

ФУНКЦІЙ. ПРОТОТИПИ ФУНКЦІЙ. ПАРАМЕТРИ ТА ЇХ ВИДИ. РЕКУРСІЯ

Модульне програмування

Відповідно до методології *структурного програмування*, будь-яку програму можна створити, використовуючи лише три конструкції (базові алгоритмічні структури):

- *послідовне виконання* – виконання команд у тому порядку, в якому вони записані у тексті програми;
- *розгалуження* – виконання певної команди чи кількох команд залежно від результатів перевірки певної конкретної умови;
- *цикл* – багатократне виконання команди чи групи команд за умови виконання певної конкретної умови.

Переваги структурного програмування:

- ✓ такі програми легко створювати та тестувати;
- ✓ скорочується термін розробки програми;
- ✓ полегшується супровід програми;
- ✓ скорочується кількість способів побудови програми, що знижує складність програмного забезпечення;
- ✓ логічно пов'язані операції знаходяться ближче одна до одної, що полегшує аналіз алгоритму, даючи змогу обходитись без блок-схем алгоритму (хоча наявність такої блок-схеми полегшує розуміння роботи алгоритму).

Недоліки структурного програмування:

- непропорційне зростання складності програми у разі збільшення об'єму її коду;
- складність створення паралельних програм

Модульне (процедурно-орієнтоване) програмування – це методологія й технологія розробки програмних комплексів, заснована на наступних принципах:

- увесь проект має бути розбитий на модулі з одним входом і одним виходом;
- логіка алгоритму та програми має допускати лише три основні структури: послідовне виконання, розгалуження та повторення. Неприпустимий оператор передачі керування в будь-яке місце програми;
- при розробці документація має створюватися одночасно із програмуванням, у вигляді коментарів до програми.

Ціллю модульного програмування є підвищення надійності програм, прискорення і полегшення їх створення. У програмах з використанням модульного

програмування добре простежується основний алгоритм, вони більш зручні в налагодженні і менш чутливі до помилок програмування. Це забезпечується за допомогою підпрограм, кожна з яких є в деякій мірі самостійним фрагментом програми. *Підпрограми пов'язуються з основною програмою кількома параметрами.* Самостійність підпрограм дає можливість локалізувати в них усі деталі програмної реалізації таким чином, що зміна цих деталей не приводить до змін у основній програмі.

Програма розділяється на дрібніші одиниці – *процедури і функції* (окремі ділянки коду, які виконують певні дії, задані алгоритмом та мають власну назву). Ці частини програми можуть викликатися з будь-якого місця у ній.

Поняття функції

Функція – незалежна частину програми, яка має власне ім'я і може викликатися з інших частин програми, оперувати даними та повертати результат. Терміном “функція” ще називають метод, підпрограму, процедуру тощо.

За необхідності функція може також повертати певне значення як результат. Функції корисні для групи команд в єдиному блоці, який можна використовувати багато разів. Рекомендується, щоб ім'я функції чітко описувало її призначення.

Кожна C++-програма має принаймні одну функцію – функцію *main()*. Крім неї, програма може містити необмежену кількість функцій, створених користувачем.

Існує два основні види функцій:

- *стандартні* (вбудовані)
- *функції користувача.*

Функції можуть запускатися (викликатися) в коді програми будь-яку кількість разів. Значення, які передаються функції, називаються її аргументами. Їх типи мають відповідати типам параметрів у заголовку функції.

Будь-яка функція повинна бути оголошена і визначена. Оголошення функції повідомляє компілятор як її викликати, а визначення – описує її тіло.

Оголошення функції описується за допомогою *прототипу* і має бути здійснене перед її викликом. *Прототип функції* не містить інформації про її внутрішню будову, він описує лише як до неї звертатись. Для виклику функції транслятору достатньо знати лише її ім'я, тип результату, кількість і типи аргументів.

