Лекція 7. Функції (продовження)

Функції і масиви (вказівник + розмір, діапазон, робота з частиною масиву).
Вказівники, посилання і специфікатор const.
Вказівник на функцію (typedef, using, два «імені» однієї функції).
Функції вищих порядків (табуляція, застосування до кожного, числові методи).
Використання лямбда-виразів.
Масиви функцій (текстове меню).

Приклад опрацювання масивів

У лекції 4 ми розглядали таку *задачу*: задано 10-ти елементні масиви a, b, c дійсних чисел. Обчисліть величину $u = \begin{cases} (a,c), \text{ якщо } (a,a) > 5, \\ (b,c) \text{ у іншому випадку.} \end{cases}$

Ми успішно написали для неї програму, проте виглядає вона як не дуже зрозуміле нагромадження циклів. Спробуємо прояснити ситуацію за допомогою функцій.

Проаналізуємо, які кроки потрібно виконати (і які ми вже зробили в попередній програмі), які з них є повторюваними, які оголошення стануть у пригоді.

Потрібно оголосити масиви, а для цього потрібно задати їхній розмір, і доцільно було б визначити синонім типу. Потрібно прочитати три масиви – отже оголосимо процедуру введення масиву. Потрібно рахувати скалярні добутки векторів. Цей алгоритм оформимо як окрему функцію. Для всіх оголошень і визначень використаємо окремі файли.

Функції для статичних масивів

Оголошення константи, синоніму типу та прототипи функцій розташовують у файлі заголовків.

файл ArrProcedures.h

```
#ifndef _Array_Procs_
#define _Array_Procs_

// Визначення типу потрібне для того, щоб зробити легкою зміну типу даних
using num_type = double;

const size_t N = 10;
using Array = num_type[N];

// Функції для роботи з масивами типу Array (зі статичними масивами)
// Поелементне введення масиву з консолі
void ReadArray(Array a);
// Обчислення скалярного добутку векторів
num_type ScalarProduct(Array a, Array b);

#endif
```

Тут тип *Array* фіксує і тип елементів масиву, і їхню кількість, тому в процедурі *ReadArray* достатньо одного параметра, щоб передати масив, елементи якого потрібно ввести з консолі. Ми вже знаємо, що за іменем масиву приховано вказівник на пам'ять його елементів. Таким чином параметр процедури *ReadArray* отримає вказівник на справжній масив і зможе змінити значення його елементів. Цього нам і потрібно.

файл ArrProcedures.cpp

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

```
void ReadArray(Array a)
{
	std::cout << "Input " << N << " values: ";
	for (size_t i = 0; i < N; ++i) std::cin >> a[i];
	std::cin.get();
}
num_type ScalarProduct(Array a, Array b)
{
	num_type s = 0;
	for (size_t i = 0; i < N; ++i) s += a[i] * b[i];
	return s;
}
```

Зверніть увагу на використання константи *N* у підпрограмах. Це ім'я – глобальне стосовно них, адже ми оголосили константу поза блоками функцій.

Подивіться, як тепер стисло і виразно виглядає головна програма:

файл Program.cpp

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

int main()
{
	using std::cin;
	using std::cout;
	// Розв'яжемо задачу за допомогою статичних масивів при n=10
	Array A, B, C;
	cout << " Vector a\n"; ReadArray(A);
	cout << " Vector b\n"; ReadArray(B);
	cout << " Vector c\n"; ReadArray(C);
	num_type u;
	if (ScalarProduct(A, A) > 5)
		u = ScalarProduct(A, C);
	else
		u = ScalarProduct(B, C);
	cout << "U = " << u << '\n';
	cin.get();
	return 0;
}
```

Функції для динамічних масивів

Попередня програма цілком добра, але має один недолік: вона працює виключно для 10-ти елементних масивів. Ми можемо легко змінити значення константи *N* на потрібне, але доведеться компілювати весь проект.

Щоб виправити цей недолік, використаємо знайомі нам динамічні масиви. Але як їх передавати у функцію? Динамічний масив задає дві величини: вказівник на його початок і розмір (кількість елементів). Отже, у функції потрібно використати два параметри. Нова версія процедури введення масиву матиме вигляд:

```
// Поелементне введення масиву з консолі
void ReadArray(num_type* a, size_t n)
{
	std::cout << "Input " << n << " values: ";
	for (size_t i = 0; i < n; ++i) std::cin >> a[i];
	std::cin.get();
}
```

Тут *a* – вказівник на початок масиву, *n* – кількість його елементів.

Цікаво, що така процедура може працювати і зі статичними, і з динамічними масивами.

Схожим чином зміниться і функція обчислення скалярного добутку:

```
// Обчислення скалярного добутку векторів
num_type ScalarProduct(const num_type* a, const num_type* b, size_t n)
{
	num_type s = 0;
	for (size_t i = 0; i < n; ++i) s += a[i] * b[i];
	return s;
}
```

Чи ви зауважили, що в оголошеннях параметрів *a* і *b* використано модифікатор *const*? Це зроблено не випадково. Коли ви передаєте у функцію вказівник на змінну, то функція отримує повну владу над аргументом і може змінювати його на власний розсуд. Таких дій ми сподівалися від *ReadArray*. Але чи обчислення скалярного добутку мало б змінити самі масиви? Звичайно, що ні! Тому в оголошенні функції використано додатковий засіб безпеки. Тепер масиви *a* і *b* можна тільки читати і заборонено змінювати. Мова C++ заохочує програмістів до використання модифікатора *const* усюди, де це має сенс. Зокрема, так захищають і параметри-посилання, якщо вони відіграють роль вхідних параметрів.

Продовжимо наші вправи з написання корисних функцій для роботи з масивами. Ми вже вміємо обчислювати скалярний добуток векторів, то чому б не визначити функцію обчислення середньоквадратичної норми. Функція створення динамічного масиву з ініціалізацією його елементів також не завадить.

```
// Обчислення норми вектора
num_type Norm(const num_type* a, size_t n)
{
	return sqrt(ScalarProduct(a, a, n));
}
// Створення динамічного масиву заданого розміру
num_type* CreateArray(size_t n, num_type x = 0)
{
	// резервуємо пам'ять для масиву
	num_type* a = new num_type[n];
	// надаємо елементам масиву початкових значень
	for (size_t i = 0; i < n; ++i) a[i] = x;
	// повертаємо готовий масив. Звільнити пам'ять від нього повинен користувач
	return a;
}
```

У функції *CreateArray* параметр *x* оголошено не так, як інші. Цей параметр має значення за замовчанням. Тепер його можна не вказувати при виклику функції. Якщо користувач хоче отримати масив, наповнений нулями, то у виклику достатньо вказати розмір масиву. Щоб отримати якесь інше наповнення, то функцію викликають з двома параметрами, наприклад, так *num_type * points = CreateArray(5, 100);* тоді параметр *x* отримає значення 100.

Параметри, що мають значення за замовчуванням, завжди розташовують наприкінці списку параметрів.

Оновлена головна програма тепер матиме такий вигляд:

```
int main()
{
	// А тепер використаємо динамічні масиви
	cout << "Input size of arrays: ";
	size_t n; cin >> n;

	// Створити динамічні масиви і ввести їх
	num_type* a = CreateArray(n);
	num_type* b = CreateArray(n);
	num_type* c = CreateArray(n);
	cout << " Vector a\n"; ReadArray(a, n);
	cout << " Vector b\n"; ReadArray(b, n);
	cout << " Vector c\n"; ReadArray(c, n);
```

```
        // Обчислення
        num_type U = (ScalarProduct(a, a, n) > 5) ?
              ScalarProduct(a, c, n) : ScalarProduct(b, c, n);
        cout << " U = " << U << '\n';
        cin.get();
        // Звільнення пам'яті
        delete[] a; delete[] b; delete[] c;
        return 0;
}
```

Передавання масиву у функцію

У попередньому параграфі ми розглянули два способи передавання масиву у функцію: випадок статичного масиву та універсальний випадок. Перший виглядає привабливіше, він коротший, та все ж є швидше екзотикою, ніж правилом. Зазвичай використовують другий (з вказівником і розміром), оскільки він підходить для масивів довільного розміру.

Покажемо, що таке передавання масиву можна використати дещо несподіваним способом. Розглянемо задачу, яку до кінця семестру повинні вміти розв'язувати усі, хто хоче залишитись серед числа студентів.

Задача. Напишіть функцію, що знаходить найбільше значення в масиві дійсних чисел. Відомо, що пошук треба починати з якогось елемента масиву, наприклад, з першого. А далі вибране значення порівнюють з усіма решту елементами масиву та заміняють на більше, якщо таке трапиться.

