Лекція 7. Узагальнення структури. Узагальнення алгоритмів

1. *Від конкретних алгоритмів до загального.*
2. *Маскування структури за допомогою спеціального класу.*
3. *Налаштування загального алгоритму під конкретні потреби.*

У попередніх лекціях ми довідалися, як робити оголошення функції (чи класу) незалежним від типу даних: за допомогою шаблона, параметром якого є тип. Продовжимо вивчення способів узагальнення. Покажемо, що функцію можна зробити незалежною не тільки від типу даних, які вона опрацьовує, але й від структури, в якій вони зберігаються. Почнімо з конкретних прикладів!

Задача 1. Оголосіть функцію, яка знаходить у масиві цілих чисел елемент, що містить задане значення.

Які параметри прийматиме така функція? – Шукане значення і масив (масив і його розмір). Що повертатиме така функція? – Індекс знайденого елемента, або –1, якщо такого значення нема.

```
typedef int ValueType;
int findValInMas(ValueType x, ValueType a[], int n)
{
	for (int i = 0; i < n; ++i)
	{
		if (x == a[i]) return i;
	}
	return -1;
}
```

Для перебирання елементів масиву добре підходить інструкція *for* – саме її й використали.

Задача 2. Оголосіть функцію, яка знаходить у лінійному однозв'язному списку цілих чисел елемент, що містить задане значення.

Які параметри прийматиме така функція? – Шукане значення і вказівник на першу ланку списку (розмір передавати не потрібно, бо список містить «природну» ознаку закінчення – порожній вказівник у полі зв'язку останньої ланки). Що повертатиме така функція? – Вказівник на ланку зі знайденим елементом, або *nullptr*, якщо такого значення нема.

```
struct Node
{
	ValueType data;
	Node* next;
	Node(ValueType x, Node* ptr = nullptr) :data(x), next(ptr){}
};
Node* findValInList(ValueType x, Node* L)
{
	while (L != nullptr && L->data != x) L = L->next;
	return L;
}
```

Перебір ланок списку триває, поки не досягнуто закінчення списку, або поки не знайдено шукане значення.

Задача 3. Дайте відповідь на запитання, чи ми розв'язали *різні* задачі, чи, можливо, однакові?

Очевидна відповідь – так, звичайно ж, різні! У задачі 1 мова йде про масив, а в задачі 2 – про список. Це різні структури даних. Функції побудовано також зовсім різні. Одна з них працює з елементами масиву, а інша – з вказівниками на структури.

Аргументи дуже переконливі, але спробуйте подивитися на запитання більш загально. Чим схожі задачі 1 та 2? В обох випадках мова йде про пошук значення в деякій колекції, виконують його послідовним перебором елементів колекції. Масив і список також мають спільні риси! Обидва контейнери можна охарактеризувати одним словом – це *послідовності* значень (і масив, і список містять скінчену кількість цілих чисел, розташованих одне за одним у певному порядку).

Отже, на рівні концепцій задачі 1 та 2 – однакові! Функції *findValInMas* і *findValInList* мають різний зовнішній вигляд через структурні та синтаксичні особливості масиву і списку, але розв'язують концептуально однакові задачі. То, можливо, нам вдасться записати їх однаково?

Задача 4. Запишіть функцію, або шаблон функції, яка знаходить задане значення в послідовності.

Почнемо зі змін функції відшукання значення в масиві. Відомо, що масив у функцію можна передавати не лише за допомогою вказівника на перший елемент і розміру масиву. Ще один спосіб – це передавання пари вказівників, які задають діапазон векторної пам'яті, яку має опрацювати функція. Перший вказівник пари вказує на початок масиву, а другий – на закінчення останнього елемента масиву. В літературі також можна зустріти таке формулювання: «на елемент, наступний після останнього».

Зміни торкнуться і тіла функції. Замість інструкції *for* можемо використати *while*, у якій належним чином задамо умову продовження пошуку. Зверніть увагу на зміну в імені функції: воно тепер починається з великої літери.

```
ValueType* FindValInMas(ValueType x, ValueType* start, ValueType* end)
{
	while (start != end && *start != x) ++start;
	return start;
}
```

Тут **start* – розіменування вказівника, яке означає доступ до елемента масиву, а оператор інкременту змушує вказівник переходити до наступного елемента масиву. Функція повертає вказівник на знайдений елемент, або *end*, якщо масив не містить шуканого значення.

