Лекція 6. Функції

Функції в С++: прототип, визначення, виклик.
Модульне програмування.
Способи передавання аргументів у функцію.
Параметри-значення.
Параметри-вказівники.
Тип посилання, використання в параметрах функцій.

Усе живе в природі розвивається, невпинно рухається від простого до складного. Такі процеси відбуваються і з мовами програмування. Адже користувачі потребують щораз досконаліших, зручніших програм. Розробники комп'ютерної техніки щоразу пропонують потужнішу техніку з ширшим набором можливостей. Ці два головні фактори зумовлюють невпинний розвиток технологій програмування. Сучасні програми настільки складні та великі, що першочергове значення має архітектура програми: її структура та зв'язки між складовими частинами цієї структури. «Зі зростанням складності структура домінує над матеріалами.» Цю тезу легко проілюструвати простим прикладом. Чи можна скласти одна на одну десять цеглин? Так, зробити це вдасться кожному, і цегляний стовпчик стоятиме надійно. Чи можна так само скласти 100–200 цеглин? Така робота вимагатиме чималих зусиль: доведеться добирати і припасовувати кожну цеглину, робити стовпчик якомога рівнішим, проте кожна цеглина може стати останньою: така висока складанка може завалитися навіть від подиху вітерця... Нічого не зробиш, діє філософський закон переходу кількості в якість. Щоб скласти таку кількість цегли доведеться вигадувати складніший спосіб вкладання, наприклад, складати два стовпчики та перев'язувати їх між собою.

Так і з програмуванням: щоб створити справді велику та складну програму не достатньо просто записати підряд багато-багато правильних операторів. Кожен новий оператор може «завалити» всю програму. Доведеться в першу чергу турбуватися про її надійну структуру.

Історично склалося так, що найпростішим, «первинним» засобом структурування програми є функції та процедури, або, одним словом, підпрограми. Значення цього винаходу для розвитку технології програмування часто порівнюють з винайденням арки в архітектурі. Арки свого часу спричинили революцію в будівництві, бо люди отримали змогу перекривати значні за розмірами пройми між опорами за допомогою звичайних невеликих цеглин, укладених спеціальним способом. Так і підпрограми незрівнянно розширили можливості як окремих програмістів, так і цілих колективів.

Процедурно-орієнтований підхід до написання програм підтримують різні мови, але при цьому однакові поняття називають дещо різними термінами, тому уточнимо термінологію. У мові С++ використовують функції з типом і без типу, їхні відповідники в мові Pascal – функції (function) і процедури (procedure), у мові FORTRAN – підпрограми-функції (function) і підпрограми (subroutine), мова Python, як і С++, все називає функціями. Надалі ми будемо вживати терміни *функція*, *процедура* та *підпрограма* як синоніми.

Функції мовою програмування С++

Отож, мовою С++ можна оголошувати функції двох видів:

Функція без типу (процедура)

```
void ім'я_функції(список_параметрів)  // заголовок
{                                     //------------
     інструкція | інструкції          //  блок
     [return;]                        //  функції
}                                     //-------------
```

Інструкція *return* не обов'язкова.

Функція, що повертає значення

```
тип_результату ім'я_функції(список_параметрів)  // заголовок
{                                               //------------
     інструкція | інструкції                    //  блок
     return вираз;                              //  функції
}                                               //-------------
```

Обов'язково хоча б одна інструкція *return* з виразом типу *тип_результату*. Типом результату може бути будь-який тип C++ крім масиву.

В обох випадках *список параметрів* може бути порожнім. Непорожній список містить оголошення, відокремлені комою. Правила побудови оголошення такі ж, як для звичайних змінних з одним уточненням: кожен параметр оголошують окремо. Інструкція *return* завершує роботу функції і повертає керування в точку виклику, таких інструкцій в блоці може бути декілька, якщо функція може завершити роботу за різних умов.

