МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

МЕТОДИЧНІ ВКАЗІВКИ

до виконання лабораторної роботи 3

за темою «Використання функцій»

з курсу «Алгоритмізація та програмування. Частина 1»

для студентів спеціальності
122 «Комп’ютерні науки»

Харків 2019

Методичні вказівки до виконання лабораторної роботи 3 за темою «Використання функцій» з курсу «Алгоритмізація та програмування. Частина 1» для студентів спеціальності 122 «Комп'ютерні науки» / уклад. Л. В. Іванов, М. О. Білова. – Харків : НТУ «ХПІ», 2019. – 23 с.

Укладачі: Л. В. Іванов
М. О. Білова

Рецензент Т. В. Козуля

Кафедра програмної інженерії та інформаційних технологій управління

ЗМІСТ

Вступ

Курс «Алгоритмізація та програмування» присвячений теоретичним та практичним аспектам розробки алгоритмів і програм мовою С++. Отримані в результаті вивчення даної дисципліни знання та навички можуть бути використані в усіх наступних курсах, під час виконання курсових та дипломних проектів, вивчення дисциплін, що пов’язані з обчислювальною технікою та програмуванням.

У процесі виконання лабораторної роботи за темою «Використання функцій» студенти мають закріпити теоретичний матеріал, отримати навички практичної роботи при реалізації програм, пов’язаних з рекурсивними функціями, статичними змінніми, агруменитами з усталеними значеннями на мові С+.

Перша частина присвячена теоретичним аспектам, зокрема визначенню функцій в мові програмування С++, області видимості функції, статичним локальним змінним. У теоретичній частині надано визначення рекурсивних функцій, розглянуто принципи роботи з ними. Охарактеризовано процес перевантаження імен функцій, роботу з посиланнями та усталеними значеннями аргументів. Теоретичні аспекти, розглянуті у методичні вказівках, супроводжуються прикладами програмного коду.

Друга частина складається з завдань для лабораторної роботи, що використовуються як поточний контроль засвоєння матеріалу. Лабораторні заняття розраховані на роботу в комп’ютерному класі під безпосереднім керівництвом викладача та самостійну роботу студентів, яка передбачає написання тексту програми, закріплення практичних навичок роботи на комп’ютері, набутих при виконанні відповідної лабораторної роботи. Варіанти індивідуальних завдань видаються викладачем на занятті.

У методичних вказівках подано п’ять завдань. У цілому методичні вказівки будуть корисні студентам різних спеціальностей і форм навчання, які вивчають мову С ++.

ТЕОРЕТИЧНА ЧАСТИНА

1. Оголошення та визначення функції

Функція (function) – це підпрограма, яка може отримати дані та повертати деяке значення. Кожна функція має своє власне ім’я. Коли це ім’я зі списком даних (параметрів) зустрічається будь-де у програмі, виконання програми переходить до тіла цієї функції. Такий перехід має назву **виклику функції** (function call). Після того як виконані інструкції в тілі функції, процес виконання повертається до попередньої функції (з якої було здійснено виклик) та виконуються її наступні інструкції. Виклик функції main() здійснюється операційною системою. Складні програми повинні бути розділені на декілька функцій, виклик яких здійснюється по черзі.

Визначення функції включає **тіло функції** – блок з програмним кодом, який виконується під час виклику функції.

Оголошення функції без визначення має назву **прототипу** (prototype). Прототип функції складається з типу результату функції, імені та списку параметрів. **Список параметрів** (parameter list) – це список усіх аргументів та їх типів, розділених комами:

```
int sum(int a, int b); // прототип
```

Прототип функції та визначення функції повинні збігатися щодо типу результату, імені та списку параметрів. Прототип функції не обов’язково містить імена параметрів:

```
int sum(int, int); // прототип
```

Визначення функції (function definition) складається із заголовка функції і тіла. **Тіло функції** (function body) – це набір інструкцій, укладених у фігурні дужки.

```
int sum(int a, int b)
{
    return a + b;
}
```

Усі функції мають тип результату. Якщо нічого не вказано явно, тип результату – `int`.

