# Лекція 8. Використання шаблонів класів і функцій

1. *Наслідування шаблонів* `template ← template; template ← class; class ← template.`
2. *Включення шаблонів.*
3. *Вкладений шаблон.*
4. *Шаблон – параметр шаблона.*
5. *Клас з шаблонним методом.*
6. *Шаблон і дружні функції.*

У попередній лекції ми познайомилися з шаблонами класів. Один з запропонованих шаблонів описував стек:

```
template <typename T> class Stack
{
protected:       // зверніть увагу: private змінено на protected !!!
	enum { MAX = 10 };
	T mem[MAX];
	int top;
	Stack(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]; }
	Stack& operator=(const Stack& st)
	{	// присвоювати можна лише порожні стеки
		if (&st != this) top = 0;
		return *this;
	}
};
```

У програмі ми використовували його для оголошення стеків з різними типами даних. Тут *Stack<int>* та *Stack<double>* – імена двох різних класів.

```
	Stack<int> numb;
	Stack<double> posit;
```

Як ще можна використовувати оголошення шаблона класу? Коротка відповідь: так само, як оголошення класу. Спробуємо проілюструвати прикладами всеможливі способи використання.

***Шаблон можна наслідувати.***

Усі шаблони стеків з попередньої лекції мають недолік: їм бракує методу, що дає змогу дізнатися, яке значення міститься у вершині стека без зміни самого стека. Відоме правило ООП: якщо бракує функціональності, оголоси підклас. То ж виправимо недолік у новому шаблоні, що наслідує оголошений раніше. Новий метод потребуватиме доступу до даних стека, тому все вдасться, якщо поля даних базового шаблону мають рівень захисту *protected*.

```
// Шаблон класу можна використати як базовий для створення підкласів.
// Новий клас-шаблон розширює можливості батьківського.

template <typename T> class ExtStack : public Stack<T>
{
public:
	//повертає значення з вершини без зміни стека
	T peek() const { return mem[top - 1]; }
};
```

У результаті отримуємо два шаблони: *Stack<T>* і *ExtStack<T>* – пов'язані типом *T*. Якщо в програмі оголосити, наприклад, *ExtStack <char> operations*, то компілятор згенерує оголошення двох класів: базового *Stack<char>* і похідного *ExtStack<char>*. Таким чином, після підстановки «char» замість «T» отримуємо звичайне наслідування класів.

Ми можемо використовувати й інші варіанти наслідування: звичайний клас може наслідувати конкретизований шаблон, або ж на базі звичайного класу можна оголосити шаблон.

*Фраґмент файла eStack.h*

```
// Конкретизований шаблон також може бути базовим класом

class StackDbl : public ExtStack<double>
{
public:
	// дає змогу побачити внутрішність стека
	void print() const;
};
```

*Файл eStack.cpp*

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

void StackDbl::print() const
{
	if (top == 0) std::cout << "Stack of doubles is empty.\n";
	else
	{
		std::cout << "  Stack of doubles:\ntop = " << top << "\nmem: ";
		for (int i = 0; i < top; ++i) std::cout << '\t' << mem[i];
		std::cout << '\n';
	}
}
```

У попередніх ми працювали з класом *Student*. Пригадуєте, студент – це особа, що має колекцію оцінок. Оцінки ми задавали масивом цілих чисел. Але, чому саме цілих, адже сучасний студент отримує три оцінки: числом за 100-бальною шкалою, словом за національною шкалою і літерою за європейською. Ми могли б оголосити шаблон класу студент, параметризований типом оцінки. А для самих оцінок використати значення типу *int*, *char*, *string*, або спеціально оголошений тип, що поєднує всі три складові. Новий шаблон не будемо перевантажувати функціональністю та складною структурою. Нам розходиться тільки на тому, щоб продемонструвати, як шаблон може наслідувати від класу.

*Файл Human.h*

```
#pragma once
#include <iostream>
#include <string>

enum Gender { Male, Female };

// Особа має ім'я та стать
class Human
{
private:
	std::string name;
	Gender sex;
public:
	Human() :name("Unknown"), sex(Male) {}
	Human(const char* n, Gender s) :name(n), sex(s) {}
	// методи для читання даних екземпляра
	Gender Sex() const { return sex; }
```