```
// Знаходить найбільше значення в масиві
double maxVal(const double* a, size_t n)
{
        double result = a[0];
        for (size_t i = 1; i < n; ++i)
              if (a[i] > result) result = a[i];
        return result;
}
```

Очікуване використання цієї функції:

```
// Експеримент з відшуканням максимального
double d[] = { 1, -2, 0, 5, 7, 3, 6, -4, 2, 5 };
size_t k = sizeof d / sizeof d[0];
cout << "max value is " << maxVal(d, k) << '\n';
```

Чи можна за допомогою цієї функції дати відповідь на запитання, до якої половини масиву – першої чи другої – належить максимальний елемент? Здавалося б, що ні. Адже функція нічого не повідомляє про індекс максимального елемента. Але не поспішайте з відповіддю. Проаналізуйте такий фрагмент коду:

```
if (maxVal(d, k / 2) > maxVal(&d[k / 2], k - k / 2))
        cout << "Max is situated at the first half of the array\n";
else
        cout << "Max is situated at the second half of the array\n";
```

Як тут працюють виклики *maxVal*?

У першому випадку *maxVal(d, k/2)* ми повідомляємо функції *менший* розмір масиву, ніж він є насправді. І в цьому нема ніякого криміналу. Функція не знає (і не може знати) нічого про наші масиви. Точніше, їй «байдуже»: які параметри їй передадуть, з такими вона і працюватиме. Помилка станеться, якщо передати у функцію *більший* ніж насправді розмір. Тоді функція добросовісно пройдеться по чужій пам'яті, вважаючи, що то переданий їй масив.

У другому випадку ми змінили трактування не тільки розміру масиву, а і його початку. Для функції він починається з елемента $d[k/2]$, і «з її точки зору» є звичайним масивом. До речі, замість оператора взяття адреси можна було використати вираз $d+k/2$ – пам'ятаєте з лекції про арифметику вказівників, що це означає?

Передавання вказівника і розміру – не єдиний спосіб передавання масиву у функцію. Ще один поширений спосіб використовує передавання пари вказівників *start*, *end*, які задають діапазон [*start*; *end*).

Функція відшукання найбільшого значення може виглядати і так:

```
// Знаходить найбільше значення в діапазоні
double maxVal(const double* start, const double* end)
{
	double result = *start++;
	while (start != end)
	{
		if (*start > result) result = *start;
		++start;
	}
	return result;
}
```

А її використання – так:

```
// Експеримент з відшуканням максимального
double d[] = { 1, -2, 0, 5, 7, 3, 6, -4, 2, 5 };
size_t k = sizeof d / sizeof d[0];
cout << "max value is " << maxVal(d, d + k) << '\n';
```

Обидва варіанти *maxVal* працюють однаково добре. Передавання діапазону притаманне алгоритмам стандартної бібліотеки C++.

Вказівники дають програмістові багато можливостей. Наприклад, ми могли б знайти не тільки найбільше *значення*, а і його *місце* в масиві. Подумайте, як працює наступний фрагмент коду:

```
// Знаходить місце найбільшого значення в діапазоні
double* max(double* start, double* end)
{
	double* pos = start;
	while (start != end)
	{
		if (*start > *pos) pos = start;
		++start;
	}
	return pos;
}
int main()
{	. . .
	double* ptr = max(d, d + k);
	cout << "max value is " << *ptr << " at index " << ptr - d << '\n';
```

Більше прикладів використання функцій і масивів можна знайти в матеріалі «Вкладені цикли в матричних задачах» та «Поєднання повторення з галуженням» теми «Методи розробки алгоритмів» у вашій Team на вкладці Файли.

Вказівник на функцію

Чи змогли б ви написати процедуру, яка друкує таблицю значень функції $y = \sin x$ на проміжку $[a; b]$ з заданим кроком h? Так, придумуємо ім'я для процедури, оголошуємо три параметри a, b, h, в її блоці організовуємо цикл.

```
// Побудова таблиці значень функції sin(x)
void Tabulate(double a, double b, double h)
{
	unsigned n = round((b - a) / h);
	// друк шапки таблиці
	std::cout << "\t x\t|\t sin(x)\n    ---------------------------------\n";
	// обчислення і друк тіла таблиці
```

```
        for (unsigned i = 0; i <= n; ++i)
        {
                double x = a + i*h;
                (std::cout << '\t').precision(4);
                (std::cout << x << "\t|\t").precision(8);
                std::cout << sin(x) << '\n';
        }
        std::cout << '\n';
}
```

Чи могли б ви так само протабулювати функцію $y = \sqrt[3]{x}$? Так, нова процедура відрізнятиметься від попередньої тільки способом обчислення *y*, а в іншому вона така ж.

