Лекція 6. Узагальнення в програмуванні. Шаблони

1. *Від конкретних даних до довільних, від частини програми до функції, від конкретного типу до довільного.*
2. *Шаблон функції: оголошення, використання, явне і неявне створення екземпляра.*
3. *Часткова спеціалізація, перевантаження шаблона функції.*
4. *Шаблон класу: оголошення, використання, явне і неявне створення.*
5. *Часткова спеціалізація шаблона класу.*

Чи знайомі ви з узагальненнями в програмуванні? Напевне, що так, навіть не підозрюючи про це. Розберемося на прикладі.

Задача. Відомо, що $P_3(x) = x^3 + 2.7x^2 - 13.5x + 9.75$. Потрібно обчислити $P_3(2.31)$.

Напишемо програму, що розв'язує цю задачу. Обчислення організуємо за схемою Горнера:

```
int main()
{
	// 1. Дуже конкретний розв'язок
	cout << "P(2.31) = " << ((2.31 + 2.7)*2.31 - 13.5)*2.31 + 9.75 << '\n';
	system("pause");
	return 0;
}
```

Це хороша програма, бо написана дуже швидко і друкує правильний результат. Але такий код більше схожий на алгоритм обчислень на калькуляторі. Його складно назвати програмою, бо він не придатний для обчислення значення полінома для довільного заданого *x*. Спробуємо виправити цей недолік.

```
int main()
{
	// 2. Узагальнення для довільного значення x
	double x;
	cout << "Input x: ";
	cin >> x;
	cout << "P(" << x << ") = " << ((x + 2.7)*x - 13.5)*x + 9.75 << '\n';
	system("pause");
	return 0;
}
```

Отримали найпростіший приклад узагальнення: перехід від конкретного значення *x* до довільного. Порядок обчислень за схемою Горнера можна записати без уточнення значення *x*. Але схема Горнера діє для довільного полінома! Тому наступний етап узагальнення – перехід до масиву коефіцієнтів полінома. Для простоти ми задамо значення масиву в ініціалізаторі. Але зауважимо, що такий підхід не обмежує загальності: елементи масиву можна також увести з консолі, прочитати з файла, обчислити чи отримати ще якимось способом. Алгоритм обчислень не зміниться від способу задання коефіцієнтів полінома.

```
int main()
{
	// 3. Узагальнення для довільних коефіцієнтів
	double a[] = { 1, 2.7, -13.5, 9.75 };
	const int n = sizeof a / sizeof *a;
	double x;
	cout << "Input x: "; cin >> x;
	double P = a[0];
	for (int i = 1; i < n; ++i) P = P * x + a[i];
```

```
	cout << " - Calculation for the array of coefficients\n";
	cout << "P(" << x << ") = " << P << '\n';
	system("pause");
	return 0;
}
```

Наступний очевидний крок – оформити обчислення значення полінома у вигляді окремої функції. Зазначимо, що таке оголошення повністю абстрагується від способу задання значень коефіцієнтів полінома та його аргумента.

```
// Алгоритм обчислення, незалежний від способу отримання
// вхідних даних і виведення результатів
double value(double x, double a[], int n)
{
	// обчислення значення полінома за схемою Горнера
	double P = a[0];
	for (int i = 1; i < n; ++i) P = P * x + a[i];
	return P;
}
```

Тепер обчислення матимуть такий вигляд (ми навіть можемо дозволити собі швидко отримати значення полінома для декількох значень аргумента):

```
int main()
{
	// 4. Узагальнення до окремого алгоритму
	double a[] = { 1, 2.7, -13.5, 9.75 };
	const int n = sizeof a / sizeof *a;
	double x;
	cout << "Input x: "; cin >> x;
	cout << " - Calculation by the function\n";
	cout << "P(" << x << ") = " << value(x, a, n) << '\n';
	cout << "P(" << x-1 << ") = " << value(x-1, a, n) << '\n';
	system("pause");
	return 0;
}
```

Оголошення функції якісно відрізняється від інших варіантів програми. Перші три з них описують «живі» обчислення, які працюють одразу після запуску програми. Функція – це ще не обчислення. Це лише план обчислень. Його можна «оживити» після того, як будуть задані всі аргумент функції та викликано її.

