Лекція 17. Побудова класів: наслідування

1. *Клас «Прямокутник». Конструктор. Два способи реалізації поведінки.*
2. *Клас «Квадрат». Різні варіанти наслідування.*
3. *Поліморфізм структур даних. Поліморфізм повідомлень.*

Мета цієї лекції – продемонструвати на прикладах процес проектування, визначення та використання класів мовою C++. Деякі з висвітлених тут питань ми вже обговорювали в попередніх лекціях, до обговорення декількох інших повернемося в наступних лекціях. Поки що ж переходимо до розв'язування конкретних завдань.

> *Завдання 1.* Оголосіть клас, що моделює сутність «прямокутник». Фігуру задано розмірами сторін. Користувачів цікавитиме площа і периметр прямокутника.

Перш ніж розпочати написання коду, ми повинні дати відповідь на запитання «чи зрозуміла умова завдання?» Не поспішайте, бо відповідь, насправді, неочевидна. :)

Залежно від нашого попереднього досвіду, «прямокутник» може асоціюватися з різними об'єктами: прямокутники – різноколірні фігури, вирізані з паперу; фігури на координатній площині, довільним чином нахилені до осей координат; прямокутні ділянки екрана з довільним вмістом і сторонами, паралельними до сторін екрана. Перелік прикладів можна продовжити. Кожен з них описує *інші* прямокутники зі своїм особливим переліком властивостей. Об'єкти реального світу невичерпні у своїй багатогранності, тому в процесі *моделювання* потрібно абстрагуватися від переважної більшості їхніх властивостей і зосередитися виключно на тих, які необхідні для розв'язання конкретної поставленої задачі.

У завданні йдеться про обчислення площі та периметра прямокутника і нічого не сказано про його зовнішній вигляд. Тому під час написання класу ми врахуємо довжину і ширину прямокутника але абстрагуємося від його координат, кольору ліній, способу заповнення тощо. Пам'ятаємо зі шкільного курсу геометрії, що прямокутник – це чотирикутник, у якого всі кути прямі. Протилежні сторони прямокутника попарно рівні, їхні довжини задано дійсними додатними числами. Якщо позначити їх a і b, то площу і периметр можна обчислити, відповідно, за формулами $S = a \times b$ і $P = 2\,(a + b)$.

Перша спроба оголошення класу *Rectangle* матиме вигляд:

Файл Rect.h

```
class Rectangle
{
private:
	double a;
	double b;
public:
	Rectangle(double sideA=1., double sideB=1.) : a(sideA), b(sideB) {}
	double area() const;
	double perim() const;
	char * toStr() const;
};
```

Тут поля даних призначено для зберігання розмірів прямокутника. Конструктор з параметрами, в якого всі параметри мають значення за замовчанням, є також і конструктором за замовчанням. Методи обчислення площі та периметра «обіцяють» не змінювати сам прямокутник. З'явився ще один метод, про який ми не згадували раніше. Судячи з його назви, метод *toStr()* повертатиме зображення об'єкта у вигляді рядка. Цей метод стане в пригоді для виведення інформації про об'єкт у консоль, чи в текстове вікно аплікації. Багато мов заохочують визначення таких методів у кожному класі (наприклад, Smalltalk, C#, Python).

Екземпляр класу *Rectangle* має просту структуру, тому нас цілком влаштують конструктор копіювання і деструктор, згенеровані компілятором

Файл Rect.cpp

```
double Rectangle::area() const
{	return a * b;
}
double Rectangle::perim() const
{	return (a + b) * 2.;
}
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;
}
```

Метод *toStr()* будує рядок у стилі С, що є зображенням об'єкта. Не дивлячись на свою багатослівність, він є досить ефективним.

Завдання виконано. Але навіть цей простий приклад дає досить простору для обговорення можливих варіантів реалізації.

1. Чи все гаразд з конструкторами?

Ні, бо існує можливість створення «нереальних об'єктів». Визначений нами конструктор без жодних перевірок копіює значення параметрів у поля. Тому допустимим є і такий код:

```
double x = -1.0; double y = 1.0;
Rectangle R(x, x+y);
// Що це за прямокутник (-1 x 0) ???
```

Вирішення проблеми – «інтелектуальний» конструктор з параметрами, що пильнує за коректністю аргументів. Запропонований нижче варіант конструктора виправляє неправильні значення на деякі стандартні:

```
Rectangle() : a(1.), b(1.) {} // тут перевірки зайві
Rectangle(double sideA, double sideB)
{	a = (sideA>0.1) ? sideA : 0.1;
	b = (sideB>0.1) ? sideB : 0.1;
}
```

2. Чи все гаразд з розмірами прямокутника?