Вийшло дуже подібно до функції *findValInList* пошуку в списку. Проте вона мала два параметри, а не три. Цю відмінність дуже легко виправити: додамо до її сигнатури третій параметр, який вказуватиме на закінчення діапазону списку, в якому потрібно виконати пошук. Таке доповнення тільки збагатить функцію, оскільки дозволить користувачеві шукати в частині списку. Якщо ж потрібно буде переглянути весь список, то третім параметром задаватимемо *nullptr*. Отримаємо таке:

```
Node* findValInListNew(ValueType x, Node* start, Node* end)
{
	while (start != end && start->data != x) start = start->next;
	return start;
}
```

Функції *FindValInMas* та *findValInListNew* майже однакові, проте залишилися відмінності, які подолати не вдалося:

	масив	список
доступ до елемента	`*start`	`start->data`
перехід до наступного елемента	`++start`	`start = start->next`

Очевидно, що «мова вказівників» на елементи масиву відрізняється від «мови структур» для доступу до полів ланки списку. Функції використовують різні інтерфейси доступу до даних. Хтось би мав узгодити ці інтерфейси. В об'єктно-орієнтованому програмування діє правило: якщо ти потребуєш певної функціональності, оголоси клас, що її надає. Ми так і зробимо і оголосимо новий клас, який «розуміє» влаштування списку, вміє доступатися до його даних та перебирати його ланки, але доступ до цих дій користувачеві надає через інтерфейс, як у вказівника.

```
// спеціальний клас-адаптер для зручного доступу до списку
class Iter
{
private:
	Node* L;
public:
	Iter(Node*ptr = nullptr) :L(ptr){}
	ValueType& operator*()
	{
		return L->data;
	}
	Iter& operator++()
	{
		L = L->next;
		return *this;
	}
	bool operator!=(const Iter& it)
	{
		return this->L != it.L;
	}
};
```

Клас *Iter* інкапсулює вказівник на ланку та методи доступу до її частин. У коментарі перед оголошенням клас не випадково названо адаптером. Він адаптує інтерфейс доступу до ланки списку до інтерфейсу доступу «як вказівник». Тепер оператор розіменування означає доступ до поля *data*, а оператор інкременту – перехід за вказівником *next*. Ми перевизначили також оператор порівняння, оскільки новій функції пошуку доведеться працювати не з вказівниками на ланки, а з екземплярами класу *Iter*, і виконувати перевірку досягнення кінця списку. Тепер можемо записати:

```
// Використання "ітератора" повністю маскує структуру списку.
// Функція ідентична до FindValInMas
Iter FindValInList(ValueType x, Iter start, Iter end)
{
	while (start != end && *start != x) ++start;
	return start;
}
```

Мети досягнуто, але як користуватися такою незвичною функцією? Продемонструємо це фраґментом програми.

```
	int a[n] = { 1, 2, 3, 4, 5 };
	Node * L = buildList(a, n);
	// Побудова ітераторів
	Iter startL = Iter(L);
	Iter endL = Iter();
	// Пошук
	Iter itr = FindValInList(4, startL, endL);
```

```
	// Перевірка, чи пошук був успішним
	if (itr != endL)
	{
		cout << "The value 4 found in the List\n";
		*itr = 9;
		cout << "The list after change\n";
		PrintList(L, nullptr);
	}
	else cout << "The value not found\n";
```

Якщо вже нам вдалося побудувати однакові функції, то залишилося перетворити їх на шаблон. Його параметрами будуть два типи: тип значень, що містяться в послідовності, та тип ітераторів послідовності.

```
template <typename ValueType, typename Iter>
Iter FindValInSuccession(ValueType x, Iter start, Iter end)
{
	while (start != end && *start != x) ++start;
	return start;
}
```

Перш ніж наводити приклади використання цього шаблона, попрацюємо ще з оголошенням списку. Його також варто перетворити на шаблон, щоб до списку можна було заносити дані довільних типів. Оголошення ланки автоматично перетвориться на вкладений шаблон. Як бути з класом ітератора? Все-таки, він тісно пов'язаний з класом контейнера, слугує для доступу до нього. Оголошувати ітератор зовсім окремо було б не дуже добре, тому оголосимо його *всередині* класу контейнера, в його відкритій частині, щоб користувачі контейнера могли створювати ітератори списку. Додатковим сервісом була б можливість запитувати в контейнера про його ітератори, тому доповнимо інтерфейс класу *List* двома новими методами: *begin()* повертатиме ітератор на початок списку, а *end()* – на кінець. Шаблон списку тепер має такий вигляд:

```
// Лінійний однозв'язний список
template <typename ValueType> class List
{
private:
	// Ланка списку містить елемент даних і вказівник на наступну ланку
	struct Node
	{
		ValueType data;
		Node* next;
		Node(ValueType x, Node* ptr = nullptr) :data(x), next(ptr){}
	};
	// Голова списку (вказівник на першу ланку)
	Node * Head;
public:
	// Клас ітератора. Ітератор використовують для обходу ланками списку
	class Iter
	{
	private:
		Node* L;
	public:
		Iter() :L(nullptr){}
		Iter(List& lst) :L(lst.Head) {}
		Iter(Node* node) :L(node) {}
		ValueType& operator*() { return L->data; }
		Iter& operator++()
		{
			L = L->next;
			return *this;
		}
		bool operator!=(const Iter& it) { return this->L != it.L; }
```

```
	};
	List(ValueType a[], int n)
	{
		this->Head = new Node(a[n - 1]);
		for (int i = n - 2; i >= 0; --i)
		{
			this->Head = new Node(a[i], this->Head);
		}
	}
	~List()
	{
		while (Head != nullptr)
		{
			Node* victim = Head;
			Head = Head->next;
			delete victim;
		}
	}
	// допоміжні методи для отримання ітераторів:
	// ... на початок списку
	Iter begin() const { return Iter(Head); }
	// ... на ланку "наступну після останньої"
	Iter end() const { return Iter(); }
};
```