Прототип функції має такий формат:

```
<тип_результату> <ім'я_функції> ([<список_параметрів>]);
```

Список параметрів описується так:

```
<тип_параметра_1> [<параметр_1>]
```

```
[, <тип_параметра_2> [<параметр_2>]
...
[, <тип_параметра_N> [<параметр_N>]
```

Наприклад

```
int sum(int a, int b); // лише тип, назва та параметри
//немає реалізації – тіла функції
```

Визначення функції має наступний формат

```
<тип_результату> <ім'я_функції> ([<список_параметрів>])
{
<тіло функції>
return <результат> //для всіх окрім void
}
```

Наприклад

```
int sum(int a, int b)
{
    return a + b; //тип результату такий самий, як і тип функції
}
```

Виконання функції починається тоді, коли в тексті програми зустрічається оператор виклику цієї функції. Функції можуть запускатися (викликатися) в коді програми будь-яку кількість разів. Значення, які передаються функції, називаються її *аргументами*. Їх типи мають відповідати типам параметрів у заголовку функції.

Виклик функції завжди позначається ім'ям функції та круглими дужками, в яких стоять змінні (константи, вирази), значення яких передаються (підставляються) замість аргументів.

Наприклад

```
int main()
{
    int i = sum(10, 7);
    cout << "The value - " << i << endl; // 17
}
```

У випадку, коли функція не має аргументів, круглі дужки все одно є обов'язковими.

Функція main() має містити лише виклики функцій, які виконуватимуть усю роботу програми. Кожна функція має вирішувати лише одну конкретну задачу, ім'я функції має бути змістовним і пояснювати призначення функції (зміст цієї задачі). Для максимальної ефективності коду доцільно використовувати функції, кожна з яких виконує якусь одну, чітко визначену задачу. Складні алгоритми, якщо це можливо, краще розбити на більш короткі та прості для розуміння функції. Якщо важко вибрати ім'я функції, – це означає, що вона вирішує багато задач, тому

потрібно цю функцію розділити на кілька простіших, кожна з яких вирішуватиме лише одну свою задачу.

Аргументи функції

Значення змінних, які передаються у функцію при виклику, називаються *фактичними параметрами* або *аргументами*, а змінні, оголошені у функції, що приймають значення аргументів, є локальними змінними функції, які створюються при вході в неї та знищуються при виході з неї.

При виклику функції локальні змінні послідовно отримують значення фактичних параметрів: значення першого фактичного аргументу копіюється у першу змінну, другий фактичний – у другу і т.д. При цьому перевіряється відповідність типів і, за потреби, виконуються їхні перетворення. Тому дуже важливими є кількість, порядок запису та відповідність типів усіх аргументів, інакше можуть виникнути помилки.

Функція може повертати або не повертати результат.

Якщо вона не повинна повертати жодного значення, як тип її результату потрібно вказувати тип `void`. В іншому випадку типом результату функції є певний тип даних.

Типи параметрів та тип результату функції – те ж саме, що і тип даних.

Розглянемо наступні прототипи функцій:

- Функція не повертає значення і не містить аргументів:

```
void test1(void);    // еквівалентно    void test1();
```

- Функція не повертає значення, але містить два аргументи типу `int` та `double`:

```
void test2(int, double);
```

- Функція повертає значення типу `int` і не містить аргументів:

```
int test3(); // еквівалентно int test3(void);
```

- Функція повертає значення типу `double` і містить два аргументи типу `int`:

```
double test4(int, int);
```

Функції виконуються в порядку їх виклику, а не в порядку їх запису оголошення чи визначення. Добрим стилем програмування вважається оголошувати функції на початку програмного коду перед визначенням функції `main()`, а визначати їх після неї.

Наступна функція виводить на екран повідомлення і не повертає жодного результату:

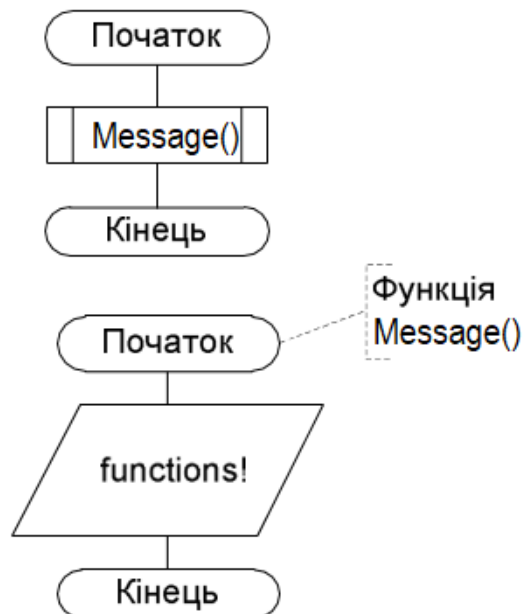
```
#include <iomanip>
#include <iostream>

using namespace std;

// оголошення прототипу функції
void Message();
```

```
// головна функція
void main() {
// виклик функції
    Message();

}
// визначення функції Show()
void Message() {
    cout << "Functions!";
}
```