Інструкція `return` завершує виконання функції та повертає управління до попередньої функції, з якої було здійснено виклик. Інструкція return всередині функції `main()` передає управління операційній системі. Значення виразу в інструкції `return` повертається до функції, з якої було здійснено виклик. Функції усіх типів (за винятком `void`) повинні містити інструкцію `return` з таким виразом. Інструкція `return` завжди завершує виконання функції. Результат функції `main()` може бути використаний операційною системою як код помилки (0 – помилок немає). Функцію не можна визначити всередині іншої функції. У C++ немає локальних функцій.

Незважаючи на відсутність обмеження на розмір функції, добре спроектовані функції зазвичай містять не дуже багато коду. Функцію з меншим розміром легше зрозуміти та підтримувати її роботу.

Аргументи передаються у функцію у тому порядку, в якому вони були оголошені і визначені. Будь-який вираз C++ може бути аргументом функції, у тому числі константа, математичні та логічні вирази, виклик інших функцій, які повертають значення:

```
int main()
{
    int x, y;
    cin >> x >> y;
    int z = sum(x, y);              // виклик функції
    int k = sum(z, x + y);          // виклик функції
    int m = sum(k, sum(x + y, z)); // виклик функції
    cout << z << ' ' << k << ' ' << m;
    return 0;
```

```
}
```

Аргументи, передані у функцію, є локальними для функції. Зміни значень аргументів всередині викликаної функції не впливають на значення в функції, з якої було здійснено виклик. Це відомо як передача за значенням і означає, що у функції створена локальна копія кожного аргументу.

Аргументи, які передаються під час виклику функції мають назву **фактичних параметрів**. Викликана функція має доступ до переданої інформації через так звані **формальні параметри**.

2. Тип void

Якщо функція не повертає ніяких значень, її тип повинен бути `void`:

```
void print(double a)
{
    cout << a << endl;
}
```

Тип `void` синтаксично є фундаментальним типом. Він може, однак, бути використаний тільки як частина більш складного типу; не існує об'єктів типу `void`.

У тілі функції з типом результату `void` інструкція `return` може бути відсутня: тіло функції виконується до кінця програмного блоку. Якщо інструкція `return` присутня, то після неї не повинно бути жодного виразу. Інструкцію `return` можна використовувати для дострокового повернення з функції. Наприклад:

```
void printReciprocal(double a)
{
    if (a == 0)
    {
        return;
```

```
    }
    cout << 1 / a << endl;
}
```

Функцію з типом результату `void` можна викликати тільки окремою інструкцією (не всередині виразу).

3. Область видимості

Кожен програмний об'єкт, зокрема змінна, має свою **область видимості**, яка визначає, де саме у програмі цей об'єкт може бути доступний.

Змінні можна створити всередині тіла функції, або іншого програмного блоку. Такі змінні є **локальними змінними**. Вони існують тільки всередині функції під час її виконання. Формальні параметри також є локальними змінними.

Глобальні змінні мають глобальну область видимості. Вони доступні у будь-якому місці програми. Глобальні змінні, визначені поза тілом функції, доступні з будь-якої функції у програмі, у тому числі з функції `main()`. Локальні змінні з тим же ім'ям, що й глобальні змінні, не пов'язані з відповідними глобальними змінними. Але локальна змінна з таким же ім'ям, як і глобальна змінна, **приховує** глобальну змінну. У таких випадках дозволу області оператор (: :) для доступу до глобальної змінної використовують операцію **доступу до області видимості** (scope resolution operator). Наприклад:

```
int k = 1;
void f()
{
    int k = 2;
    cout << k;    // 2
    cout << ::k; // 1
}
```

Синтаксис мови C++ дозволяє використовувати глобальні змінні, їх використання є небажаним. Глобальні змінні створюють небезпеку, оскільки вони є спільними даними. Будь-яка функція може змінити значення глобальної змінної, після чого інша функція, в якій передбачено використання певного значення глобальної змінної, може неправильно виконуватися. Такі помилки дуже важко знаходити.