Заголовок описує інтерфейс (правила взаємодії з підпрограмою) і вказує, *що* вона робить. Блок описує, *як* вона це робить. Підпрограма працює за принципом «чорного ящика»: відомо, що він робить, і не відомо, як. Видимим для зовнішнього світу є тільки заголовок: він містить ім'я процедури та оголошення формальних параметрів. Ця інформація потрібна для того, щоб правильно викликати підпрограму, передати їй вхідні дані та отримати від неї результат. Блок повністю прихований від сторонніх: яке кому діло, як підпрограма робить те, що вона робить? Таке приховування є наслідком поділу праці, а значить, і відповідальності за її результати. Виробник гарантує високу ефективність і надійність його підпрограм, і у користувача з цього приводу може голова не боліти. Він може спокійно міркувати про те, як ці чудові підпрограми використати.

Наведемо короткий приклад використання функції. Для обчислення алгебричних виразів часто потрібні функції обчислення коренів. У стандартній бібліотеці *cmath* є функції *sqrt* та *cbrt* для обчислення квадратного та кубічного коренів відповідно. Корінь четвертого степеня з дійсного x можна обчислювати за допомогою $sqrt(sqrt(x))$ – не дуже гарно, але працює. Як обчислити корінь п'ятого степеня? Можна спробувати зробити це як піднесення до степеня 1/5: $pow(x, 1./5.)$, проте стандартна функція *pow* працює тільки для додатних x. То що ж робити? У реалізації функції *pow* використано відому тотожність: $a^x = e^{x \ln a}$, $a > 0$ і стандартні $exp(x)$ та $log(x)$. Ми теж можемо використати такий підхід, додатково проаналізувавши знак основи степеня: $x^{1/5} = \begin{cases} -e^{1/5\ln(-x)}, & x < 0 \\ 0, & x = 0; \\ e^{1/5\ln x}, & x > 0. \end{cases}$ Опишемо цю *формулу* за допомогою *функції*:

```
// обчислення кореня п'ятого степеня дійсного числа
double ptrt(double x)
{
	double res;
	if (x < 0)
		res = -exp(0.2 * log(-x));
	else if (x == 0)
		res = 0;
	else
		res = exp(0.2 * log(x));
	return res;
}
```

Із заголовка зрозуміло, що функція називається *ptrt* (подібно до *sqrt*), приймає один параметр дійсного типу і повертає результат обчислень дійсного типу. Так само, як і у формулі, в оголошенні функції не відоме конкретне значення x. Його задають, коли настає момент використання функції (чи формули).

Тепер *ptrt* можна використати в головній програмі:

```
int main()
{
	using std::cin;
	using std::cout;
	/* ЗАДАЧА. Задано дійсне x. Обчислити значення y = x^1/5 + (sin^2(x)-x)^1/5.
	   (тут x^1/5 означає корінь п'ятого степеня з x)*/
	cout << "Input x: ";
	double x;  (cin >> x).get();
	double y = ptrt(x) + ptrt(pow(sin(x), 2) - x);
	cout << " x = " << x << " y = " << y << '\n';
	cin.get();
	return 0;
}
```

Бачимо, що власну функцію можна використовувати так само, як і стандартні.

Процедури і функції мають однаковий сенс та аналогічну структуру, а відрізняються призначенням, способом використання та виглядом заголовка. Процедури описують виконання сукупності дій, що змінюють певні дані чи структури даних. Наприклад, введення значень змінних, виведення їх на екран чи запис до файла, перетворення матриці тощо. Процедуру можна трактувати як «мудрий оператор». До процедури звертаються за допомогою інструкції виклику процедури. У цьому операторі вказують ім'я процедури та фактичні параметри (аргументи). Пригадайте приклади використання стандартних функцій *system("pause")*, *SetConsoleOutputCP(1251)*.