Завершення виконання та повернення результату функції

Команда **return** завершує виконання функції і повертає управління в місце виклику. *Виконання функції* завершується після виконання команди **return** або при досягненні фігурної дужки **}**, яка закриває блок функції.

Якщо функція повертає результат, то тип результату, вказаний перед іменем функції, має відповідати типу виразу, вказаного після ключового слова **return**.

Значення виразу, записаного в команді **return** ; – буде результатом функції.

Якщо функція не повертає результату, то замість типу результату в заголовку функції слід вказати ключове слово **void**. Команда **return** в цьому випадку – необов'язкова.

За умовчанням, якщо тип результату функції явно не вказаний, в мові C/C++ він вважається **int**. Якщо функція немає параметрів, то все рівно після імені функції слід вказувати порожні круглі дужки або слово **void** в круглих дужках

```
int main()
{
    ...
    return 0;
}
```

```

або
int main(void)
{
    ...
    return 0;
}
    
```

Команда виклику **void**-функції не може увійти до складу виразу, бо **void**-функція не повертає значення – це має бути окрема команда (окремий оператор).

Наприклад, команда виклику **void**-функції, визначеної так:

```

void printSquare(int n) // визначення функції
{
    cout << n * n << endl;
}
    
```

виглядає так:

```

printSquare(5); // виклик функції в окремій команді
    
```

Команда виклику функції з конкретним типом результату (**int**-функції), визначеної так:

```

int square(int n) // визначення функції
{
    return n * n;
}
    
```

може бути наступною:

```

cout << square(5) << endl; // виклик функції в команді виведення а
    
```

або такою:

```

int k = square(2); // виклик функції в команді присвоєння
    
```

Цей варіант виклику:

```

square(2); // виклик функції в окремій команді
    
```

приводить до виконання всіх дій, записаних в блоці функції. При цьому значення, яке обчислює функція, – ніде не використовується.

Нижче наведено приклад реалізації та виклику функції для обчислення індексу маси тіла людини:

```

#include <iomanip>
#include <iostream>

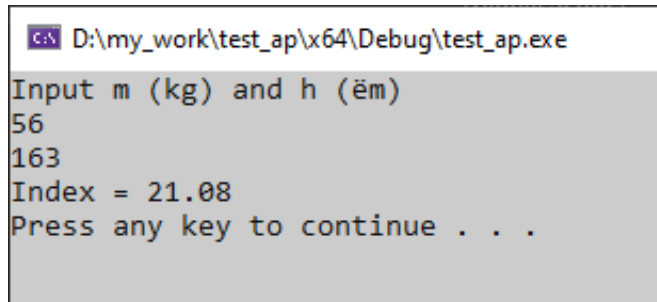
using namespace std;

// оголошення прототипу
double index_m(double, int);
//головна функція
int main() {
    double m;
    int h ;// локальні змінні для функції main()
    cout << "Input m (kg) and h (cm)" << endl;
    cin >> m;
}
    
```

```

cin >> h;
cout << fixed << setprecision(2);
cout << "Index = " << index_m(m, h)<< endl;
system("pause");
return 0;
}
// визначення функції middle()
double index_m(double a, int b) {
double index; // a, b, index – локальні змінні для функції
index = (a / (b*b))*10000;
return index; // повернення значення змінної
}
    
```

Результат виконання програми



```

D:\my_work\test_ap\x64\Debug\test_ap.exe
Input m (kg) and h (m)
56
163
Index = 21.08
Press any key to continue . . .
    
```

Якщо значенням оператора повернення є вираз, то спочатку обчислюється значення виразу, а тоді воно передається у місце виклику функції у програмі.