Проблема виникне, якщо ми захочемо змінити їх – ми не визначили методів доступу до полів об'єкта. Вирішення – визначити такі методи

```
void Rectangle::setA(double sideA)
{
	a = (sideA>0.1) ? sideA : 0.1;
}
void Rectangle::setB(double sideB)
{
	b = (sideB>0.1) ? sideB : 0.1;
}
```

Або такий, цікавіший варіант

```
Rectangle& Rectangle::setA(double sideA)
{	a = (sideA>0.1) ? sideA : 0.1;
	return this*;
}
Rectangle& Rectangle::setB(double sideB)
{	b = (sideB>0.1) ? sideB : 0.1;
	return this*;
}
```

Ці методи можна було б використати і в конструкторі.

3. Чи все гаразд з обчисленням площі?

Відповідь залежить від того, як будуть використовувати метод *area()*. Якщо його викликатимуть часто за незмінних розмірів прямокутника, то багатократне обчислення одного і того ж виразу – неприпустима розкіш. Вирішення – змінити реалізацію. Наприклад, так: зберігати обчислену завчасу площу і за запитом повертати готовий результат.

```
class Rectangle
{
private:
	double a;
	double b;
	// деталі обчислення і зберігання площі
	double S;
	void setArea() { S = a * b; }
public:
	Rectangle(double sideA=1., double sideB=1.) : a(sideA), b(sideB)
	{	setArea();
	}
	double square() const { return S; } // все, що залишилося зробити
	double perim() const;
	char * toStr() const;
	Rectangle& Rectangle::setA(double sideA)
	{	a = (sideA>0.1) ? sideA : 0.1;
		this -> setArea(); // зміна розміру означає зміну площі
		return this*;
	}
	Rectangle& Rectangle::setB(double sideB)
	{	b = (sideB>0.1) ? sideB : 0.1;
		this -> setArea();// зміна розміру означає зміну площі
		return this*;
	}
};
```

Ми не випадково розташували метод *setArea()* в закритій частині: він не є частиною інтерфейсу, а лише способом реалізації поведінки об'єкта.

4. Чи все гаразд із засобами виведення екземпляра на друк?

Екземпляр *Rectangle R;* можна надрукувати так: *cout << R.toStr();* але це не надто зручно. Було б доцільно перевантажити оператор виведення в потік для типу *Rectangle*, наприклад, за допомогою відповідного методу класу та зовнішньої функції:

```
void Rectangle::printOn(ostream& os) const
{
	os << "Rectangle ( " << a << " x " << b << " )";
}
ostream& operator<<(ostream& os, const Rectangle& R)
{
	R.printOn(os);
	return os;
}
```

Зрозуміло, що оголошення методу *printOn* потрібно додати до оголошення класу.

Наслідування

> *Завдання 2.* Оголосіть клас, що моделює сутність «квадрат». Як і в попередньому завданні, фігуру задано довжиною сторони, користувачів цікавитиме площа і периметр квадрата.

Ми могли б взяти за зразок клас *Rectangle* і «з нуля» визначити новий, наприклад, *Square*. Він був би дуже схожим до попереднього. Але класи надають нам потужний спосіб повторного використання коду – наслідування. Новий клас можна оголосити на базі наявного. Тоді новий клас називатиметься підкласом, а старий – надкласом. Підклас успадковує структуру надкласу і наслідує його поведінку. Підклас може видозмінювати цю поведінку, доповнювати її. Надклас описує загальний випадок поняття, а підклас – конкретизує його, описує особливий випадок.

Квадрат – це прямокутник, у якого всі сторони рівні. Квадрат «є» особливий прямокутник. Відношення «є» («is-a» англійською) моделюють наслідуванням. Клас «квадрат» можна оголосити підкласом класу «прямокутник».

Файл Sqr.h

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

Рівні сторони прямокутнику задають у конструкторі. Об’єкт підкласу містить (неявно) вкладений екземпляр надкласу. Єдиний спосіб ініціалізувати його – викликати конструктор базового класу в списку ініціалізації. Рядок зображення об’єкта-квадрата мав би відрізнятися від зображення прямокутника.