Сенс функції полягає перш за все у тому, щоб описати алгоритм обчислення нового значення. Наприклад, числового, логічного чи вказівного типу. Тому функцію можна трактувати як «мудрий вираз» чи «мудрий операнд». Відповідно, виклик функції є одним з можливих операндів арифметичного (логічного, рядкового) виразу. Такий виклик позначає те значення, яке обчислює (частіше кажуть «повертає») функція.

У мові програмування С++ дозволено викликати функції, що повертають значення, як процедури – окремою інструкцією. Це буває зручно, коли функція крім обчислень виконує ще якусь роботу, а сам результат у певному випадку не важливий.

Що ж таке підпрограма?

Можна дати декілька відповідей:

1. це обмежена *частина коду* програми з власним іменем та набором параметрів; за іменем підпрограму можна викликати (і виконати) з різних місць головної програми, за допомогою параметрів її можна налаштувати на різні умови виконання в точці виклику;
2. це *засіб структурування* програми, поділу її на відносно незалежні частини (зв'язки всередині однієї підпрограми набагато суттєвіші, ніж між різними підпрограмами); програма, яка викликає підпрограму, є незалежною від її влаштування, зміни в блоці підпрограми не впливають на код головної програми;
3. це *засіб розподілу праці* програмістів: достатньо поділити велику задачу на частини, домовитись, яка підпрограма (з яким іменем і параметрами) вирішуватиме кожну з них, і далі можна створювати кожну з підпрограм незалежно від інших, це можуть робити різні люди;
4. це *засіб пришвидшення компіляції*: модульна програма компілюється швидше, оскільки повторного опрацювання потребують тільки ті підпрограми, які зазнали змін; для великих проектів пришвидшення компіляції сягає десятків разів;
5. це *засіб повторного використання коду* (і пришвидшення написання програм): наявність набору стандартних підпрограм полегшує працю програміста, бо йому не доводиться знову і знову кодувати обчислення, наприклад, функції *sin x* чи операторів уведення; нерідко підпрограми використовують у програмах, написаних іншою мовою програмування.

Сучасний підхід до програмування заохочує оформлення у вигляді підпрограми кожної відносно самостійної частини алгоритму, навіть і тоді, коли в головній програмі її буде використано тільки один раз. У цьому є резон: підпрограму легше написати і відтестувати, ніж всю програму; програма вийде краще структурованою; підпрограму можна буде використати в майбутньому.

Для того, щоб ***використати функцію в програмі*** мовою C++, потрібно виконати три кроки:

1. Оголосити прототип функції – він описує заголовок функції, а, отже, спосіб використання, чи інтерфейс функції.
2. Визначити функцію: навести повне визначення функції з заголовком, який відповідає прототипові, і блоком, який визначає, що саме робить функція.
3. Викликати функцію. При цьому потрібно вказати правильну кількість аргументів, типи яких відповідають типам параметрів функції.

Невеличкі програми можуть містити все в одному файлі. Для наведеного раніше прикладу обчислення кубічного кореня структура оголошень могла б бути такою:

файл Source.cpp

```
#include <iostream>
double ptrt(double); // прототип функції

int main()           // головна програма
{      . . .
       double y = ptrt(x) + ptrt(pow(sin(x), 2) - x);
       . . .
       return 0;
}
double ptrt(double x) // визначення функції
{
       double res;
       . . .
       return res;
}
```

У «справжніх» програмах визначення функцій розташовують у окремому файлі, адже модульність програм мовою C++ ґрунтується на зберіганні коду в декількох файлах, які можна компілювати незалежно. При цьому для зберігання прототипів використовують заголовкові файли – файли з розширенням імені *.h* – від англійського *header*.