Програмування в термінах функцій використовує вищий рівень абстракцій, ніж програмування в термінах інструкцій. Початківцю буває складно піднятися на нього. Але для тих, хто вільно володіє функціями, можна рухатися далі, до нових способів узагальнення.

Чи завжди поліном має дійсні коефіцієнти? Ні. Коефіцієнти можуть бути також цілими, раціональними, комплексними. Навіть для дійсних чисел можна використати різні типи: *double* або *float*. Чи можна оголошену нами функцію *value* використати для всіх цих типів? Ні, для кожного типу потрібно мати окрему функцію. Мова програмування C++ дозволяє перевантажувати функції для різних типів параметрів, тому ми могли б оголосити цілу сім'ю однойменних функцій *value* для всіх перелічених типів чисел. Тим більше, що алгоритм обчислень у всіх випадках буде однаковим. Проте мова C++ надає кращий інструмент для таких потреб: ми можемо оголосити шаблон функції, що не залежить від типу коефіцієнтів полінома.

```
// Алгоритм обчислення, незалежний також від типу даних
template <typename TNum>
TNum Value(TNum x, TNum a[], int n)
{
	TNum P = a[0];
	for (int i = 1; i < n; ++i) P = P * x + a[i];
	return P;
}
```

Шаблон оголошують за допомогою ключового слова *template*. У кутніх дужках після нього вказують параметри шаблона. Здебільшого параметрами шаблона є імена типів. Легко бачити, що шаблон *Value* дуже схожий на функцію *value*. Лише тип *double* замінили на узагальнений тип *TNum*. Якщо замість *TNum* підставити, наприклад, *int*, то отримаємо функцію для обчислення значення цілочислового полінома. Скористаємося цією можливістю в новій програмі.

```
#include <complex>
typedef std::complex<double> Complex;

int main()
{
	// 5. Узагальнення типу коефіцієнтів
	double a[] = { 1, 2.7, -13.5, 9.75 };
	const int n = sizeof a / sizeof *a;
	double x;
	cout << "Input x: "; cin >> x;
	int b[] = { 1, 3, -14, 10 };
	const int m = sizeof b / sizeof *b;
	Complex c[] = { Complex(1.5,0.5),Complex(2.0,-1.5),Complex(-4.0,0.0) };
	const int k = sizeof c / sizeof *c;
	Complex z(1, 0);

	cout << " - Calculation by the template of function\n";
	cout << "P(" << x << ") = " << Value<double>(x, a, n) << '\n';  // 1
	cout << "P(" << x - 1 << ") = " << Value(x - 1, a, n) << '\n';  // 2
	cout << "Q(" << 2 << ") = " << Value(2, b, m) << '\n';          // 3
	cout << "Q(" << 1 << ") = " << Value<int>(1, b, m) << '\n';     // 4
	cout << "R(" << z << ") = " << Value(z, c, k) << '\n';          // 5
	system("pause");
	return 0;
}
```

Тут обчислено значення трьох різних поліномів: з дійсними, цілими та комплексними коефіцієнтами. У виклику шаблонної функції можна вказувати в кутніх дужках тип, для якого її використовують. Якщо такої вказівки нема, то компілятор сам виводить тип, яким потрібно замінити параметр шаблона.

Раніше ми сказали, що функція – це ще не дія, лише план майбутньої дії. Так от, шаблон функції – це ще не функція, а тільки план написання функції. За цим планом повний текст функції для конкретного типу, потрібного програмістові, генерує сам компілятор.