Файл Sqr.cpp

```
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;
}
```

Фантастика! Ці декілька рядків коду справді визначають новий клас! Схематично відношення між класами зображене на рис. 1.

Проте є певний сумнів: навіщо квадратові два поля даних? Для зберігання розміру цілком вистачило б і одного. Можливо, базовим зробити клас *Square*? Така архітектура зображена на рис. 2. Базовий клас використовує одне поле і описує функціональність квадрата ($P = 4a$; $S = a^2$). Підклас додає ще одне поле і перевизначає методи (периметр і площу рахують інакше). У результаті ми отримали економію пам’яті (для квадратів) і перевизначення багатьох методів у підкласі.

Обидва варіанти мають право на життя. То якому віддати перевагу? Який критерій застосувати: обсяг пам’яті чи обсяг коду? Давайте поглянемо на ситуацію з іншого боку: що моделюють запропоновані архітектури? Перша – ієрархію понять (квадрат є особливим прямокутником), друга – ієрархію структур (прямокутник містить більше даних). Але другий варіант також стверджує, що «прямокутник – це такий квадрат, у якого одна сторона більша від іншої». Нісенітниця. Ми прийшли до суперечності з відношенням понять у реальному світі.

Мусять бути якісь дуже вагомі причини, щоб віддати перевагу другому способу підпорядкування класів. Натомість на користь першого свідчать і відповідність понять, і легкість програмування. На ньому і спинимося.

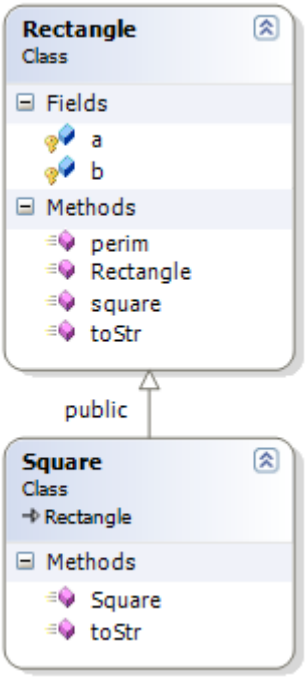

Рис. 1. Квадрат є прямокутник

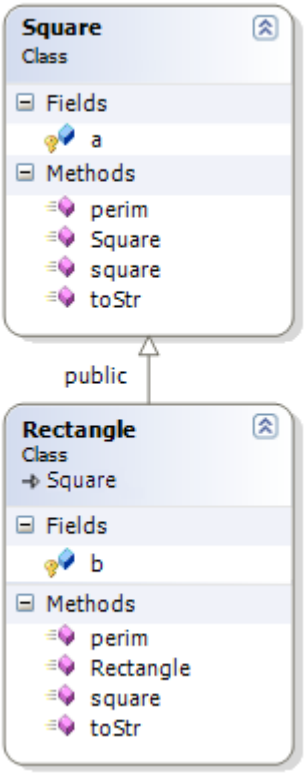

Рис. 2. Прямокутник має більше полів

Поліморфізм

Одна з причин застосування наслідування – це можливість використовувати одні і ті ж змінні для розміщення різнотипних об’єктів (але з однієї ієрархії типів). Ця можливість забезпечена розширеними правилами сумісності для споріднених класів. Допустимими є такі присвоєння (*class Square : public Rectangle*):

```
// Кожна змінна має конкретний тип,
// відбувається перетворення екземпляра підкласу на екземпляр базового класу
Rectangle R; Square S(5); R = S;

// Посилання rR – поліморфне, бо може означати довільний об’єкт ієрархії Rectangle
// Тип екземпляра Square зберігається. Застосовують як поліморфний параметр
// для передавання об’єктів у методи чи функції
Square S(5); Rectangle & rR = S;

// Поліморфний вказівник pR може містити адресу екземпляра довільного типу з ієрархії.
// Використовують для побудови поліморфних колекцій
Rectangle * pR; int k; cin >> k;
if (k % 2 == 0)
       pR = new Square S(5);
else pR = new Rectangle(3,4);
cout << pR->toStr();
```

Розглянемо детальніше останній приклад. Неможливо визначити наперед, на який об’єкт вказуватиме *pR*: залежно від парності введеного користувачем числа, це може бути квадрат або прямокутник. Завдяки сумісності вказівників на споріднені класи *pR** цілком може бути квадратом зі стороною 5. Увівши парне ціле для *k*, сподіваємося отримати у

відповідь «*Square of side 5.*» Проте, станеться не так, як гадалося. Програма відповість «*Rectangle of size 5. x 5.*» Об'єкт ми створили правильно, але чомусь спрацьовує метод *Rectangle::toStr()* замість *Square::toStr()*.

