Лекція 2. Абстрактні методи. Абстрактні класи

1. *Класифікація об'єктів. Як обрати базовий клас?*
2. *Ієрархія класів плоских геометричних фігур. Абстрактний клас.*
3. *Визначення операторів для ієрархії класів.*
4. *Які члени класу не наслідуються. Оголошення конструкторів, деструкторів, оператора присвоєння у підкласах.*
5. *Поліморфна колекція об'єктів. Завантаження об'єктів з файла, зберігання до файла.*

Виконаємо для початку невелике логічне завдання: класифікуйте (поділіть на категорії) об'єкти, зображені на рис. 1.

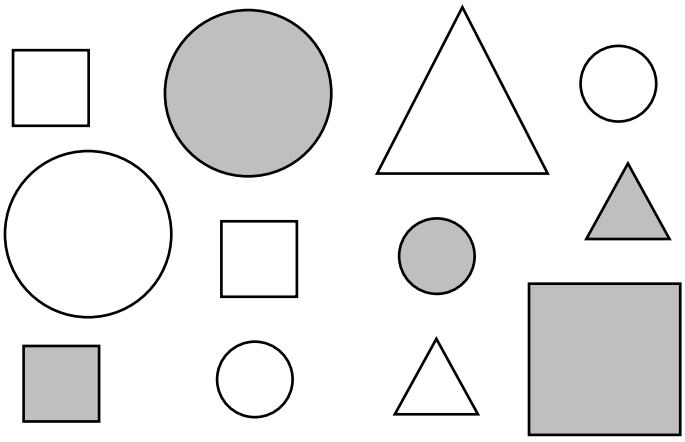

Рис. 1. Множина графічних об'єктів

На гадку спадає декілька способів класифікації:

- великі за розміром – маленькі;
- зафарбовані – незафарбовані;
- круги – трикутники – квадрати.

Можна було б вигадати ще щось, наприклад, поділити за рядками чи стовпцями. Який же зі способів обрати? Відповідь неможливо дати, якщо лише розглядати рисунок. Потрібна додаткова інформація. Що саме потрібно робити з цими об'єктами? Якщо обчислювати площі, то актуальним буде поділ за формою, якщо виготовляти, то за виглядом (чи заповнені), якщо перебирати, то за рядками. Очевидно, для правильної класифікації потрібно цікавитися не стільки *станом* об'єкта, скільки його *поведінкою*.

У попередній лекції ми розглядали різні способи організації ієрархії класів «Прямокутник», «Квадрат». Варіант *Rectangle* ← *Square* призводить до того, що квадрат містить «зайве» поле – другу сторону. Здається, варіант *Square* ← *Rectangle* більш економний: у класі *Square* оголошують поле для однієї сторони, а в класі *Rectangle* використовують його як успадковане і оголошують ще одне поле для другої. Проте, другий варіант – це результат класифікації *за структурою*, а перший – *за поведінкою*. Свідченням цього є склад методів класів у кожному з випадків наслідування. У другому випадку доведеться перевизначати *всі* методи: обчислення площі та периметра, виведення в потік. У першому достатньо перевизначити тільки виведення в потік. До того ж перший варіант правильно відображає відношення між поняттями в реальному світі: квадрат є частковим випадком прямокутника. Саме тому ми й віддали йому перевагу в попередній лекції.

Наслідування дає змогу вирішити в об'єктно-орієнтованій програмі принаймні два завдання: повторне використання коду та структурування сутностей програми.

Щодо повторного використання: програмісти люблять використовувати чужий код для власних потреб, бо це суттєво полегшує життя. Але як тільки виникає потреба у змінах цього коду, навіть незначних, для кращого пристосування до потреб задачі, можуть виникати

проблеми. ООП вирішує їх так, що ви можете оголосити новий клас на базі наявного, навіть без доступу до тексту цього класу. Говорять, що новий клас *породжено* від старого, його називають *підкласом*, *дочірнім* класом. Старий клас називають *базовим*, *батьківським* або *надкласом*. Підклас успадковує з базового всі поля даних, наслідує його поведінку.

За допомогою наслідування ви можете:

- доповнити функціональність класу (оголосити в підкласі нові методи);
- розширити перелік елементів даних (оголосити додаткові поля даних);
- змінити поведінку класу (перевизначити методи).