Неявне створення екземпляра шаблонної функції стається тоді, коли вперше зустрічається виклик для нового типу даних. У наведеній програмі в рядку, позначеному коментарем «1», буде згенеровано текст функції *Value<double>*. У рядку «2» генерування не відбувається, бо потрібна функція *Value<double>* уже є. У рядку «3» потрібно обчислити значення за масивом цілих чисел, тому компілятор виведе тип *int* і згенерує функцію *Value<int>*. Ще один екземпляр шаблона з'явиться у рядку «5» – функція *Value<std::complex<double>>* для полінома з комплексними коефіцієнтами. Всього в програмі буде використано три перевантажені функції *Value*. Це означає, що використання шаблонів скорочує час написання програм (не потрібно багатократно копіювати код, що відрізняється лише типом даних), але не зменшує розміри компільованого коду, бо всі потрібні екземпляри функцій у ньому присутні.

Зазвичай екземпляри шаблона генеруються неявно в місці виклику. Проте програміст може змусити компілятора згенерувати функцію за його вимогою. Для цього використовують оголошення такого вигляду:

```
template float Value<float>(float x, float a[], int n);
```

Продовжимо наші вправи з шаблонами функцій і оголосимо ще один. Хотілося б мати засіб для виведення полінома на консоль. Оголосимо для цього шаблон функції.

```
// Алгоритм друкування довільного полінома
template <typename TNum>
void Print(TNum a[], int n)
{
	if (a[0] != 1) std::cout << a[0];
	for (int i = 1; i < n; ++i)
	{
		std::cout << " x^" << n - i << ' ';
		if (a[i] > 0) std::cout << '+';
		std::cout << a[i];
	}
	std::cout << '\n';
}
```

Це не дуже досконалий алгоритм. Він друкує всі коефіцієнти без розбору, в тому числі одиниці та нулі. Читач може сам удосконалити його. Нам же для навчання вистачить і такого. Скористаємося ним, щоб побачити, задані в програмі поліноми.

```
	Print(a, n);   // x^3 +2.7 x^2 -13.5 x^1 +9.75
	Print(b, m);   // x^3 +3 x^2 -14 x^1 +10
	Print(c, k);   // Виникнуть проблеми !!!
```

Многочлени з дійсними та цілими коефіцієнтами матимуть досить пристойний вигляд, а от комплексний надрукувати не вдасться. Виникне помилка компілювання, оскільки для типу комплексних чисел не визначено оператор «>». Щоб виправити її використаємо ще одну синтаксичну можливість. У програмах C++ можна оголошувати явні спеціалізації шаблонів. Якщо шаблон – загальне правило, то спеціалізація – виняток з правила (для окремого типу даних). Спеціалізація для типу комплексних чисел матиме такий вигляд:

```
template<>
void Print(std::complex<double> a[], int n)
{
	std::cout << a[0];
	for (int i = 1; i < n; ++i)
	{
		std::cout << " x^" << n - i << " + " << a[i];
	}
	std::cout << '\n';
}
```

На завершення обговорення шаблонів функцій зазначимо, що їх, як і функції можна перевантажувати. Наприклад, ще один шаблон *Value* (див. нижче) відрізняється від оголошеного раніше сигнатурою, тому безпроблемно перевантажує його.

```
// Обчислення середнього арифметичного чисел довільного типу
template <typename TNum>
TNum Value(TNum a, TNum b)
{
	return (a + b)*0.5;
}
```

Наведемо ще два приклади використання шаблонів функцій. У попередній лекції ми розглядали оголошення класу *RectangularPyramid*, що наслідував від *Shape3D* і від *Rectangle*. Тоді виникла проблема з оператором виведення на консоль: такі оператори було оголошено для обох базових класів, і компілятор не міг вибрати, який з них використовувати для виведення піраміди. Щоб виправити помилку, потрібно було визначити ще один оператор – для типу *RectangularPyramid*. Три однакових за змістом оператори (кожен викликає метод *printOn* об'єкта) – чи не забагато? Щоб не продукувати стільки одноманітного коду, ми могли б оголосити шаблон функції, і нехай всю нудну роботу зробить компілятор.

```
template<typename T>
ostream& operator<<(ostream& os, const T& obj)
{
	obj.printOn(os); return os;
}
```

Такий шаблон підійде для виведення в потік об'єктів довільних класів, які мають метод *printOn*.