Оператор `return` переважно є останнім оператором тіла функції, але він може бути записаний у будь-якому місці тіла функції.

Якщо проект містить багато файлів (програма складається із багатьох модулів), то прототипи функцій зазвичай розташовують у заголовних файлах (з розширенням `.h`), а повні описи цих функцій – у файлах реалізації (з розширенням `.cpp`).

Особливості використання функцій

Є два способи, якими можна передати інформацію у функцію за допомогою параметрів:

- 1) передавати значення аргументів;
- 2) передавати адреси аргументів.

В мові C++ інформацію у функцію можна передавати за допомогою параметрів, які поділяються на наступні три види:

- параметри-значення ;
- параметри-вказівники ;
- параметри-посилання .

Параметри-значення

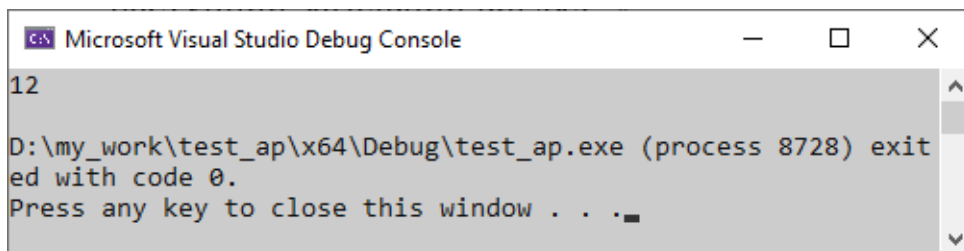
За замовчуванням для передавання аргументів у функцію використовується виклик за значенням, при якому у функцію передаються копії фактичних значень аргументів, з якими працюють оператори функції. Параметри-значення мають наступний загальний вигляд:

<тип_параметру> <ім'я_параметру>

Наприклад:

```
#include <iomanip>
#include <iostream>
using namespace std;

//реалізація функції
int sum(int a, int b) // параметри-значення
{
    return a + b;
}
//головна функція
int main()
{
    int x = 1;
    int y = 2;
    cout << sum(x, y) << endl; //виклик функції
    return 0;
}
```



Параметр-значення – це копія (інший екземпляр в пам'яті) аргументу. Параметри-значення можна використовувати лише для передавання інформації у функцію, бо зміна цих параметрів в тілі функції ніяк не вплине на аргументи.

Параметри-вказівники

Параметри-вказівники між типом та іменем параметру містять знак * «зірочка»:

<тип_параметру> *<ім'я_параметру>

Наприклад:

```
#include <iomanip>
#include <iostream>
using namespace std;

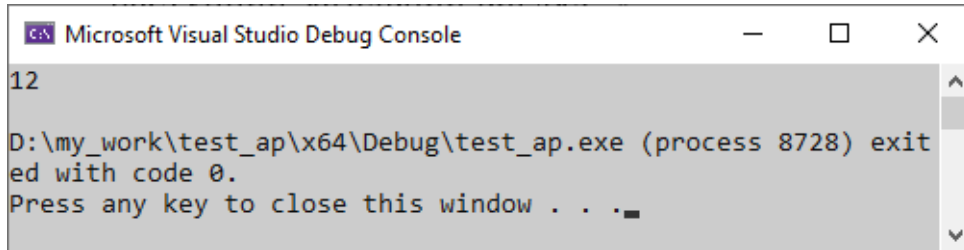
//реалізація функції
int sum(int *a, int *b) // параметри-значення
{
```



```

    return *a + *b;
}
//головна функція
int main()
{
    int x = 10; int* x_ = &x;
    int y = 2; int* y_ = &y;
    cout << sum(x_, y_) << endl; //виклик функції
    return 0;
}

```



Вказівник містить адресу деякої змінної. Операція `&` дозволяє отримати адресу. Таким чином, `&x` – це адреса змінної `x`. Цю адресу ми присвоюємо змінній `x_` – вказівнику на ціле, аналогічно – для змінної `y` та вказівника `y_`. В команді

```
int* x_ = &x;
```