```
// Побудова таблиці значень функції cbrt(x)
void Tabulate(double a, double b, double h)
{
        unsigned n = round((b - a) / h);
        // друк шапки таблиці
        std::cout << "\t x\t|\t cbrt(x)\n    --------------------------------\n";
        // обчислення і друк тіла таблиці
        for (unsigned i = 0; i <= n; ++i)
        {
                double x = a + i*h;
                (std::cout << '\t').precision(4);
                (std::cout << x << "\t|\t").precision(8);
                std::cout << cbrt(x) << '\n';
        }
        std::cout << '\n';
}
```

А чи могли б ви написати процедуру друкування таблиці значень *довільної* функції $f(x)$? Це запитання набагато цікавіше! Процедурі побудови таблиці потрібно якось передати інформацію про те, яку функцію табулювати. Параметрами мали б бути не тільки *a*, *b* і *h*, а й *f*. Тобто, функцію *f* треба розглядати не як алгоритм, а *як дані*. Здійснити задумане допоможе новий тип даних – вказівник на функцію.

Синтаксис

У попередній лекції ми оголошували функцію обчислення кубічного кореня дійсного числа. Її прототип мав вигляд

```
double cbrt(double x);
```

Круглі дужки після імені *cbrt* – це оператор оголошення функції. Щоб оголосити вказівник, потрібно використати оператор «*». Оскільки оператор «( )» має вищий пріоритет, потрібно взяти оголошення вказівника в дужки. Отримаємо:

```
double (*pF)(double x);
```

Тепер *pF* – це вказівник, який може зберігати адресу довільної функції, що приймає один параметр типу *double* і повертає результат типу *double*. Ми могли б написати:

```
pF = &cbrt;
```

Після цього *pF* стає «другим іменем» для функції *cbrt*: її тепер можна викликати за вказівником:

```
double y = (*pF)(-8.0); // y = -2.0
```

Для вказівників на функції компілятор підтримує спрощений синтаксис без вказання операторів «&» і «*», який переважно і використовують:

```
if (day == "Sunday") pF = sin;
else pF = cbrt;                 // взяття адреси можна не вказувати
```

```
// тепер наступний виклик залежить від дня тижня
double y = pF(-8.0);             // вказівник можна не розіменовувати
```

Зверніть увагу на різне значення наступних інструкцій:

```
y = cbrt(-8.0); // виклик функції, присвоєння результату
pF = cbrt;      // використання функції як сутності, присвоєння адреси
```

Повернемося до розгляду можливих типів вказівників на функцію. З оголошення *pF* видно, що він може зберігати вказівник на довільну функцію одного дійсного аргумента з дійсним же результатом. Вказівник можна брати справді на довільну функцію: і власну, і стандартну. Вказівник можна оголосити і для інших типів функцій: з іншим складом параметрів та іншим типом результату. Тут нема жодних обмежень.

На практиці для оголошення вказівників на функції зручно використовувати синоніми типу. Замість попереднього оголошення *pF* варто використовувати таке:

```
typedef double (*func)(double);
func pF = cbrt;
```

А ще краще, таке:

```
using func = double(*)(double);
func pF = cbrt;
```

Приклади інших типів:

```
typedef void(*proc)(void); // всі процедури без параметрів
// усі функції, що приймають два цілих і повертають логічне
using binary_predicate = bool(*)(int, int);
```

Вказівник на функцію як параметр функції

Самі по собі вказівники на функцію використовують рідко. Найчастіше вони бувають параметрами функцій: функцій, які використовують в роботі інші функції. Такі «великі» функції називають функціями вищих порядків. Отож, задумана нами процедура побудови таблиці значень довільної функції $f(x)$ матиме вигляд:

```
// Побудова таблиці значень функції
void Tabulate(func f, double a, double b, double h)
{
	unsigned n = round((b - a) / h);
	// друк шапки таблиці
	std::cout << "\t x\t|\t f(x)\n    ---------------------------------\n";
	// обчислення і друк тіла таблиці
	for (unsigned i = 0; i <= n; ++i)
	{
		double x = a + i*h;
		(std::cout << '\t').precision(4);
		(std::cout << x << "\t|\t").precision(8);
		std::cout << f(x) << '\n';
	}
	std::cout << '\n';
}
```

А її використання:

```
double parabola(double);

int main()
{
	// табулювання стандартної функції
	std::cout << "\t\t\tTable 1. Sin(x)\n";
	Tabulate(sin, 0.0, M_PI_2, M_PI_2 / 12);
```