4. Статичні локальні змінні

Як правило, локальні змінні, створені у функції, зникають після виходу з функції. Коли виклик функції здійснюється знову, змінні створюються та ініціалізуються заново. Якщо треба, щоб дані зберігалися протягом усього життєвого циклу програми, така змінна може бути визначена як статична (з модифікатором static). Ініціалізація виконується тільки тоді, коли функція викликається вперше, а змінні зберігають свої значення проміж викликами функції.

Незважаючи на те що статичні змінні існують протягом життєвого циклу програми, не існує механізмів доступу до таких змінних ззовні функції.

5. Рекурсія

Функція може викликати себе. Такий механізм має назву **рекурсії.**

Рекурсія може бути безпосередньою або опосередкованою. Безпосередня рекурсія передбачає виклик функції безпосередньо з цієї функції, опосередкована передбачає, що перша функція викликає другу, а потім друга викликає першу функцію.

Важливо відзначити, що коли функція викликає саму себе, створюється нова копія даних цієї функції. Локальні змінні у новій копії не залежать від локальних змінних попередньої копії.

Іноді рекурсія може бути використана замість циклу. Наприклад, можна обчислити суму $y = 1^2 + 2^2 + 3^2 + ... + n^2$ взагалі без циклів:

```
#include <iostream>
using namespace std;
double sum(int n)
{
    if (n <= 1)
    {
        return 1;
    }
    else
    {
        return n * n + sum(n - 1);
    }
}

int main()
{
    cout << sum(5);
    return 0;
}
```

Неправильне вживання рекурсії може призвести до переповнення програмного стека.

6. Функції-підстановки

Коли здійснюється виклик функції, виконання програми переходить до відповідної групи інструкцій. Коли виконання функції завершується, виконання переходить назад на наступний рядок функції, з якої було здійснено виклик. Ці дії можуть бути пов’язані з деяким зниженням ефективності виконання.

Існують так звані **функції-підстановки**, або вбудовані. Такі функції мають модифікатор `inline`:

```
inline int min(int a, int b)
{
    return a < b ? a : b;
}
```

Якщо функція оголошена як вбудована, компілятор підставляє у точку виклику її тіло. Доцільно з модифікатором `inline` описувати найпростіші функції, багаторазова підстановка яких не істотно вплине на розміри програми.

7. Перевантаження імен функцій

Мова C++ дозволяє створити декілька функцій з однаковими іменами. Це називається **перевантаженням імен функцій** (function name overloading). Перевантажені функції дозволяють програмістам реалізувати різну семантику для функції залежно від типів і кількості аргументів.

Функції повинні відрізнятися списками параметрів: різними типами параметрів, різною кількістю параметрів, або обома ознаками. Наприклад:

```
int sum(int a, int b)
{
    return a + b;
}

double sum(double a, double b)
{
    return a + b;
}

double sum(double a, double b, double c)
{
    return a + b + c;
}

int main()
{
    cout << sum(1, 2) << endl;       // перша функція sum()
    cout << sum(1.0, 2.5) << endl;   // друга функція sum()
    cout << sum(1, 2, 2.6) << endl; // третя функція sum()
    return 0;
}
```

Примітка*:* дві функції з однаковим ім’ям і списком параметрів, але з різними типами результату, генерують помилку компіляції:

```
int f(double x);
double f(double x); // Помилка!
```