Ще один приклад стосується передавання функції масиву довільного розміру. Зазвичай для цього використовують пару параметрів: вказівник на початок масиву і розмір масиву. Мені завжди хотілося позбутися того другого параметра, щоб задавати в місці виклику тільки ім'я масиву. Так було б зручніше. У шаблонах можна використовувати різні параметри: не тільки параметри типу, а й числові. Скористаємося цією можливістю і винесемо розмір масиву в параметри шаблона. Напишемо якусь простеньку функцію, наприклад, виведення масиву на консоль. Ось, що вийде:

```
// шаблони функції, що друкує масив довільного розміру, заданого параметром шаблону
template <typename T, int N>
void print(T x[])
{
	for (int i = 0; i < N; ++i) cout << '\t' << x[i];
	cout << '\n';
}
```

Заголовок функції справді звільнився від другого параметра. Спробуємо скористатися цим чудовим шаблоном і надрукувати масив п'яти цілих:

```
	int b[5] = { 10, 20, 30, 40, 50 };
	print(b);
```

На жаль, така спроба приречена на невдачу. Компілятор повідомить, що він не в змозі вивести з форми виклику значення всіх параметрів шаблону. Те, що *T = int*, він встановив, а значення *N* для нього ніяк не пов'язане з *b*. Правильна форма виклику *print* така:

```
	print<int,5>(b);
```

Це не спрощує ситуацію, а якраз навпаки. Проте, бажаний спосіб оголошення шаблона існує! Значення *N* можна пов'язати з параметром функції, якщо визначити його так:

```
// шаблони функції, що друкує масив довільного розміру, виведеного компілятором
template <typename T, int N>
void printMas(T (&x)[N])
{
	for (int i = 0; i < N; ++i) cout << '\t' << x[i];
	cout << '\n';
}
```

Використання *printMas* матиме вигляд, як у програмі нижче:

```
int main()
{
	const int n = 3;
	int a[n] = { 1, 2, 3 };
	int b[5] = { 10, 20, 30, 40, 50 };
	double c[4] = { 1.5, 3.4, 6.7, 9.8 };
	// За аргументом функції print можна вивести тип елементів масиву,
	// але не його розмір. Тому вказанна яргументів шаблону - обов'язкове
	print<int,n>(a);
	// Шаблон printMas має "мудріший" аргумент функції, оскільки членами
	// типу є і тип елемента, і розмір масиву. Тому всі аргументи шаблону
	// виводяться неявно
	printMas(a);
```

```
	printMas(b);
	printMas(c);
	system("pause");
	return 0;
}
```

Зауважте, що для кожного виклику *printMas* компілятор згенерує окремий екземпляр функції.

Шаблони функцій – один з інструментів узагальненого програмування мовою C++. Далі ми обговоримо ще один: оголошення та використання шаблонів класів.

Найбільш відповідним кандидатом на узагальнення типу даних є клас-контейнер. Справді, функціональність таких класів зосереджена на тому, *як* вони зберігають дані, а не *які* дані. Для різних типів даних реалізація універсального контейнера буде однаковою, тому природньо оголошувати контейнери як шаблони. Проілюструємо ці міркування оголошеннями різних варіантів стека. Використаємо стек на основі масиву, копіювання стека заборонимо, оголосивши конструктор копіювання і оператор присвоєння в закритій частині класу.

Відомим нам способом відокремлення типу від коду є використання *typedef*.

Файл stack.h

```
// перша спроба зробити оголошення класу незалежним від типу даних
// замість unsigned long можна вказати довільний тип

typedef unsigned long Item;

// Стек на основі вбудованого масиву сталого розміру.
// Перевіряти коректність дій занесення/вилучення даних
// повинен користувач, копіювання стека заборонено.
class stack
{
private:
	enum { MAX = 10 };
	Item mem[MAX];
	int top;
	stack(const stack&);            // запобігає копіюванню
	stack& operator=(const stack&); // цілої структури
public:
	stack() :top(0) {}
	bool isEmpty() const { return top == 0; }
	bool isFull() const { return top == MAX; }
	void push(const Item& x) { mem[top++] = x; }
	Item pop() { return mem[--top]; }
};
```