У чому ж причина? У способі оголошення цих методів. Аналізуючи повідомлення *pR -> toStr()*, компілятор встановить, що *pR* вказує на *Rectangle* і пов'яже виклик з методом *Rectangle::toStr()* – інакше він просто й не може. Ми мусимо якимось чином повідомити компіляторові, що прив'язування виклику методу потрібно відкласти на потім, до етапу виконання, коли стане відомим справжній тип об'єкта *pR**. З цією метою використовують модифікатор *virtual* в оголошенні методу. У базовому класі:

```
class Rectangle
{
private:
	double a;
	double b;
public:
	Rectangle(double sideA=1., double sideB=1.) : a(sideA), b(sideB) {}
	double area() const;
	double perim() const;
	virtual char * toStr() const; // віртуальний метод, пізнє зв'язування
	virtual ~Rectangle() {}       // деструктор також мусить бути віртуальним
};
```

Відповідний метод у підкласі автоматично стає віртуальним навіть без додаткових змін, та, все ж, бажано вказувати *virtual* також і в підкласах.

Тепер пов'язування виклику методу компілятор організує за іншою схемою. Для кожного класу з віртуальними методами буде організовано спеціальну структуру – таблицю віртуальних методів (VMT), яка міститиме вказівники на код кожного з віртуальних методів. Схематично це зображено на рис. 3. Кожен об'єкт має доступ до VMT свого класу і сам вибирає необхідний метод. Шлях відшукання методу для прямокутника зображено штриховою лінією, для квадрата – суцільною.

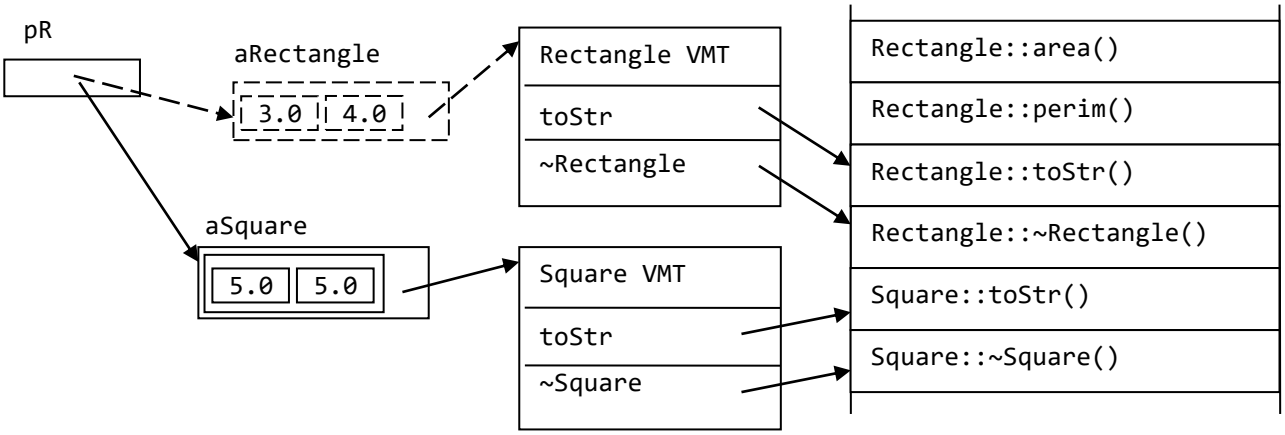

Рис. 3. Виклик віртуального методу

Деструктори в ієрархії класів також мають бути віртуальними. Для того, щоб компілятор правильно опрацьовував знищення об'єктів звільняв саме стільки пам'яті, скільки займає об'єкт.

Повідомлення *pR -> toStr()* є прикладом *поліморфного* повідомлення. Насправді, його опрацьовує об'єкт невідомого типу. Ми не можемо знати наперед, чи це прямокутник, чи квадрат. Завдяки пізньому зв'язуванню буде викликано один з методів: *Rectangle::toStr()* або *Square::toStr()* – відповідно до справжнього типу об'єкта-отримувача повідомлення.