Тепер наведемо приклади використання шаблону функції пошуку в послідовності для опрацювання масиву і списку цілих чисел.

```
	int a[n] = { 1, 2, 3, 4, 5 };

	cout << " --- Array\n";
	// ValueType == int, Iter == int*
	int * pos = FindValInSuccession(3, a, a + n);
	if (pos != a + n)
	{
		cout << "The value 3 found at position " << pos - a << '\n';
	}
	else
	{
		cout << "The value not found\n";
	}

	// Тепер список - екземпляр шаблону
	List<int> Lst(a, n);
	cout << " --- List\n";
	// ValueType == int, Iter == List<int>::Iter

	// Ітератори можемо створити «вручну»
	List<int>::Iter Itr = FindValInSuccession(3, List<int>::Iter(Lst), List<int>::Iter());
	if (Itr != List<int>::Iter())
		cout << "The value 3 found in the List\n";
	else
		cout << "The value not found\n";

	// ... або попросити їх у списку
	List<int>::Iter startIt = Lst.begin();
	List<int>::Iter endIt = Lst.end();
	Itr = FindValInSuccession(8, startIt, endIt);
	if (Itr != endIt)
		cout << "The value 8 found in the List\n";
	else
		cout << "The value not found\n";
```

Ми повністю розв'язали задачу 4. Шаблон *FindValInSuccession* чудово працює і з масивом, і зі списком. За допомогою класу *Iter* нам вдалося відокремити алгоритм пошуку від структури контейнера, узагальнити його для різних структур. До речі, написаний нами шаблон функції може опрацювати і нешаблонний клас список, згаданий раніше в лекції, і стандартні контейнери з бібліотеки STL (Standard Template Library).

Та чи залишилося в *FindValInSuccession* хоч одне «конкретне» місце? Чи не можна узагальнити ще щось? При уважному розгляді тексту шаблона функції з'ясовується, що вона все ще не досить загальна: для пошуку вона використовує конкретний критерій **start* != *x*. Але ж пошук додатного значення, чи, наприклад, кратного трьом принципово не відрізняється від вже –описаного. Тому і критерій пошуку можемо відокремити від алгоритму пошуку. Суть критерію в тому, що елемент контейнера випробовують на відповідність деякій умові. Зробимо цю умову параметром нового шаблона.

```
template <typename Predic, typename Iter>
Iter FindAnyInSuccession(Predic cond, Iter start, Iter end)
{
	while (start != end && !cond(*start)) ++start;
	return start;
}
```

Тут *Predic* – це тип, для якого визначено оператор круглі дужки, що приймає один аргумент і повертає значення типу *bool*. Таким типом, наприклад, може бути вказівник на функцію з відповідною сигнатурою. Використовувати оновлений шаблон не набагато складніше від попереднього.

```
	// за допомогою лямбда загальний алгоритм можна налаштувати на різні умови виконання
	// нижче - звичайний пошук значення
	Itr = FindAnyInSuccession([](int x){ return x == 0; }, Lst.begin(), Lst.end());
```

Вказувати тип аргумента лямбда-виразу в цьому прикладі потрібно обов'язково, оскільки компілятор не може вивести його самостійно через загальність типу *Predic*. Тепер з шаблоном *FindAnyInSuccession* ми справді можемо знаходити в послідовностях елементи, які задовольняють різноманітні вимоги.