Використання цього класу ілюструє невеличка програма:

```
#include <iostream>
#include "stack.h"

int main()
{
	const int n = 4;
	Item a[n] = { 4,25,39,478 };
	stack s;
	std::cout << " --- Source data:\n";
	for (int i = 0; i < n; ++i)
	{
		s.push(a[i]);
		std::cout << '\t' << a[i];
	}
	std::cout << '\n';
	std::cout << " --- Data from the stack:\n";
```

```
	while (!s.isEmpty()) std::cout << '\t' << s.pop();
	std::cout << '\n';
	system("pause");
	return 0;
}
```

На консолі отримаємо

```
--- Source data:
        4       25      39      478
 --- Data from the stack:
        478     39      25      4
```

Все просто і очевидно, але в описаного підходу є очевидні недоліки: в програмі можна використовувати стеки лише якогось одного типу, два стеки з різними типами даних оголосити не вдасться; щоб змінити тип даних, доведеться перекомпілювати проект. А як же виглядає використання шаблона класу? Подібно до шаблона функції.

Файл stackPtn.h

```
// Шаблон стека на основі векторної пам'яті.
// Для зберігання даних використано вбудований масив фіксованого розміру.

template <typename T>
class Stack
{
private:
	enum { MAX = 10 };
	T mem[MAX];
	int top;
	Stack(const Stack&);
	Stack& operator=(const Stack&);
public:
	Stack() :top(0) {}
	bool isEmpty() const {return top == 0;}
	bool isFull() const {return top == MAX;}
	void push(const T& x) { mem[top++] = x; }
	T pop() { return mem[--top]; }
};
```

Майже нічого не змінилося, тільки додали заголовок шаблона і замінили *Item* на *T*. Проте тепер маємо змогу використовувати в програмі стільки стеків різного типу, скільки потрібно. Для кожного нового конкретного типу даних компілятор згенерує нове оголошення класу та відкомпілює його. Приклад використання – у програмі нижче. Кутні дужки і тип в оголошенні екземпляра шаблонного класу вказувати потрібно обов'язково, на відміну від використання шаблонів функцій.

```
#include <iostream>
using std::cin;
using std::cout;
#include "stackPtn.h"

/*  ЗАДАЧА. Задано послідовність цілих чисел. Надрукуйте спочатку всі недодатні у зворотному
  порядку, а потім - всі додатні, збільшені в два з половино рази.
    ПОЯСНЕННЯ. Для обертання порядку чисел використаємо два стеки: стек цілих для всієї
  послідовності і стек дійсних для додатних. Стеки оголосимо з використанням шаблона. */

int main()
{
	Stack<int> numb;     // генерування класу Stack<int>
	Stack<double> posit; // генерування класу Stack<double>
	int x;
```

```
	while (!numb.isFull())
	{
		cout<<" ? x: "; cin>>x;
		numb.push(x);
	}
	cout << " Not positive numbers are:\n";
	while (!numb.isEmpty())
	{
		x = numb.pop();
		if (x > 0) posit.push(2.5*x);
		else cout << '\t' << x;
	}
	cout<<'\n';
	if (!posit.isEmpty()) cout<<" Positives * 2.5 are:\n";
	while (!posit.isEmpty()) cout <<'\t' << posit.pop();
	cout<<'\n';
	return 0;
}
```

На консолі отримаємо

```
 ? x: 5
 ? x: -1
 ? x: 10
 ? x: -2
 ? x: -3
 ? x: 15
 ? x: 20
 ? x: 25
 ? x: -4
 ? x: 30
 Not positive numbers are:
        -4      -3      -2      -1
 Positives * 2.5 are:
        12.5    25      37.5    50      62.5    75
```

Шаблон *Stack<T>* використовує масив сталого розміру, що є його недоліком. Виправимо його і оголосимо нову версію шаблона, що використовує динамічний масив. Змінимо також влаштування методів занесення і вилучення даних: вони тепер повертають ознаку того, чи вдалося виконати дію. Методи вийшли чималими, тому їхнє оголошення довелося розташувати окремо від оголошення класу. Зверніть увагу на те, як це зроблено: кожен метод шаблона класу оголошено як шаблон. Такі оголошення методів не переносять до *cpp*-файла: весь шаблон цілком розташовують у одному заголовковому файлі.