Тепер наш приклад матиме таку структуру:

файл Function.h

```
#pragma once

// обчислення кореня п'ятого степеня дійсного числа
double ptrt(double);
```

файл Function.cpp

```
#include <cmath>
#include "Functions.h"

// обчислення кореня п'ятого степеня дійсного числа
double ptrt(double x)
{
       double res;
       if (x > 0) res = exp(0.2 * log(x));
       else if (x == 0) res = 0;
       else res = -exp(0.2 * log(-x));
       return res;
}
```

файл Program.cpp

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

int main()
{
	using std::cin;	using std::cout;
	cout << "Input x: ";
	double x;  (cin >> x).get();
	double y = ptrt(x) + ptrt(pow(sin(x), 2) - x);
	cout << " x = " << x << " y = " << y << '\n';
	cin.get();
	return 0;
}
```

Директива *#include "Functions.h"* вставляє на початок файла головної програми текст заголовкового файла. Так компілятор отримує інформацію про прототип функції *ptrt* і може компілювати програму. Така ж директива є у файлі з визначенням функції. Так компілятор зможе перевірити, чи відповідає визначення заявленому прототипові.

Директива *#pragma once* у файлі заголовків потрібна для того, щоб цей файл не потрапив до котрогось *cpp*-файла двічі-тричі. (Це можливо у великих проектах з багатьма файлами та директивами *include*.) Замість *pragma* можна використовувати «сторожа включення» – спеціальне символьне ім'я та директиви умовного опрацювання:

файл Function.h

```
#ifndef _FUNCTION_QUARD
#define _FUNCTION_QUARD

// обчислення кореня п'ятого степеня дійсного числа
double ptrt(double);

#endif
```

Серед фахівців нема єдиної думки, який спосіб кращий. Сучасні середовища програмування пропонують застосовувати перший.

Зверніть увагу на використання *include* в головній програмі:

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

Імена стандартних файлів заголовків вказують у кутніх дужках, а власних – у лапках.

Незалежність коду

Виділення коду в окрему функцію робить його незалежним від головної програми, а також робить незалежною програму від коду функції. Ви можете змінювати блок функції без шкоди для головної програми. Вона навіть «не відчує» таких змін, поки прототип функції залишатиметься незмінним.

Оголошену раніше функцію обчислення кубічного кореня ми могли б переписати без використання додаткових внутрішніх змінних:

```
// обчислення кореня п'ятого степеня дійсного числа
double ptrt(double x)
{
	if (x < 0) return -exp(0.2 * log(-x));
	else if (x > 0) return exp(0.2 * log(x));
	else return 0.0;
}
```

Ми не змінили заголовка, тому користувачам функції буде важко здогадатися про зроблені зміни. Хіба що хтось помітить економніше використання пам'яті.

Зверніть увагу на багаторазове використання інструкції *return*. Функція завершує роботу, як тільки визначиться зі способом обчислення кореня.

Тест блока можна ще скоротити, якщо забрати з нього обидва *else*. Подумайте, чому таке вилучення не вплине на алгоритм виконання функції, та чи варто таке робити з огляду на читабельність тексту програми. А як переписати блок функції за допомогою тернарного оператора?

Ми могли б продовжити експерименти з текстом функції. Можливо, нам варто обійтися без загадкових *exp* і *log*? Допоможе в цьому алгоритм Ньютона обчислення коренів:

```
double ptrt(double x)
{	double y = 1.0; double z = 0.0;
	while (abs(y - z) > 1.e-10)
	{	z = y;
		double p = z * z;
		y = (x / (p * p) + 4.*z) / 5.;
	}
	return y;
}
```

Навіть такі радикальні зміни ніяк не вплинуть на головну програму! Хіба що на швидкодію програми. На комп'ютерах без математичного співпроцесора така функція працювала б у рази швидше – але тепер таких уже не роблять. Як воно буде на сучасних машинах можна перевірити експериментально.

Дисципліна доступу до імен

У мові програмування C++ діє блокова дисципліна доступу до імен, згідно з якою областю видимості будь-якого імені є той блок, де воно оголошене: від точки оголошення до кінця блоку. Тому усі змінні, оголошені в підпрограмі, доступні виключно в її межах. Такі змінні називають *локальними*. Підпрограма може містити оголошення локальних змінних в заголовку (формальні параметри) та в самому блоці (внутрішні робочі змінні). Ім'я доступне всередині блоку включно з вкладеними в нього «дрібнішими» блоками.

Імена, оголошені поза блоками функцій, є *глобальними*. Вони доступні в межах файла. Глобальними є імена всіх функцій. Глобальними будуть імена змінних, типів, констант, якщо оголосити їх поза межами функцій. Якщо зустрінуться однакові глобальне і локальне імена, то видимим у блоці буде локальне. За блокової дисципліни доступу діє правило екранування: локальне ім'я закриває собою таке саме глобальне, і, якщо локальне та глобальне імена співпадають, то в межах блоку діє локальне ім'я.

У попередньому прикладі ім'я змінної *x* головної програми співпало з іменем формального параметра функції *ptrt*. При цьому жодного «конфлікту доступу до імен» не буде. Ми маємо справу з *різними* змінними. Це дуже добре, бо автор будь-якої підпрограми може не турбуватися про використання імен у інших підпрограмах: вже готових, чи ще не написаних. Локальний контекст імен ізолює його від можливих збігів.

Різниця між глобальними та локальними змінними є не тільки теоретичною – фізично вони розташовуються в різних частинах пам'яті: глобальні – в статичній пам'яті програми, локальні – в стеку, в частині, відведеній для блока функції.

Механізм параметрів

Що таке формальний параметр? Чим він відрізняється від фактичного? Навіщо вони обидва потрібні? Все це дуже важливі питання, які заслуговують якнайпильнішої Вашої уваги. Спробуємо дати на них зрозумілі відповіді.

Як ми обчислюємо абсолютну величину даного дійсного числа? Дуже просто: якщо число додатне чи нуль, то воно ж і є шукана величина, а якщо від'ємне, то треба відкинути знак «мінус». Це загальний опис алгоритму обчислення. Він не залежить від конкретного заданого значення, а слово «число» ніби резервує для нього місце в алгоритмі. Означення модуля допомогою формул можна описати так: $|x| = \begin{cases} x, & x \ge 0 \\ -x, & x < 0 \end{cases}$. Тут змінна *x* відіграє таку ж

роль: передбачається, що замість неї підставлятимемо якісь числа. Аналогію можна продовжити. В тексті оголошеної раніше функції *ptrt* ми використали ім'я *x*, щоб описати алгоритм обчислення кубічного кореня довільного числа. Змінна *x* у ньому позначає місця, куди буде підставлено конкретне значення після звертання до *ptrt*.

Ми уже говорили, що кожна підпрограма описує певний алгоритм, причому, для загального випадку. Щоб виконати алгоритм здебільшого потрібні конкретні вхідні дані, наприклад, кут для обчислення тангенса, чи послідовність чисел для відшукання найбільшого. Підпрограма мала б якимось чином отримувати ці дані, а також якимось чином повертати головній програмі результат обчислень. Саме для цього і використовують параметри.

Проектуючи процедуру, програміст аналізує, які дані є вхідними для алгоритму, які потрібні робочі змінні, що є результатом. Робочі змінні оголошують у блоці процедури за потребою. Назовні вони невидимі: використання робочих змінних – внутрішня справа самої процедури. Вхідні дані в момент написання процедури ще не відомі, тому замість них можна використати хіба що позначки – змінні відповідних до потреб алгоритму типів. Такі змінні і називають *формальними параметрами* процедури. В момент виклику їм потрібно надати конкретних значень, тому оголошення формальних параметрів розташовують у заголовку процедури (щоб вони були видимі ззовні). До формальних параметрів відносять також змінні, які позначають результати роботи підпрограми і передають їх назовні.

Фактичні параметри – це конкретні значення, що підставляються у підпрограму замість формальних параметрів у момент виклику. Фактичним параметром може бути число, вираз, змінна головної програми. Між формальними та фактичними параметрами має бути строга відповідність за кількістю та типом, відповідно до їх місця в списку параметрів.

Мусимо з'ясувати, як саме встановлюється зв'язок між параметрами і відповідними аргументами. У програмуванні відомо два способи передавання аргументів: *за значенням* та *за адресою*. **Передавання за значенням** використовують для того, щоб передати інформацію всередину функції. Так оголошують вхідні параметри. Відповідними їм фактичними параметрами можуть бути і змінні, і константи, і вирази. Це зручно для задання вхідних даних. Тип фактичного параметра сумісний за присвоєнням з типом відповідного формального. **Передавання за адресою** використовують для того, щоб передати інформацію назовні з функції. Це – результати. Відповідними їм фактичними параметрами можуть бути виключно змінні відповідного типу. Такий спосіб використовують також для передавання між програмою і функціями структур даних: масивів, структур, рядків – як вхідних, так і результатів.

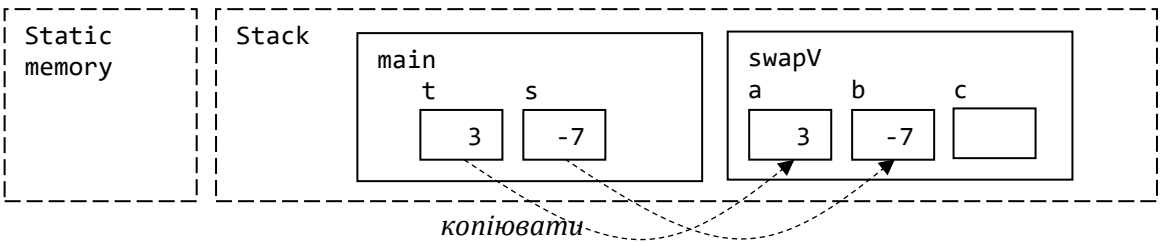
У мові програмування C++ зазвичай діє передавання параметрів за значенням. Для цього використовують оголошення параметрів простих типів. Щоб передати параметри за адресою, можна використати два різні способи: явне передавання адрес за допомогою вказівників (pointers), або приховане за допомогою посилань (references). Розглянемо усі.