У попередніх лекціях ми використовували клас *ArrayDb*, що інкапсулює масив дійсних чисел. Саме час перетворити його на шаблон і пристосувати до використання з шаблонами *FindValInSuccession* та *FindAnyInSuccession*.

```
template <typename T>
class Array
{
private:
	unsigned int size;
	T* arr;
public:
	class Iter
	{
	private:
		T * ptr;
	public:
		Iter(T* p = nullptr) :ptr(p){}
		T& operator*() { return *ptr; }
		Iter& operator++() { ++ptr; return *this; }
		bool operator!=(const Iter& it) const { return this->ptr != it.ptr; }
	};

	Array() : arr(0), size(0) {}
	explicit Array(unsigned int n, T val = T());
	Array(const T* pn, unsigned int n);
	Array(const Array& a);
```

```
        virtual ~Array() { delete[] arr; }
        unsigned int arSize() const { return size; }
        T average() const
        {
                T sum = T();
                for (size_t i = 0; i < size; i++) sum += arr[i];
                return sum / size;
        }
        T& operator[](int i)
        {
                if (i<0 || i >= size) throw std::out_of_range("Array: index out of range");
                return arr[i];
        }
        const T& operator[](int i) const
        {
                if (i<0 || i >= size) throw std::out_of_range("Array: index out of range");
                return arr[i];
        }
        Array& operator=(const Array& a);
        Iter begin() { return Iter(arr); }
        Iter end() { return Iter(arr + size); }

        template <typename T>
        friend ostream& operator<<(ostream& os, const Array<T>& a);
};
```

Що змінилося від такого перетворення? Тип *double* всюди замінили на *T*; нулі замінили на значення типу *T* за замовчуванням (за допомогою конструктора за замовчуванням) – для числових типів це і є нуль; дружній оператор виведення перетворився на дружній шаблон. Тепер шаблон класу і шаблон функції-оператора пов'язані спільним типом *T*. Як тільки буде оголошено *Array<int>*, компілятор згенерує визначення класу та визначення функції *operator<< <int>(ostream&, const Array<int>&)*.

У відкритій частині шаблону з'явився вкладений шаблон класу *Iter* і два методи: *begin* та *end* – що повертають екземпляри ітератора на початок і кінець масиву відповідно. Тепер *Array* можна використовувати спільно з *FindValInSuccession*, наприклад.

```
        Array<int> A(a, 5);
        Array<int>::Iter itFind = FindValInSuccession(5, A.begin(), A.end());
```

Виявляється, від оголошеного раніше шаблону *FindAnyInSuccession* можна добитися більшого. Ви справді вважаєте, що він вміє лише знаходити елементи? Тоді погляньте на такий виклик.

```
// "пошук", що друкує послідовність значень
FindAnyInSuccession([](int x){ cout << '\t' << x; return false; }, Lst.begin(), Lst.end());
```

Лямбда-вираз спеціально сконструйовано так, щоб він завжди повертав хибу і заставив функцію перебрати всі елементи послідовності та виконати з ними певну дію, у нашому прикладі це виведення на консоль.

Ми можемо продовжити наші вправи і використати *FindAnyInSuccession* для обчислень:

```
        // "пошук", що обчислює суму елементів списку
        int total = 0;
        FindAnyInSuccession([&total](int x){ total += x; return false; },
                Lst.begin(),Lst.end());
        cout << "The total = " << total << '\n';
```

Треба сказати, що таке використання *FindAnyInSuccession* – свого роду хуліганство. Адже таке «нетрадиційне» застосування пошуку тільки заплутує код. Для друку послідовності та накопичення суми варто було б оголосити інші шаблони. Приклади наведені з однією

метою: продемонструвати читачеві можливості узагальнення. І пошук, і друк, і обчислення суми *перебирають елементи послідовності*. Ця спільність і зробила можливими наведені приклади.

Лекція мала на меті приготувати читача до вивчення бібліотеки STL, що є частиною стандарту мови C++. Ця бібліотека шаблонів оперує поняттями чотирьох категорій.

1. *Контейнери* – шаблони класів, призначених для зберігання колекцій об'єктів. Прообразом такого контейнера в лекції був *template <typename ValueType> class List*.
2. *Ітератори* – сутності, що приховують структуру контейнера та надають уніфікований інтерфейс доступу до його елементів. Роль ітератора в лекції зіграв *List<ValueType>::Iter*.
3. *Алгоритми* – шаблони функцій, що розв'язують типові задачі опрацювання контейнерів. Для доступу до контейнерів алгоритми використовують ітератори. Написані нами шаблони *FindValInSuccession* та *FindAnyInSuccession* дуже на них схожі.
4. *Функтори* – сутності, що вміють опрацьовувати оператор круглі дужки, і слугують для налаштування алгоритмів на певні умови виконання. Лямбда-вирази, як в наших прикладах, цілком можуть бути такими функторами.

Глибокому вивченню STL будуть присвячені наступні лекції.