```
	std::string Name() const { return name; }
	// виведення в потік
	virtual void printOn(std::ostream& os) const;
	// Особа вміє висловлюватися
	Human& say(std::string phrase)
	{	std::cout << name << ": \"" << phrase << "\"\n";
		return *this;
	}
};
// Студент - це особа, що має залікову та колекцію оцінок
// Тип оцінки не обмежено, вимоги - operator==, operator<<
template<typename TPoint>
class Student :public Human
{
private:
	std::string indNo;
	TPoint point[50] = { TPoint() };
public:
	Student() :Human(), indNo("2701234c") {}
	Student(const char* n, Gender s, const char* i) : Human(n, s), indNo(i) {}
	Student<TPoint>& setPoint(TPoint p, int i) { point[i] = p; return *this; }
	virtual void printOn(std::ostream& os) const;
};
std::ostream& operator<<(std::ostream& os, Gender gender);
std::ostream& operator<<(std::ostream& os, const Human& human);

template<typename TPoint>
inline void Student<TPoint>::printOn(std::ostream & os) const
{	Human::printOn(os);
	os << " is a student: [";
	int i = 0;
	while (!(point[i] == TPoint())) std::cout << ' ' << point[i++];
	os << " ]";
}

// Спеціальний тип для всестороннього відображення оцінки
struct Point
{
	int num;
	char letter;
	std::string word;
	Point() : num(0), letter('F'), word("unsatisfied") {}
	Point(int point);
	bool operator==(const Point& point) const
	{	return this->num == point.num;
	}
};
std::ostream& operator<<(std::ostream&os, const Point& p);
```

*Файл Human.cpp*

---

```
#include "Human.h"
void Human::printOn(std::ostream & os) const
{	os << name << " (" << sex << ')';
}
std::ostream & operator<<(std::ostream & os, Gender gender)
{	switch (gender)
	{
	case Male: os << "Male"; break;
	case Female: os << "Female"; break;
	default: os << "Unknown gender"; break;
	}
	return os;
}
std::ostream & operator<<(std::ostream & os, const Human & human)
{	human.printOn(os); return os;	}
```

```
std::ostream & operator<<(std::ostream & os, const Point & p)
{	os << '<' << p.num << ',' << p.letter << ',' << p.word << '>';
	return os;
}
Point::Point(int point)
{	num = point < 0 ? -point : point;
	if (num > 100) num = 100;
	if (num < 51)
	{	letter = 'F'; word = "unsatisfied";	}
	else if (num < 61)
	{	letter = 'E'; word = "satisfied";	}
	else if (num < 71)
	{	letter = 'D'; word = "satisfied";	}
	else if (num < 81)
	{	letter = 'C'; word = "good";	}
	else if (num < 90)
	{	letter = 'B'; word = "good";	}
	else
	{	letter = 'A'; word = "excellent";	}
}
```

*Файл Program.cpp*

```
#include "Human.h"
using std::cout;

int main()
{
	Student<int> Gen("Programmer Bill", Male, "2700120c");
	Gen.setPoint(99, 0).setPoint(85, 1).setPoint(93, 2);
	Student<std::string> Ukr("Mathematician Banakh", Male, "2700251c");
	Ukr.setPoint("good", 0).setPoint("excellent", 1).setPoint("excellent", 2);
	Student<Point> Euro("Analysist Julia", Female, "2701331c");
	Euro.setPoint(98, 0).setPoint(86, 1).setPoint(91, 2);
	cout << " --- Our students are:\n" << Gen << '\n' << Ukr << '\n' << Euro << '\n';
	system("pause");
    return 0;
}
```

***Шаблон може стати частиною іншого шаблона.***

Повернемося до обговорення різних способів влаштування стека. Ми вже використовували раніше клас, що огортає динамічний масив дійсних чисел, приховує дії з вказівником, повідомляє його розмір тощо. Такий клас легко перетворити на шаблон.

*Фраґмент файла arrayPtn.h*

```
// Шаблон динамічного масиву фіксованого розміру.
// Клас інкапсулює роботу з вказівниками, перевіряє
// правильність звертань за індексом

#include <stdexcept>
#include <sstream>

template <typename Type> class Array
{
private:
	unsigned int size;
	Type* arr;
	void checkIndex(int i) const; // для методів operator[]
public:
	Array(): arr(0), size(0) {}
	explicit Array(unsigned int n, const Type& val = 0.0);
	Array(const Type* pn, unsigned int n);
```

```
	Array(const Array& a);
	virtual ~Array();
	unsigned int arSize() const { return size; }
	Type& operator[](int i);
	const Type& operator[](int i) const;
	Array& operator=(const Array& a);
};
template<typename Type>
inline void Array<Type>::checkIndex(int i) const
{
	if (i < 0 || i >= size)
	{	std::ostringstream mess;
		// виняток міститиме докладний опис помилки з індексом
		mess << "Error in Array limits: " << i << " is a bad index.";
		throw std::out_of_range(mess.str());
	}
}
```

Тепер можемо оголосити шаблон стека, що містить шаблон масиву.

*Файл stackHPtn.h*