Файл stackDPtn.h

```
// Шаблон стека на основі векторної пам'яті.
// Для зберігання даних використано динамічний масив.
// Розмір масиву задають в конструкторі стека.

template <typename Type>
class DStack
{
private:
	enum { MAX = 10 };
	int size;
	Type* mem;
	int top;
	DStack(const DStack& st);
	DStack& operator=(const DStack& st);
public:
	explicit DStack(int ss = MAX):size(ss),top(0){ mem = new Type[size]; }
	~DStack() { delete [] mem; }
	bool isEmpty() const {return top == 0;}
	bool isFull() const {return top == size;}
```

```
	bool push(const Type& x);
	bool pop(Type& res);
};

template <typename Type> bool DStack<Type>::push(const Type &x)
{
	if (top<size)
	{
		mem[top++] = x;
		return true;
	}
	return false;
}

template <typename Type> bool DStack<Type>::pop(Type &res)
{
	if (top>0)
	{
		res = mem[--top];
		return true;
	}
	return false;
}
```

Створення такого стека в програмі матиме вигляд:

```
DStack<int> stA(5);
// ... обчислення count
DStack<double> stB(count);
```

Шаблони класів надають програмістові ще один спосіб оголошення стека потрібного розміру. Ми можемо винести розмір внутрішнього масиву в параметри шаблона. Тоді отримаємо ще один варіант стека.

Файл stackAPtn.h

```
// Шаблон стека на основі векторної пам'яті.
// Для зберігання даних використано вбудований масив заданого розміру.
// Розмір масиву є параметром-виразом шаблону.

template <typename Type, int N>
class AStack
{
private:
	Type mem[N];
	int top;
public:
	AStack():top(0){};
	bool isEmpty() const {return top == 0;}
	bool isFull() const {return top == N;}
	bool push(const Type& x);
	bool pop(Type& res);
};

template <typename Type, int N> bool AStack<Type,N>::push(const Type &x)
{
	if (top<N)
	{
		mem[top++] = x;
		return true;
	}
	return false;
}
```

```
template <typename Type, int N> bool AStack<Type,N>::pop(Type &res)
{
	if (top>0)
	{
		res = mem[--top];
		return true;
	}
	return false;
}
```

Оголошення в програмі:

```
AStack<int,20> all;
AStack<double,10> plus;
AStack<int,5> small;
```

Звернемо увагу читача на те, що такий варіант оголошення шаблона класу може призвести до «розбухання» коду, оскільки змінні *all* і *small*, оголошені вище – це екземпляри *різних* класів. Для кожної з них компілятор згенерує окреме оголошення класу.

Шаблон *AStack<T,N>* демонструє також те, що у шаблонів може бути декілька параметрів. Параметри шаблона, як і параметри функції можуть мати значення за замовчуванням.

Відомий стандартний шаблон з двома параметрами-типами *std::pair<T1,T2>*. Його використовують для швидкого поєднання в одному об'єкті двох значень. Наприклад, для того, щоб повернути з функції два результати одночасно. Щоб використати його в програмі, достатньо приєднати заголовковий файл *utility*.

```
template<class T1, class T2> struct pair
{
	T1 first; T2 second;
	typedef T1 first_type;
	typedef T2 second_type;
};
```

Для цього типу за допомогою шаблонів функцій перевантажено низку операторів, оголошено допоміжні функції. За інформацією про них можна звернутися до документації.

На завершення лекції зауважимо, що ключові слова *class* і *typename* в оголошенні шаблонів використовують як синоніми, *typename* – сучасний варіант. Визначення шаблона повністю розташовують у заголовковому файлі: методи визначають у класі, або відразу після нього (як в оголошенні *DStack* чи *AStack*).

Про інші способи використання шаблонів йтиметься у наступних лекціях.