8. Усталені значення аргументів

У багатьох випадках для аргументів функції можна вказати усталені значення (default values), які можуть бути цілком задовільними у багатьох випадках. Для вирішення цієї проблеми реалізовано механізм **аргументів з усталеними значеннями**, що дозволяє визначати лише ті аргументи функції, які є істотними у конкретному контексті. Наприклад:

```
int sum(int x, int y = 0, int z = 0)
{
    return x + y + z;
}

int main()
{
    cout << sum(5) << endl;        // 5, y = 0, z = 0
    cout << sum(1, 2) << endl;     // 3, z = 0
    cout << sum(1, 2, 5) << endl; // 8
    return 0;
}
```

Аргументи з усталеними значеннями використовують у випадках, коли параметри у кінці списку можуть бути відсутні. Усі параметри функції можуть бути з усталеними значеннями. Слід пам’ятати, що аргументи з усталеними значеннями повинні бути останніми в списку:

```
void f(double x, int y = 0, int h); // помилка
```

Усталене значення не може бути повторене у більш пізніх описах функції, навіть якщо перевизначення ідентичне оригіналу. Таким чином, наведений нижче код видає помилку:

```
void f(double x, int y = 1);
void f(double x, int y = 1) { } // помилка
  // Слід писати void f(double x, int y) { }
```

9. Посилання

У мові C++ визначено механізм створення так званих посилань. **Посилання** (reference) може бути визначене як друге ім’я (псевдонім) існуючої змінної (об’єкта).

Можна описати посилання ім’ям типу за значенням типу, за яким слідує оператор посилання (`&`) та ім’я посилання:

```
int i = 10;
int &j = i; // посилання на i
j = 11;
cout << i;  // 11
```

Не можна створювати посилання не вказуючи об’єкт, на який посилаємось. Посилання повинні бути ініціалізовані під час створення:

```
int &k; // синтаксична помилка!
```

Можна визначити посилання на константний об’єкт:

```
int m = 2;
const int &n = m; // посилання на i
double x = n + 1; // OK
m = 11;           // OK
n = 12;           // помилка!
```

Посилання можуть бути використані як аргументи функції. Наприклад, можна створити функцію, яка обмінює значення двох цілих змінних:

```
void swap(int &x1, int &x2)
{
    int x = x1;
    x1 = x2;
    x2 = x;
}
```

Тепер функцію `swap()` можна викликати з функції `main()`:

```
void main()
{
    int a = 1;
    int b = 2;
    swap(a, b);
    cout << a << endl; // 2
    cout << b << endl; // 1
}
```

Посилання можуть бути використані як тип результату функції. Дуже важливо гарантувати, що посилання пов’язано зі змінною, яка існує після виходу з функції:

```
int& f()
{
    int k;
    return k; // Помилка; k знищується після виходу з
функції
}
// А це правильно:
int& f()
{
    static int k;
    return k; // OK
}

void main()
{
    f() = 10;
    cout << f(); // 10
}
```

Функція повертає псевдонім статичної змінної, яка не знищується після виходу з функції. Таким чином, можна здійснити таке присвоєння:

```
f() = 10;
```

ПРИКЛАДИ ПРОГРАМ

1. Статичні локальні змінні. У наведеній нижче програмі обчислюється сума цілих значень, які вводить користувач:

```
#include <iostream>
using namespace std;
int add(int i)
{
    static int sum = 0;
    sum += i;
    return sum;
}
int main()
{
    int i;
    do
    {
        cin >> i;
        cout << add(i) << endl;
    }
    while (i);
    return 0;
}
```

2. Поділ програми на незалежні частини. Майже всі програми можна розділити на відносно незалежні частини. Програма, яка обчислює деякі значення, як правило, містить частини, відповідальні за введення, розрахунки і виведення. Припустимо, що нам необхідно реалізувати програму, яка обчислює суму квадратів цілих чисел:

$$y = 1^2 + 2^2 + 3^2 + ... + n^2.$$

Перша реалізація не поділена на частини:

```
#include <iostream>
```

```
using namespace std;
int main()
{
    int n;
    cout << "Input n:";
    cin >> n;
    int y = 0;
    for (int i = 1; i <= n; i++)
    {
        y += i * i;
    }
    cout << "y = " << y;
    return 0;
}
```