оголошено вказівник на ціле `x_` – це змінна, яка може отримувати значення лише адрес змінних типу `int`. Таким чином, `int` – це базовий тип для вказівників `x_` та `y_`. Для доступу до значення змінної, адреса якої міститься в деякому вказівнику, слід роз-іменувати цей вказівник. Так звана дія «роз-іменування вказівника» – це перехід до комірки, адреса початку якої міститься у вказівнику, а розмір визначається базовим типом вказівника. В команді

```
return *a + *b;
```

спочатку роз-іменовуються вказівники `a` та `b`, тобто здійснюється «перехід за стрілочкою» – доступ до комірок, адреси яких зберігаються у вказівниках `a` та `b` відповідно. Потім обчислюється сума значень, що записані в цих комірках. Результат повертається в місце виклику функції `sum()`.

Передавання інформації у функцію за допомогою параметрів-вказівників – це застарілий спосіб, який в C++ не використовується.

Параметри-посилання

Параметри-посилання між типом та іменем параметру містять знак `&` «амперсанд»:

```
<тип_параметру> &<ім'я_параметру>
```

Наприклад:

```

#include <iomanip>
#include <iostream>

```

```
using namespace std;

//реалізація функції
int sum(int &a, int &b) // параметри-значення
{
    return a + b;
}
//головна функція
int main()
{
    int x = 10;
    int y = 2;
    cout << sum(x, y) << endl; //виклик функції
    return 0;
}
```

Параметр-посилання – це синонім (інша назва тої самої комірки пам'яті) аргументу.

Аргументом, який відповідає параметру-посиланню, може бути лише змінна того ж самого типу. Адреса цієї змінної передається у функцію. Параметр того самого типу, що і аргумент забезпечує виділення для параметра комірки того ж самого розміру, що і в аргументу. Оскільки адреси початку та розміри комірок пам'яті, виділених для параметра та аргументу – однакові, то параметр і аргумент займають одну і ту ж саму комірку пам'яті.

Параметр-посилання можна використовувати як для передавання інформації у функцію, так і для передавання інформації із функції, бо зміна цього параметру в тілі функції одразу приводить до відповідної зміни аргументу

Параметри зі значеннями за замовчуванням

Щоб спростити виклик функції, в її заголовку можна задати значення параметрів за замовчуванням. Ці параметри повинні записуватись останніми у списку параметрів функції. Вони можуть пропускатися у виклику функції. Якщо у виклику функції є пропущений параметр, тоді всі параметри, що стоять після нього, повинні бути пропущені.

Значеннями параметрів за замовчуванням можуть бути константи, глобальні змінні та вирази.

Розглянемо приклад прототипу функції з параметром за замовчуванням:

```
int sum(int a, int b = 0); // параметр b має значення за замовчуванням 0
```

Варіант виклику функції

```
int x = 10;
int y = 2;
sum(x, y); //результат 12
sum(x); //результат 10
```

Рекурсія

Рекурсія – це спосіб організації обчислювального процесу, при якому функція звертається сама до себе. Такі звернення називаються рекурсивними викликами, а функція, що їх містить, – рекурсивною. Це означає, що для вирішення певної задачі потрібно серед допоміжних підзадач вирішити таку саму задачу, тільки з іншими значеннями параметрів.

Розрізняють **пряму та непряму** рекурсію.

Пряма рекурсія полягає в тому, що певна функція у своєму тілі викликає сама себе, а **непряма** – коли дві чи більше функцій викликають одна одну.

Максимальна кількість копій рекурсивної функції, що одночасно може знаходитися в пам'яті комп'ютера, називається **глибиною рекурсії**. Від глибини рекурсії значною мірою залежить час виконання програми та обсяг необхідної стекової пам'яті. Значні витрати стекової пам'яті пов'язані з тим, що в рекурсивній підпрограмі, переважно, оголошується велика кількість локальних об'єктів: змінних, констант, типів, вкладених підпрограм тощо.

Рекурсивна функція обов'язково повинна містити **умову завершення**, інакше вона буде нескінченною. При кожному рекурсивному виклику підпрограми, в стеку виділяється пам'ять для всіх її локальних об'єктів, тому велика глибина рекурсії може призвести до нестачі стекової пам'яті.