Відношення «надклас-підклас» дає змогу організувати класи об'єктно-орієнтованої програми в ієрархічну структури, тим самим упорядкувати поняття. Як наслідок, абстрактні поняття будуть змодельовані класами, розташованими на вищих рівнях ієрархії. Їхня поведінка є найбільш загальною. На нижніх рівнях ієрархії – конкретні класи з деталізованою поведінкою.

Практика показує, що спочатку ієрархія класів будується знизу вгору: конкретні класи моделюють сутності реального світу, надкласи утворюють за допомогою генералізації – виділення спільних рис класів і узагальнення поведінки. Пізніше, в ході експлуатації, ієрархія класів розростається донизу за рахунок додавання нових підкласів з новими рисами поведінки, тобто, за допомогою деталізації.

Мова C++ дає змогу організовувати наслідування декількома способами. Ми розглядаємо один з них – *відкрите* наслідування. Рушимо далі.

> *Завдання.* Оголосіть ієрархію класів, що моделюють сутності «прямокутник», «квадрат», «круг», «трикутник». Усі фігури повідомляють свої площу і периметр, вміють будувати своє зображення рядком літер.

У завданні йде мова про ієрархію, бо всі згадані об'єкти мають спільні риси, схожу поведінку. Кого ж вибрати в якості базового класу? Наприклад, один з оголошених раніше класів: *Rectangle* або *Square*? Це не дуже добра ідея. Ми ж не можемо сказати, що «круг – це якийсь особливий квадрат». Натомість кожен з перелічених в завданні об'єктів «є плоскою геометричною фігурою». Всяка плоска геометрична фігура має площу і периметр, та ще якісь виміри. Такі міркування приведуть нас до ієрархії, зображеної на рис. 1.

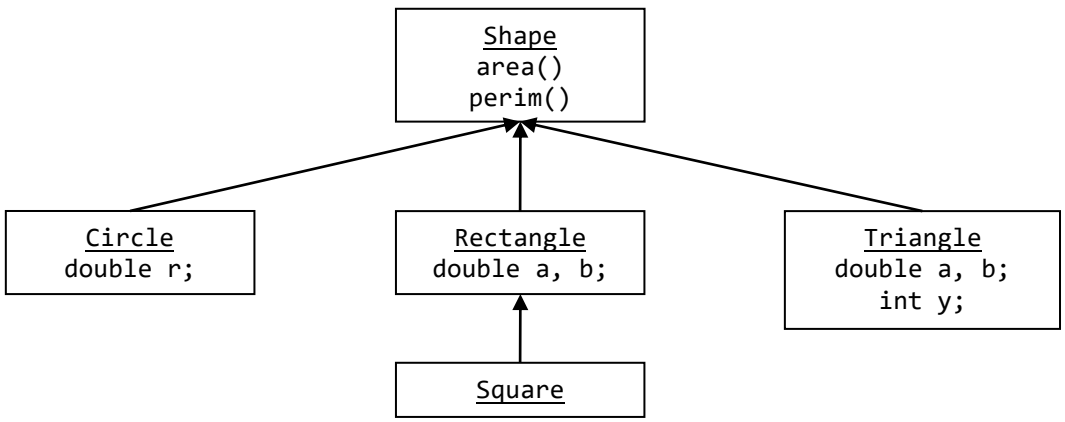

Рис. 2. Проект ієрархії плоских геометричних фігур

У такій ієрархії клас *Shape* описує не конкретний об'єкт, а певне узагальнене поняття. Ми точно знаємо, що кожна фігура має площу і периметр, але не знаємо, як їх обчислювати, поки не знаємо конкретного типу фігури. Мова C++ має спеціальні засоби для опису такої ситуації. Розглянемо наступний код.

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

```
	virtual double area() const = 0;
	virtual double perim() const = 0;
	virtual void printOn(ostream&) const = 0;
	virtual char * toStr() const = 0;
};
```

Тут замість визначення методів використано конструкцію « = 0;» Вона означає, що метод абстрактний, або «чисто віртуальний» в термінології C++, і не потребує ще якогось визначення. Клас, що містить хоча б один абстрактний метод, є абстрактним. Головне його призначення – стати базовим для ієрархії класів і задати протокол взаємодії з ними. Підклас абстрактного класу мав би перевизначати абстрактні методи. Якщо хоча б один абстрактний метод не буде перевизначено в підкласі, то це означатиме, що підклас наслідує абстрактний метод і тому сам є абстрактним. Якщо абстрактний клас містить поля даних, то можна визначити конструктор для їхньої ініціалізації. Такий конструктор використовують лише в конструкторах підкласів. Створити ж екземпляр абстрактного класу – неможливо, компілятор не дозволить зробити це.