```
#include "arrayPtn.h"
/*
Шаблон стека на основі векторної пам'яті. Приклад відношення "has-a" між шаблонами.
Для зберігання даних використано динамічний масив-клас. Саме він інкапсулює все
керування динамічною пам'яттю, тому клас-стек суттєво спрощується. Єдина незручність
виникає з перевіркою розміру стека. Методи занесення-вилучення даних можуть
спричинити виняток (у масиві)
*/
template <typename Type>
class HStack
{
private:
	enum { MAX = 10 };
	Array<Type> mem;
	int top;
public:
	explicit HStack(int ss = MAX):mem(ss),top(0){ }
	bool isEmpty() const {return top == 0;}
	bool isFull() const {return top == mem.arSize();}
	void push(const Type& x) { mem[top] = x; ++top; }
	Type pop();
};

template<typename Type>
inline Type HStack<Type>::pop()
{
	try
	{	return mem[--top];
	}
	catch (std::out_of_range)
	{	top = 0; throw;
	}
}
```

***Шаблон можна оголосити всередині іншого шаблона.***

Ми вже маємо досвід оголошення вкладених класів: всередині класу *List* (список) ми оголошували структуру *Node* (ланка списку). Якщо спробувати перетворити такий клас на шаблон, незалежний від типу елементів, то й вкладений тип потрібно зробити шаблоном. Виявляється, вкладений шаблон можна оголошувати трьома різними способами, які відрізняються окремими деталям. Розглянемо всі три.

```
// 1. Вкладений тип Node є «неявним» шаблоном. Його повне ім'я – DblList<T>::Node
//    При оголошенні Node конкретизацію типу T можна не вказувати, бо вона залежить
//    від конкретизації DblList. Поза класом: DblList<int>::Node node;
template <typename T> class DblList
{
public:
	struct Node
	{
		T data;
		Node *prev, *next;
		Node(T x, Node* p = nullptr, Node* n = nullptr) :data(x), prev(p), next(n) {}
	};
private:
	Node* top;
	Node* bottom;
... };
// 2. Вкладений тип Node є шаблоном з залежним типом. Його повне ім'я – DblList<T>::Node<T>
//    При оголошенні Node<T> кутні дужки – обов'язкові, а тип обов'язково такий самий, як
//    у конкретизації DblList. Поза класом: DblList<int>::Node<int> node;
template <typename T> class DblList
{
public:
	template <typename T> struct Node
	{
		T data;
		Node *prev, *next;
		Node(T x, Node* p = nullptr, Node* n = nullptr) :data(x), prev(p), next(n) {}
	};
private:
	Node<T> * top;
	Node<T> * bottom;
... };
// 3. Вкладений тип Node є незалежним шаблоном. Його повне ім'я – DblList<T>::Node<D>
//    При оголошенні всередині списку Node<D> кутні дужки – обов'язкові, а тип такий, як
//    у параметрі шаблона DblList. Поза класом можна і так: DblList<int>::Node<double> node;
template <typename T> class DblList
{
public:
	template <typename D> struct Node
	{
		D data;
		Node *prev, *next;
		Node(D x, Node* p = nullptr, Node* n = nullptr) :data(x), prev(p), next(n) {}
	};
private:
	Node<T> * top;
	Node<T> * bottom;
... };
```

Ще один приклад оголошення шаблона в шаблоні. (Клас *Beta* вигадано заради прикладу.)

```
template <typename T> class Beta
{
	// містить два значення, вміє їх друкувати, виконувати обчислення
private:
	// попереднє оголошення шаблона класу
	template <typename V> class Hold;
	// використання цього шаблона
	Hold<T> q;
	Hold<int> n;
public:
	Beta(T t, int i) :q(t), n(i) {}
	void Show() const
	{	q.show(); n.show();   }
```

```
	// шаблоний метод, тип результату залежить від типу першого параметра
	template <typename U> U calc(U u, T t);
};
// визначення попередньо оголошеного шаблона класу
template <typename T> template <typename V>
class Beta<T>::Hold
{
private:
	V val;
public:
	Hold(V v = V()) :val(v) {}
	void show() const { std::cout << val << '\t'; }
	V value() const { return val; }
};
// визначення шаблона методу
template <typename T> template <typename U>
U Beta<T>::calc(U u, T t)
{	return (n.value()*u + q.value())*t;
}
```

***Шаблон може стати параметром іншого шаблона***, причому двома різними способами: при використанні та при оголошенні. Скористаємося оголошеними раніше шаблонами *Array* та *Stack*, щоб продемонструвати першу можливість. Наприклад, матрицю 3×4 можна утворити як масив масивів: масив трьох масивів, наповнених чотирма одиницями кожен.