```
// обмін місцями двох значень різними способами:
// параметри-значення
void swapV(int a, int b)
{
      int c = a;
      a = b;
      b = c;
}

int main()
{     /* Тепер випробуємо різні способи передавання параметрів*/
      int t = 3;   int s = -7;
      cout << "\nMain program pass t = " << t << " and s = " << s << " by value\n";
      swapV(t, s);
      cout << "===== after swap: t = " << t << " and s = " << s << '\n';
      cin.get();
      return 0;
}
```

Отримані результати:

```
Main program pass t = 3 and s = -7 by value
===== after swap: t = 3 and s = -7
```



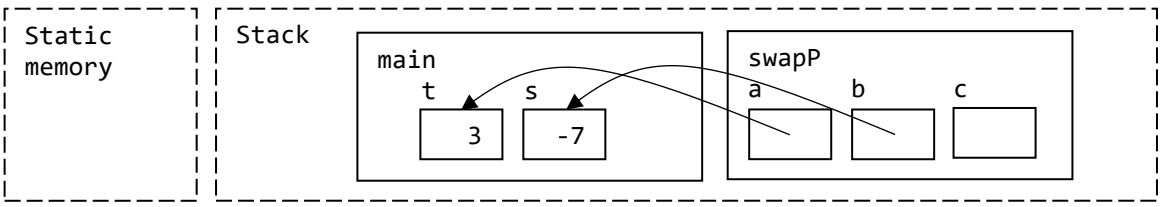
Бачимо, що значення *t* і *s* не змінилися. Чому? Параметри-значення – це вхідні параметри. У момент виклику відбулося «об'єднання» оголошення параметрів і значень аргументів: *int a=t; int b=s;*, після чого зв'язок між параметрами і аргументами розірвався (схематично зображено на рисунку). Було створено дві локальні змінні, ініціалізовані значеннями аргументів. Параметри передали значення 3 і –7 всередину *swapV*. У результаті виконання блока функції локальні змінні *a* та *b* обмінялися значеннями, проте цього ніяк не видно ззовні.