Синтаксис « = 0;» можна трактувати як «вказівник на код методу – порожній» і нема потреби визначати його тіло. У сучасний стандарт мови C++ введено слово *abstract* для позначення абстрактних методів. Його використовують замість « = 0;».

Усі методи класу *Shape* оголошено віртуальними, бо всі вони залежать від конкретного типу фігури і мають бути перевизначені у підкласах. Метод, незалежний від типу отримувача повідомлення, оголошують звичайним чином. Надалі його використовують всі класи ієрархії. Можливо, ми захочемо порівнювати фігури, наприклад, знаходити фігуру з найбільшою площею. Тоді доречно перевантажити оператор порівняння, причому, достатньо зробити це лише один раз – у базовому (абстрактному) класі. Такий оператор використовує метод обчислення площі (див. нижче), і це дуже цікаво: базовий клас ще не знає, як буде визначено обчислення площі в підкласах, але вже може використати цю функціональність. Завдяки поліморфізму (посилань і повідомлень) оператор працюватиме правильно з екземплярами всіх підкласів. Поруч з базовим класом можна оголосити також зовнішню функцію – оператор виведення в потік. Вона використовуватиме віртуальний метод *printOn*.

Повне визначення ієрархії класів матиме вигляд, наведений нижче.

Файл FlatShapes.h

```
// абстрактний клас повідомляє, що вміють робити (мусять вміти) його підкласи
// деструктор оголошено віртуальним, щоб всі деструктори були віртуальними і правильно
//   працювали в поліморфних колекціях

class Shape
{
public:
	virtual ~Shape() {}
	virtual double area() const abstract;
	virtual double perim() const abstract;
	virtual void printOn(ostream&) const abstract;
	virtual void storeOn(ofstream&) const abstract;
	virtual char * toStr() const abstract;
	bool operator>(const Shape& other) const
	{
		return this->area() > other.area();
	}
};

// єдиний оператор виведення для всієї ієрархії
ostream& operator<<(ostream& os, const Shape& s);

// конструктор з параметрами, в якого всі параметри мають значення за замовчанням,
//   стає також і конструктором за замовчанням
```

```
class Rectangle : public Shape
{
protected:
	double a;
	double b;
public:
	Rectangle(double sideA=1., double sideB=1.) : a(sideA), b(sideB) {}
	virtual double area() const override;
	virtual double perim() const override;
	virtual void printOn(ostream&) const override;
	virtual void storeOn(ofstream&) const override;
	virtual char * toStr() const override;
};

class Circle : public Shape
{
private:
	double r;
public:
	Circle(double radius=1.) : r(radius) {}
	virtual double area() const override;
	virtual double perim() const override;
	virtual void printOn(ostream&) const override;
	virtual void storeOn(ofstream&) const override;
	virtual char * toStr() const override;
	double radius() const { return r; }
};

class Triangle : public Shape
{
private:
	double a; // сторони трикутника
	double b;
	int y;    // і кут у градусах
	double angle() const { return M_PI*y/180; }
public:
	Triangle(double sideA=3., double sideB=4., int angle=90) : a(sideA), b(sideB),
y(angle) {}
	virtual double area() const override;
	virtual double perim() const override;
	virtual void printOn(ostream&) const override;
	virtual void storeOn(ofstream&) const override;
	virtual char * toStr() const override;
};

// особливість квадрата у способі конструювання та ще виведення на друк
class Square : public Rectangle
{
public:
	Square(double side=1.) : Rectangle(side,side) {};
	virtual void printOn(ostream&) const override;
	virtual void storeOn(ofstream&) const override;
	virtual char * toStr() const override;
};
```

Трикутник задано двома сторонами та кутом між ними, конструктори покладаються на правильність задання аргументів – все досить спрощено, щоб не витрачати багато місця на перевірки.

