювання.

12.Присвоювання об'єктів

Якщо об'єкти мають однаковий тип, їх можна привласнювати один одному. При цьому дані, що зберігаються в об'єкті, що знаходиться в правій частині оператора присвоювання, привласнюються змінним-членам об'єкта, що знаходиться в лівій частині. Проілюструємо цей механізм за допомогою наступної програми, що виводить на екран число 99.

```
// Присвоювання об'єктів.
#include <iostream>
using namespace std;
class myclass {
       int i;
public:
       void set_i(int n) { i = n; }
```

```
        int get_i() { return i; }
};
int main() {
        myclass ob1, ob2;
        ob1.set_i(99);
        ob2 = ob1; // Присвоювання даних об'єкта ob1 об'єкту ob2.
        cout << "Це змінна і з об'єкта і: " << ob2.get_i();
        return 0;
    }
```

За замовчуванням усі дані з одного об'єкта привласнюються відповідним змінним іншого об'єкта за допомогою побітового копіювання. Однак оператор присвоювання можна перевантажити й визначити іншу процедуру.