```
// параметри-вказівники
void swapP(int* a, int* b)
{
	int c = *a;
	*a = *b;
	*b = c;
}

int main()
{
	/* Тепер випробуємо різні способи передавання параметрів*/
	int t = 3;   int s = -7;
	cout << "\nMain program pass t = " << t << " and s = " << s << " by pointer\n";
	swapP(&t, &s);
	cout << "===== after swap: t = " << t << " and s = " << s << '\n';
	cin.get();
	return 0;
}
```

Отримані результати:

```
Main program pass t = 3 and s = -7 by pointer
===== after swap: t = -7 and s = 3
```



Аргументом параметра-вказівника має бути адреса змінної того типу, на який вказує вказівник. Бачимо, що в інструкції виклику функції *swapP* ми використали оператори взяття адреси для змінних *t* і *s*. Така форма явно підказує, що ми передаємо адреси, і значення *t* і *s* мали б змінитися.

У момент виклику відбулося «об'єднання» оголошення параметрів і значень аргументів: *int* a=&t; int* b=&s;*. Новостворені локальні змінні містять адреси змінних головної програми (див. рис. угорі), і з блока функції явно видно, що всі маніпуляції відбуваються саме з ними. Отримані результати засвідчують, що обмін відбувся!

Отже, параметри-вказівники можна використати для отримання результату з функції.

Ще один спосіб передавання параметрів за адресою використовує новий для нас тип – тип-посилання. Посилання – це спосіб створення в програмі «двійників» для певних змінних, або спосіб надання змінній додаткового імені.

Загальна схема визначення посилання:

```
<тип_змінної>& <нове_ім'я> = <змінна>;
```

Ініціалізатор в такому оголошенні – обов'язковий!

Наведемо приклад:

```
int var = 5;    // «оригінальна» змінна
int& ref = var; // двійник змінної
```

Після такого оголошення *ref* стає новим іменем для *var*. Усе, що ми робитимемо з *ref*, ставатиметься з *var*, і навпаки. Навіть оператор взяття адреси для обох змінних повертає однаковий результат – їх неможливо відрізнити. Це трохи дивно, бо в основі посилань все-таки вказівники. В момент створення *ref* компілятор робить її значенням незмінний вказівник на *var* так, ніби оголошено

```
int*const ref = &var;
```

Надалі при кожному звертанні до *ref* компілятор автоматично розіменовує цей вказівник і всі дії, записані для *ref*, відбуваються з *var*.

Для чого оголошувати в програмі посилання? Нема потреби робити це в межах одного блоку, як у прикладі вище. В програмах посилання використовують для передавання за адресою параметрів у функції.

```
// параметри-посилання
void swapR(int& a, int& b)
{
    int c = a;
    a = b;
    b = c;
}
int main()
{
    /* Тепер випробуємо різні способи передавання параметрів*/
    int t = 3;   int s = -7;
    cout << "\nMain program pass t = "<< t <<" and s = "<< s <<" by reference\n";
    swapR(t, s);
    cout << "===== after swap: t = " << t << " and s = " << s << '\n';
    cin.get();
    return 0;
}
```

Отримані результати:

```
Main program pass t = 3 and s = -7 by reference
===== after swap: t = -7 and s = 3
```

Аргументом параметра-посилання має бути змінна того типу, на який буде посилання.

У момент виклику відбулося «об'єднання» оголошення параметрів і значень аргументів: *int& a=t; int& b=s;*. Новостворені локальні змінні стають двійниками змінних головної програми, а «за лаштунками» містять їхні адреси (як на рисунку з вказівниками).

Параметри-посилання також повертають результат роботи функції! І працювати з ними зручніше, ніж з вказівниками, оскільки не потрібно морочитися з взяттям адреси і розіменуванням. Проте, є одне «але»: погляньте на виклик *swapR* – він нічим не відрізняється від виклику *swapV*, яка не змінювала своїх параметрів. Це певний недолік читабельності коду. Для повного розуміння ситуації потрібно бачити не тільки виклик, а й прототип функції. Але посилання такі зручні в використанні, що зручність перемогла недоліки, і застосовують їх повсякчас, особливо, для передавання структур і об'єктів.

Приклад використання параметрів функції

На завершення лекції розглянемо ще один приклад написання функції, який допоможе нам краще зрозуміти способи використання параметрів-значень і параметрів-посилань.

Усі ми пам'ятаємо зі школи алгоритм розв'язування квадратного рівняння за допомогою дискримінанта:

$$ax^2 + bx + c = 0,\ a \neq 0;\ \ D = b^2 - 4ac,\ x_{1,2} = \frac{-b \pm \sqrt{D}}{2a},\ \text{якщо } D \geq 0.$$

Давайте втілимо його в процедуру! Вхідними параметрами такої процедури мали б бути коефіцієнти рівняння. Для обчислених коренів можемо використати ще два параметри-посилання – не параметри-значення, а параметри-посилання, щоб отримати результат з процедури. Але як довідатися, чи дійсні корені рівняння існують, чи були вони обчислені? Використаємо ще один параметр-результат, який міститиме ознаку того, чи обчислення завершилися успіхом.

```
void solve(double a, double b, double c, bool& success, double& x1, double& x2)
{
	if (a == 0.0)
	{
		success = false;
		return;
	}
	double D = b*b - 4.0*a*c;
	if (D >= 0.0)
	{
		D = sqrt(D);
		b = -b;
		a *= 2.0;
		x1 = (b - D) / a;
		x2 = (b + D) / a;
		success = true;
	}
	else
		success = false;
}
```

Використання такої процедури могло б бути таким:

```
	// Розв'язування квадратного рівняння x^2 + 18x - 63 = 0
	double x1, x2;
	bool success;
	solve(1., 18., -63., success, x1, x2);
	if (success)
		cout << "x1 = " << x1 << "   x2 = " << x2 << '\n';
	else
		cout << "Roots could not be found\n";
```

Для отримання результату довелося оголосити три змінні, а у виклику вказати шість параметрів. Трохи забагато. Чи можна зменшити цю кількість? Так, якщо використати для отримання результатів обчислень дещо особливе.

Результатами є три величини: два дійсних числа та одне логічне значення. Мова C++ дозволяє об'єднати їх в одну сутність. Для цього ми оголосимо новий тип – структуру, що містить поля даних відповідного типу:

```
// розв'язки квадратного рівняння
struct sqRoots
{	bool success;
	double x1, x2;
	sqRoots(bool s, double u = 0.0, double v = 0.0) // конструктор структури
	{	success = s;  x1 = u; x2 = v;
	}
};
```

Тепер функція обчислення коренів квадратного рівняння матиме вигляд:

```
sqRoots Solve(double a, double b, double c)
{
	if (a == 0.0)
		return sqRoots(false);
	double D = b*b - 4.0*a*c;
	if (D >= 0.0)
	{
		D = sqrt(D);
		b = -b;
		a *= 2.0;
		return sqRoots(true, (b - D) / a, (b + D) / a);
	}
	else
		return sqRoots(false);
}
```

А її використання:

```
sqRoots result = Solve(1., 18., -63.);
if (result.success)
	cout << "x1 = " << result.x1 << "   x2 = " << result.x2 << '\n';
else
	cout << "Roots could not be found\n";
```

Докладно про оголошення і використання структур говоритимемо згодом.

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++.