Модифікатор *override* повідомляє компіляторові, що ми хочемо перевизначити метод базового класу, і спонукає його до перевірок: чи не помилилися ми з сигнатурою цього методу. Цей модифікатор варто вказувати кожного разу при перевизначенні методів.

```
#include "FlatShapes.h"

ostream& operator<<(ostream& os, const Shape& s)
{
	s.printOn(os);
	return os;
}
//-------------------------------------------------------------
double Rectangle::area() const
{	return a * b;
}
double Rectangle::perim() const
{	return (a + b) * 2.;
}
void Rectangle::printOn(ostream& os) const
{	os  << "Rectangle of size " << std::fixed << std::setprecision(1)
		<< a <<" x "<< b << ';';
}
void Rectangle::storeOn(ofstream& fout) const
{
	fout << "R " << a << ' ' << b;
}
char * Rectangle::toStr() const
{	char aChar[10] = {0}; gcvt(this->a,8,aChar);
	char bChar[10] = {0}; gcvt(this->b,8,bChar);
	char * result = new char[21+strlen(aChar)+strlen(bChar)];
	char * ptr = strcpy(result, "Rectangle of size ");
	ptr = strcat(ptr, aChar); ptr = strcat(ptr, " x "); strcat(ptr, bChar);
	return result;
}
//-------------------------------------------------------------
double Circle::area() const
{	return M_PI*r*r;
}
double Circle::perim() const
{	return 2.*M_PI*r;
}
void Circle::printOn(ostream& os) const
{	os  << "Circle of radius " << std::fixed << std::setprecision(1) << r << ';';
}
void Circle::storeOn(ofstream& fout) const
{	fout << "C " << r;
}
char * Circle::toStr() const
{	char rChar[10] = {0}; gcvt(this->r,8,rChar);
	char * result = new char[17+strlen(rChar)];
	char * ptr = strcpy(result, "Circle of radius ");
	strcat(ptr, rChar);
	return result;
}
//-------------------------------------------------------------
double Triangle::area() const
{	return 0.5*a*b*std::sin(angle());
}
double Triangle::perim() const
{	return a+b+std::sqrt(a*a+b*b-2.*a*b*std::cos(angle()));
}
void Triangle::printOn(ostream& os) const
{	os  << "Triangle of side " << std::fixed << std::setprecision(1)
		<< a << ',' << b << " with angle " << y << " degree;";
}
void Triangle::storeOn(ofstream& fout) const
{	fout << "T " << a << ' ' << b << ' ' << y;
```

```
}
char * Triangle::toStr() const
{      char aChar[10] = {0}; gcvt(this->a,8,aChar);
       char bChar[10] = {0}; gcvt(this->b,8,bChar);
       char yChar[5] = {0}; itoa(this->y,yChar,10);
       char * result = new char[39+strlen(aChar)+strlen(bChar)+strlen(yChar)];
       char * ptr = strcpy(result, "Triangle of side ");
       ptr = strcat(ptr, aChar); ptr = strcat(ptr, ", ");
       ptr = strcat(ptr, bChar); ptr = strcat(ptr, " with angle ");
       ptr = strcat(ptr, yChar); strcat(ptr, " degrees");
       return result;
}
//-----------------------------------------------------------
void Square::printOn(ostream& os) const
{      os  << "Square of side " << std::fixed << std::setprecision(1) << a << ';';
}
void Square::storeOn(ofstream& fout) const
{      fout << "S " << a;
}
char * Square::toStr() const
{      char aChar[10] = {0}; gcvt(this->a,8,aChar);
       char * result = new char[15+strlen(aChar)];
       char * ptr = strcpy(result, "Square of side ");
       strcat(ptr, aChar);
       return result;
}
```

Повернемося ще раз до оголошення оператора виведення в потік. Воно досить показове і може бути прикладом того, як варто визначати оператори введення чи виведення для ієрархії класів у загальному випадку. Нагадаємо: у *кожному* класі ми визначили *віртуальний метод* виведення в потік і *одну функцію*, яка цей метод використовує. Досить складна схема. А чи не можна було зменшити об'єм програмування і визначити в кожному підкласі свою дружню функцію, яка б перевантажувала оператор виведення? Підхід з дружніми функціями досить поширений, проте він працює тільки для об'єктів відомих типів. Функції (навіть дружні) не є членами класу і не можуть бути віртуальними, тому не забезпечують поліморфізму поведінки. Таким чином, використане нами рішення є єдино можливим.

Схему з віртуальними методами і функцією-оператором можна використовувати і в інших випадках. Наприклад, оператор порівняння можна було б перевантажити не методом, як зробили ми, а функцією:

```
bool operator>(const Shape& a, const Shape& b)
{      // зовнішня функція використовує віртуальний метод area
       return a->area() > b.area();
}
```

Щоб реалізувати оператор уведення з потоку, потрібно оголосити в базовому класі віртуальний абстрактний метод *readFrom(istream&)*, перевизначити його в підкласах і перевантажити оператор функцією, яка викликає цей метод.