Найпростіше рішення полягає у використанні глобальних змінних:

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

int n;
int y = 0;

void read()
{
    cout << "Input n:";
    cin >> n;
}

void calc()
{
    for (int i = 1; i <= n; i++)
    {
        y += i * i;
    }
```

```
}

void write()
{
    cout << "y = " << y;
}

int main()
{
    read();
    calc();
    write();
    return 0;
}
```

Використання глобальних змінних не є гарною ідеєю. Кращий підхід полягає у використанні параметрів:

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

int read()
{
    int n;
    cout << "Input n:";
    cin >> n;
    return n;
}

int calc(int n)
{
    int y = 0;
    for (int i = 1; i <= n; i++)
    {
```

```
        y += i * i;
    }
    return y;
}

void write(int y)
{
    cout << "y = " << y;
}

int main()
{
    int n = read();
    int y = calc(n);
    write(y);
    return 0;
}
```

Функція `main()` може бути реалізована взагалі без змінних:

```
...
int main()
{
    write(calc(read()));
    return 0;
}
```

3. Лінійне рівняння. Функція `solve()` у наступному прикладі знаходить корінь лінійного рівняння і повертає `true`, якщо цей корінь існує. У протилежному випадку вона повертає `false`.

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

```
bool solve(double a, double b, double& x)
{
    if (a == 0)
    {
        return false;
    }
    x = -b / a;
    return true;
}
int main()
{
    double a, b, x;
    cin >> a >> b;
    if (solve(a, b, x))
    {
        cout << x;
    }
    else
    {
        cout << "Error";
    }
    return 0;
}
```

ВПРАВИ ДЛЯ КОНТРОЛЮ

Завдання 1. Створити програму для тестування функції `signum()`.

Завдання 2. Здійснити розробку та тестування функції, яка обчислює добуток трьох аргументів.

Завдання 3. Здійснити розробку та тестування функції, яка обчислює добуток перших *n* непарних значень.

Завдання 4. Здійснити розробку та тестування функції, яка обчислює e^x через суму.

Завдання 5. Здійснити розробку та тестування функції, яка обчислює факторіал.

Завдання 6. Здійснити розробку та тестування функції, яка здійснює виведення всіх парних значень у заданому діапазоні.

Завдання 7. Здійснити розробку та тестування функції, яка здійснює виведення добутку перших *n* парних значень.

Завдання 8. Здійснити розробку та тестування функції, яка обчислює найбільший спільний дільник двох цілих чисел.

ЗАВДАННЯ НА ЛАБОРАТОРНУ РОБОТУ

1. Статичні локальні змінні

Написати програму, яка обчислює та виводить мінімум і максимум цілих чисел, у міру того, як користувач вводить ці числа. Слід використати статичні локальні змінні.

2. Рекурсія

Написати програму, яка зчитує *x* і *n* і обчислює *y* за допомогою рекурсивної функції:

$$y = (x+1)(x+2)(x+3)(x+4)...(x+n).$$

3. Аргументи з усталеними значеннями

Створити функції, які повертають 1, аргумент, і добуток аргументів залежно від кількості аргументів. Першу функцію реалізувати з усталеними значеннями аргументів, інші функції – через механізм перевантаження імен. У функції `main()` слід здійснити тестування всіх функцій.

4. Квадратне рівняння

Створити функцію для розв’язання квадратного рівняння. Функція повинна повертати кількість коренів або –1, якщо рівняння має безліч розв’язків. Функція повинна отримати коефіцієнти якості аргументів та повертати корені як аргументи-посилання.

5. Індивідуальне завдання

Створити програму, яка реалізує індивідуальне завдання попередньої лабораторної роботи. Програма повинна бути розділена на декілька функцій. Функція `y()` повинна отримувати значення `x` і `n` як аргументи і повертати значення, розраховані за формулою, що наведена в індивідуальному завданні. Створити окрему функцію для зчитування даних. Не використовувати глобальні змінні.