```
        std::cin.get();
        // табулювання власної функції
        std::cout << "\t\t\tTable 2. x^2+2x-3\n";
        Tabulate(parabola, -4.0, 2.5, 0.25);
        std::cin.get();
        return 0;
}
double parabola(double x)
{
        return (x + 2.) * x - 3.;
}
```

Чи звернули ви увагу на те, що нам довелося оголосити функцію *parabola*, щоб отримати таблицю значень квадратного тричлена? Чи не єдине її призначення – стати аргументом у виклику *Tabulate*. Трохи громіздко, чи не так. До того ж численні оголошення таких дрібних «одноразових» функцій засмічують простір імен. Є економніший спосіб.

Вказівники на функції та функції вищих порядків прийшли у C++ з мов функціонального програмування. А слідом за ними примандрували й анонімні функції, які ще називають *лямбда-виразами*. Лямбда – це спосіб оголосити функцію в тому місці, де її потрібно викликати чи використати як параметр. Лямбда є функцією без імені, але з набором параметрів і блоком виконання. Зазвичай лямбда-виразом оголошують невеликі функції, потрібні для виконання функціям вищих порядків. Наприклад, якби ми хотіли надрукувати таблицю значень функції $y = \sqrt{1 - x^2}$, то виклик *Tabulate* мав би вигляд

```
// функцію задано лямбда-виразом
std::cout << "\t\t\tTable 3. Sqrt(1-x^2)\n";
Tabulate([](double x){return sqrt(1. - x*x); }, -1.0, 1.0, 0.1);
std::cin.get();
```

Оголошення лямбда виразу починається з квадратних дужок (ніби замість імені), далі – список параметрів і блок. Тип результату компілятор може вивести самостійно. Докладніше про синтаксис лямбда і приклади використання можна прочитати тут:

https://docs.microsoft.com/en-us/previous-versions/visualstudio/visual-studio-2010/dd293603(v=vs.100) чи тут https://ru.cppreference.com/w/cpp/language/lambda

Зокрема, вам би було цікаво дізнатися, що за допомогою [] лямбда вираз може захоплювати зовнішні стосовно себе імена з контексту виконання і змінювати їхні значення.

Додаткові приклади використання вказівників на функцію знайдете в матеріалі «Обчислення з заданою точністю» теми «Методи розробки алгоритмів» у вашій Teams.

Масиви функцій

Так, у назві нема помилок! За допомогою вказівників на функції можна вибудовувати масиви функцій. Такі структури забезпечують напрочуд гнучке планування алгоритму, дають змогу викликати функцію за індексом тощо.

Наведемо один приклад, у якому за допомогою масиву функцій моделюється робота простого текстового меню

```
using proc = void(*)(void);

// Процедури, що виконують окремі "розділи" великої програми
void CalculateRoots();
void Integrate();
void DeelWithArray();

int main()
{
        // Моделювання роботи меню
        proc menu[4] = { CalculateRoots, Integrate, DeelWithArray,
                []{ std::cout << "Bye!\n"; } }; // процедуру-прощання задано лямбда-виразом
```

```
	unsigned k = 0;
	do
	{
		std::cout << "Input your choice:\n0 - CalculateRoots\n"
			<< "1 - Integrate\n2 - DeelWithArray\n3 - Exit\n >> ";
		std::cin >> k; k %= 4;
		menu[k](); // виклик вибраного підрозділу меню
	} while (k != 3);
	system("pause");
	return 0;
}
```

Визначення процедур-розділів читач може вигадати самостійно. Зрозуміло, що їхні назви і кількість повністю залежать від потреб і фантазії автора програми.

Література

1. Бублик В.В. Об'єктно-орієнтоване програмування мовою С++.
2. Стивен Прата Язык программирования С++.
3. Дудзяний І.М. Програмування мовою С++.
4. Бьерн Страуструп Язык программирования С++.

Приклад функції для введення з консолі числового масиву. Користувач може ввести менше ніж *n* елементів масиву. Для того, щоб перервати введення достроково, достатньо ввести «неправильний» символ, наприклад '#', "quit", 'X' тощо. Поява нечислового символа призводить до виникнення помилки потоку введення, яка і зупиняє цикл *while*. (Зауважимо, що результатом виконання оператора *cin>>x* в логічних виразах є *true*, якщо дані прочитано без помилок, або *false* у протилежному випадку.) Тепер, щоб відновити працездатність потоку введення, скидаємо всі прапорці помилок методом *clear()* і забираємо з потоку всі непрочитані символи аж до кінця рядка. Як результат функція повертає кількість прочитаних елементів масиву.

```
// "Удосконалена" функція введення числового масиву
size_t readMas(double * a, size_t n)
{
	cout << "Input up to " << n << " numbers (not digit to break)\n";
	size_t loaded = 0;
	while (loaded < n && cin >> a[loaded]) ++loaded;
	if (loaded < n)
	{
		cin.clear();
		while (cin.get() != '\n') continue;
	}
	return loaded;
}
```