Чи зауважили читачі, що в ієрархії класів оголошено єдиний деструктор *~Shape()*? І той – віртуальний. Це не випадково. Екземпляри різних підкласів потребують різного обсягу пам'яті, тому деструктори *мають* бути віртуальними. Оголошення деструктора базового класу віртуальним гарантує, що деструктори всіх класів ієрархії будуть віртуальними, чого нам і треба. Деструктори підкласів можна не оголошувати: компілятор згенерує їх автоматично, і цього цілком достатньо.

З попередніх лекцій ми знаємо, що компілятор автоматично генерує конструктори, деструктор і оператор присвоєння, якщо їх не визначив програміст. У наших прикладах ми визначали конструктори для створення (вони ж – конструктори за замовчуванням) і

деструктор базового класу. Конструктори копіювання, деструктори підкласів і оператор присвоєння ми залишали компіляторові. Так можна робити, якщо об'єкт не захоплює для своїх потреб динамічної пам'яті. У протилежному випадку необхідно все згадане визначати власноруч. Адже «автоматичний» конструктор копіювання чи оператор присвоєння виконують не глибоке а поверхове копіювання екземпляра – лише значення його полів без урахування структур, на які можуть вказувати ці поля. Явно задають конструктор і в тому випадку, коли він повинен змінювати статичні члени класу.

Нагадаємо, що конструктор копіювання (явний чи автоматичний) працює, коли

- новий об'єкт ініціалізовано об'єктом того ж класу;
- об'єкт передано функції за значенням;
- функція повертає об'єкт за значенням;
- компілятор генерує тимчасовий об'єкт.

Використання посилань на об'єкти для передавання параметрів і повернення результату функції запобігає багатократному (і зайвому) виклику конструктора копіювання. Не забувайте про модифікатор *const* для захисту параметрів та позначення методів.

Зазначимо, які члени класу не наслідуються.

1. **Конструктори**. Для кожного класу є свої: або згенеровані компілятором, або визначені програмістом. У будь-якому випадку конструктор підкласу використовує батьківський конструктор для ініціалізації успадкованих полів: явно, якщо його вказати у списку ініціалізаторів, або автоматично (конструктор за замовчуванням), якщо цього не зробити.

```
class Base
{
private:     double a;
public:      Base(double x) : a(x) {}
       ...
};
class Deriv : public Base
{
private:     double b;
public:
       Deriv(double x, double y) : Base(x), b(y) {}; //єдиний спосіб ініціалізувати a
       ...
};
```

2. **Деструктори**. Так само, як і конструктори, для кожного класу – свої. Деструктор підкласу автоматично викликає деструктор надкласу для знищення вкладеного об'єкта (успадкованих полів). Деструктор базового класу обов'язково оголошують віртуальним, навіть, якщо клас міг би обійтися без деструктора. Таке оголошення заставляє компілятор генерувати віртуальні деструктори для всіх підкласів, а це означає, що звільнення пам'яті об'єктів – елементів поліморфних структур – відбуватиметься правильно.

3. **Оператори присвоєння**. Оператор присвоєння працює тоді, коли один об'єкт присвоюють іншому (не плутайте з ініціалізацією нового об'єкта, йде мова про присвоєння наявному). У C++ (як і в Object Pascal) об'єктові батьківського класу можна присвоїти об'єкт породженого (але не навпаки). При цьому відбувається копіювання тільки тих полів, які успадковані з батьківського класу.

Оператор присвоєння, визначений програмістом для підкласу має спеціальну структуру.

```
Deriv& Deriv::operator=(const Deriv& other)
{      if (this == &other) return *this;  // захист від самоприсвоєння
       Base::operator=(other);   // копіювання успадкованої частини
       b = other.b;              // копіювання власної частини
       return *this;
}
```

тут батьківський оператор присвоєння викликано як функцію з діапазону видимості батьківського класу.

Для того, щоб прослідкувати за роботою конструкторів-деструкторів, визначте їх явно і наділіть кожного з них здатністю повідомляти вас про своє виконання.