КОНТРОЛЬНІ ЗАПИТАННЯ

1. Що таке функція?
2. Що таке заголовок функції?
3. Що таке тіло функції?
4. Що таке прототип функції?
5. Чи є імена формальних параметрів завжди обов'язковими у заголовку функції?
6. Як створити локальну функцію?
7. Як здійснюється передача параметрів у функцію?
8. У чому різниця між формальними і фактичними параметрами?
9. Чи є інструкція `return` завжди обов'язковою?
10. Як використовувати результат функції з типом результату `void`?
11. Що таке локальна змінна?
12. Що таке область видимості (scope)?
13. Що таке глобальна область видимості?
14. Як отримати доступ до імен з глобальної області видимості?
15. Чи можуть локальні змінні приховати глобальні змінні?
16. У чому різниця між статичними і нестатичними локальними змінними?
17. Яким є життєвий цикл статичної локальної змінної?
18. Як використовують статичні локальні змінні?

19. Що таке рекурсія?

20. Що таке функція-підстановка (inline)?

21. Як перевантажити ім’я функції?

22. Чи можна створити дві глобальні функції з однаковими іменами і списками параметрів, але з різними типами результату?

23. Як описати аргументи з усталеними значеннями?

24. Що таке посилання?

25. Як ініціалізувати посилання?

26. Для чого використовують посилання?

27. Чи можна повернути посилання з функції?

28. Чи можна повертати посилання на нестатичну локальну змінну?

Рекомендована література

1. Bjarne Stroustrup. The C++ Programming Language. Third Edition / Bjarne Stroustrup. – Addison-Wesley, 1997.

2. Stanley B. Lippman C++ Primer. Third Edition / Stanley B. Lippman, Josee Lajoie. – Addison-Wesley, 1988.

3. Deitel H. M. C++. How to Program. Third Edition / H. M. Deitel, P. J. Deitel. – Prentice Hall, 2001.

4. Голуб Б. М. C#. Концепція та синтаксис / Б. М. Голуб. – Львів: Видавничий центр ЛНУ ім. Івана Франка, 2006. – 136 с.

5. Грицюк Ю. І. Програмування мовою С++: навч. посібник / Ю. І. Грицюк, Т. Є. Рак. – Львів : Вид-во Львівського ДУ БЖД, 2011. – 292 с.

Internet-джерела

1. The C++ Programming Language (Bjarne Stroustrup's homepage) // http://www2.research.att.com/~bs/C++.html

2. ISO/IEC 14882:2003 Programming languages - C++ (International Stamdard) // http://cs.nyu.edu/courses/summer12/CSCI-GA.2110-001/downloads/C++%20Standard%202003.pdf

3. The C++ Resources Network // http://www.cplusplus.com/

4. The C++ Tutorial // http://www.learncpp.com/

5. C++ – Вікіпідручник // http://uk.wikibooks.org/wiki/C++

6. C++ – Вікіпіедія // http://uk.wikipedia.org/wiki/C++

7. Корх О. Основи мови програмування С++ // http://korkholeh.googlepages.com/cppfund.pdf

Навчальне видання

Методичні вказівки

до виконання лабораторної роботи 3

за темою «Використання функцій»

з курсу «Алгоритмізація та програмування. Частина 1»

для студентів спеціальності

122 «Комп’ютерні науки»

Укладачі:

ІВАНОВ Лев Вадимович

БІЛОВА Марія Олексіївна

Відповідальний за випуск М. Д. Годлевський

Роботу до видання рекомендував О. В. Горілий

План 2018 р., поз. 312

Підписано до друку 28.05.2019. Гарнітура Times New Roman.

Ум. друк, арк. 0,9.

Видавничий центр НТУ «ХПІ»,

вул. Кирпичова, 2, м.Харків-2, 61002

Свідоцтво про державну реєстрацію ДК № 3478 від 21.08.2017 р.

Самостійне електронне видання