```
	Array<Array<int>> matr(3, Array<int>(4, 1));
	for (int i = 0; i < 3; ++i)
	{
		for (int j = 0; j < 4; ++j) cout << '\t' << matr[i][j];
		cout << '\n';
	}
```

Масив п’яти стеків, чи стек масивів утворити так само легко:

```
	Array<Stack<int>> staks(5, Stack<int>());
	for (int i = 0; i < 40; ++i) staks[i % 5].push(i);
	cout << staks[1].pop() << ' ' << staks[4].pop() << '\n';
	Stack<Array<int>> arrays;
	arrays.push(Array<int>(2));
	arrays.push(Array<int>(3, 5));
	cout << arrays.pop()[0] << ' ' << arrays.pop()[0] << '\n';
```

Числа, що діляться на 5, потраплятимуть до першого стека, які дають в остачі 1 – до другого і так далі. До стека поклали масив з двох нулів і масив з трьох п’ятірок.

***При оголошенні шаблона можна явно вказати, що один з його параметрів є шаблоном***. Нижче наведено шаблон оператора введення, який може працювати з довільним шаблонним класом, який підтримує методи *clear*, *resize*, *putLast*. Імовірно, цей клас – деякий контейнер.

```
template <typename T, template <typename Y> class Cont>
std::istream& operator>>(std::istream& is, Cont<T>& C)
{
	size_t n;
	is >> n;
	C.clear().resize(n);
	for (size_t i = 0; i < n; ++i)
	{
		typename Cont<T>::ValueType x;
		is >> x;
		C.putLast(x);
	}
	return is;
}
```

Наведемо ще один приклад використання параметра-шаблона. Для цього покажемо ще один спосіб побудови стека. Послідовні контейнери, як побудований нами раніше шаблони *Array* чи *DblList*, зазвичай підтримують достатньо функцій для реалізації стека. Масив стане стеком, якщо дозволити додавати чи вилучати елементи з його закінчення і заборонити доступ усередину масиву. Таке перетворення можна виконати за допомогою оголошення нового класу, який використовує реалізацію масиву чи списку (потрібну її частину) і надає відповідний стекові інтерфейс. Подивимося, як це зробити за допомогою шаблонів.

```
#include "DblList.h"
#include "Array.h"

template <typename T, template <typename T> class Cont = DblList>
class Stack
{
private:
	Cont<T> mem;
public:
	bool isEmpty() const { return mem.size() == 0; }
	void push(const T& x) { mem.putLast(x); }
	T pop() { return mem.getLast(); }
	T& peek() { return mem.back(); }
};
```

Видно, що параметр *Cont* шаблона *Stack* є шаблоном, що залежить від одного узагальненого типу *T* – того ж, що й шаблон *Stack*. Видно також, що він має значення за замовчуванням і, тому, є необов'язковим. Хто ж такий *Cont*? Цей тип мав би підтримувати методи *size(), putLast(), getLast(), back()*. Класи *Array* та *DblList* серед інших мають такі методи, тому в програмі можемо використовувати такі стеки:

```
	Stack<int> S;
	Stack<int, Array> T;
```

У першому випадку стек цілих використовує реалізацію за замовчуванням, список, а в другому – явно вказаний масив.

***Звичайний клас може містити шаблонний метод.***

Наприклад, оголошений раніше клас *Beta<int>* містить шаблонний метод *calc<U>*. Стандартний клас *string* має багато конструкторів, серед яких є і шаблонні.

```
string(size_t _Count, char _Ch);
template <class InputIterator> string(InputIterator _First, InputIterator _Last);
```

***Шаблон класу може мати дружні функції***: звичайні (нешаблонні), шаблони функцій, пов'язані типом, і «вільні» шаблони функцій.

```
// Шаблон класу з нешаблонними дружніми функціями
template <class T> class HasFriend
{
	friend void counts(); // дружня до ВСІХ спеціалізацій

	//   звичайна функція з шаблонним параметром
	// у програмі потрібно оголосити по одній такій функції для кожної зі спеціалізацій
	// шаблона HasFriend<тип>, отримаємо декілька перевантажених функцій, кожна з яких
	// дружня до «своєї» спеціалізації
	friend void report(HasFriend<T>&);
	...
};
// Шаблон класу з пов'язаними дружніми шаблонами функцій
// 1. Попереднє оголошення шаблонів функцій
template <typename T> void counts();
template <typename T> void report(T&);
```

```
// 2. Визначення шаблона класу з дружніми функціями
template <typename TT>
class HasFriendT
{
	// спеціалізація шаблона count типом шаблона HasFriendT
	friend void counts<TT>();
	// спеціалізація шаблона report, тип шаблона HasFriendT виводиться компілятором
	friend void report<>(HasFriend<TT>&);
	...
};
// 3. Визначення шаблонів функцій
template <typename T> void counts()
{ ... }

// Шаблон класу і дружній шаблон функції, не пов’язаний типом
template <typename TT>
class ManyFriend
{
	// кожна спеціалізація функції дружня до кожної спеціалізації класу
	template <typename C, typename D> friend void show2(C&, D&);
	...
};
```