Тепер варто випробувати створені класи. Головна програма створює поліморфний масив плоских фігур (у поліморфних колекціях типом елемента є вказівник на базовий клас), випробовує звичайне виведення на екран, обчислення, перетворення на рядки та взаємодію з файлами. Справді, добре було б уміти зберігати об'єкти до файла та читати їх звідти. Визначений нами *operator*<< можна використовувати для виведення на консоль і до текстового файла. От тільки такий файл зручно буде читати людині, а не програмі: важкувато буде знаходити числа серед слів, якими фігури описують себе. Спеціально для зберігання об'єктів до файла в класах було визначено методи *storeOn*. Вони записують лише числові дані фігури та одну літеру – код типу фігури (підкласів небагато, тому для кодування достатньо одного символа). Такий «шифр» легко розпізнати програмно і відтворити об'єкти з файла. Саме це і робить наведена нижче програма за допомогою інструкції *switch*.

Зверніть увагу на таке: у більшості випадків нас не цікавить конкретний тип елемента масиву, оскільки працюють віртуальні методи. Щоб надрукувати елемент, ми просто пишемо *cout << *s[i]* – байдуже, хто ховається за тим вказівником. І тільки для створення об'єкта потрібно точно знати його тип. Завдяки розпізнаванню літер, що позначають типи фігур, програма правильно вибирає потрібні конструктори.

Файл Program.cpp

```
#include "FlatShapes.h"

	// випробування ієрархії плоских фігур
int main()
{
	std::cout << " * Static array *\n----------------------\n";
	Shape* s[] = {
		new Rectangle(6., 4.),
		new Square(5.),
		new Triangle(),
		new Circle(5.),
		new Square() };
	int n = sizeof s / sizeof *s;
	for (int i = 0; i < n; ++i)
		std::cout << *s[i] << "\n\tS = " << s[i]->area() << "  \tP = "
			<< s[i]->perim() << '\n';
	std::cin.get();
	std::cout << "\nShapes as strings\n\n";
	for (int i = 0; i < n; ++i)	std::cout << s[i]->toStr() << '\n';
	system("pause");

	std::cout << "\n\nCompareing shapes\n";
	int maxNo = 0;
	for (int i = 1; i < n; ++i)
	{
		if (s[i] > s[maxNo]) maxNo = i;
	}
	std::cout << "\nThe largest shape is " << *s[maxNo] << '\n';
	system("pause");

	std::cout << "\n\n Storeing and loading shapes to a dynamic array\n";
	// зберігання фігур до файла
	ofstream fout("Shapes.txt");
	// кількість фігур
	fout << n + 2 << '\n';
	// зберігання масиву
	Triangle rt(3.5, 3.5, 60); rt.storeOn(fout); fout << '\n';
	for (int i = 0; i < n; ++i)
	{
```

```
            s[i]->storeOn(fout); fout << '\n';
        }
        // зберігання окремих об'єктів
        Triangle tt(3, 5, 120); tt.storeOn(fout); fout << '\n';
        fout.close();
        std::cout << "\nShapes are stored\n";

        // завантаження об'єктів з файла
        std::ifstream fin("Shapes.txt");
        // створення масиву відповідного розміру
        fin >> n; fin.get();
        Shape ** D = new Shape*[n];
        // завантаження даних, розпізнавання типу фігури, створення об'єкта
        for (int i = 0; i < n; ++i)
        {
            double x, y; int g;
            char type; fin >> type;

            switch (type)
            {
            case 'R': fin >> x >> y; fin.get();
                  D[i] = new Rectangle(x, y); break;
            case 'C': fin >> x; fin.get();
                  D[i] = new Circle(x); break;
            case 'T': fin >> x >> y >> g; fin.get();
                  D[i] = new Triangle(x, y, g); break;
            case 'S': fin >> x; fin.get();
                  D[i] = new Square(x);
            }
        }
        // контрольне виведення на екран
        for (int i = 0; i < n; ++i)
        {
            std::cout << *D[i] << '\n';
        }
        system("pause");
        return 0;
}
```

Зауваження. У попередній лекції ми обговорювали питання "навіщо квадратові друге поле, успадковане від прямокутника" і вирішували, чи не зробити клас квадрата базовим. Тепер можна запропонувати третій спосіб вирішення проблеми, адже у нас є абстрактний базовий клас *Shape*. Увага: і *Square*, і *Rectangle* могли б наслідувати *Shape*. Вони могли б стати на один рівень ієрархії класів. Звичайно, методи обчислення площі та периметру квадрата доведеться визначати заново, проте зайве поле даних у ньому не появиться.