Кожний рекурсивний виклик створює нову копію змінних рекурсивної функції, при цьому стара копія змінних не знищується, а зберігається в стеку. Це єдиний випадок, коли під час виконання програми назві однієї змінної відповідають кілька її копій. Цей процес відбувається у такій послідовності:

1. в стеку резервується місце для формальних параметрів, куди записуються значення фактичних параметрів. Переважно це виконується в порядку, зворотному до їхнього місця у списку параметрів функції;
2. при виклику функції в стек записується точка повернення, тобто адреса тої частини програми, де міститься виклик функції;
3. на початку тіла функції у стеку резервується місце для локальних (автоматичних) змінних.

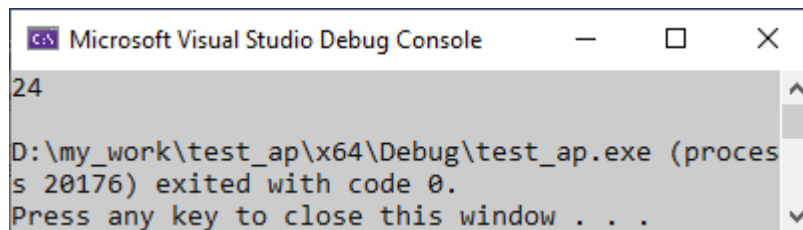
Зазначені змінні створюють групу (фрейм стека). Стек “пам'ятає історію” рекурсивних викликів у вигляді послідовності (ланцюга) таких фреймів. Програма у кохний конкретний момент працює з останнім викликом і з останнім фреймом. По завершенні рекурсії програма повертається до попередньої версії рекурсивної функції й до попереднього фрейму у стеку.

Створення нових копій рекурсивної функції до моменту досягнення граничної

умови називається *рекурсивним спуском*. Завершення роботи рекурсивних функцій, від останньої до найпершої, що ініціювала рекурсивні виклики, називається *рекурсивним підйомом*.

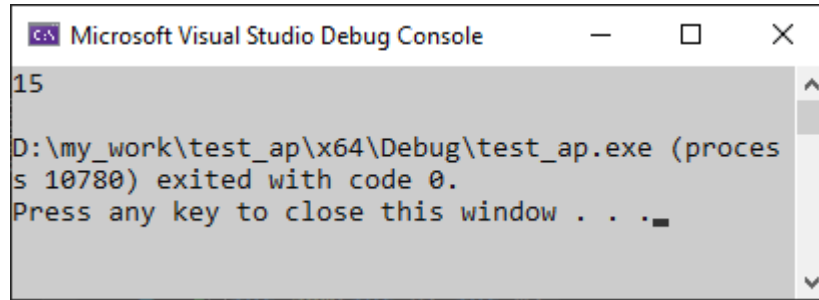
Напишемо функцію fact(), яка використовує таке визначення факторіалу, та програму, що її використовує:

```
#include <iostream>
using namespace std;
int fact(int n)
{
    int P = 1;
    for (int i = 1; i <= n; i++)
        P *= i;
    return P;
}
int main()
{
    int k = fact(4);
    cout << k << endl;
    return 0;
}
```



В наступному прикладі рекурсивно обчислюється сума всіх цілих чисел деякого діапазону

```
#include <iostream>
using namespace std;
//рекурсивне обчислення суми чисел від n до m
int sum(int n, int m)
{
    int s = 0;
    if (n == m) s = n;
    else
        s = sum(n,m-1)+m;
    return s;
}
//головна функція
int main()
{
    int k = sum(1,5); //виклик функції для чисел від 1 до 5
    cout << k << endl;
    return 0;
}
```



The image shows a screenshot of the 'Microsoft Visual Studio Debug Console' window. The window has a title bar with the Visual Studio icon and the text 'Microsoft Visual Studio Debug Console'. Inside the console, the text reads: '15', 'D:\my_work\test_ap\x64\Debug\test_ap.exe (process 10780) exited with code 0.', and 'Press any key to close this window . . .'. There is a vertical scrollbar on the right side of the console area.

```
15
D:\my_work\test_ap\x64\Debug\test_ap.exe (process 10780) exited with code 0.
Press any key to close this window . . .
```

Зазвичай, програмний код, який реалізує рекурсивний спосіб – простіший.