**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**

**НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ "ЛЬВІВСЬКА ПОЛІТЕХНІКА"**

Інститут **КНІТ**

Кафедра **ПЗ**

# ЗВІТ

До лабоаторної роботи № 7

**На тему:** *«* *ВКАЗІВНИКИ НА ФУНКЦІЇ. РЕКУРСИВНІ ФУНКЦІЇ.»*

**З дисципліни:** *“Основи програмування”*

**Лектор:**

ст. викладач кафедри ПЗ

Муха Т.О.

**Виконав:**

студ. групи ПЗ-13

Топала Т.Д.

**Прийняла:**

доц. кафедри ПЗ

Дяконюк Л.М.

«\_\_\_» \_\_\_\_\_\_ 2021 р.

∑ = \_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_

Львів – 2021

**Тема роботи:** ОСОБЛИВОСТІ РОБОТИ З ФУНКЦІЯМИ В С. ДИРЕКТИВИ ПРЕПРОЦЕСОРА.

**Мета роботи:** поглиблене вивчення можливостей функцій в мові С з використанням механізмів рекурсії та вказівників.

## Індивідуальне завдання

1. Ознайомитися з теоретичним матеріалом викладеним вище в даній інструкції і виконати приклади програм.
2. Одержати індивідуальне завдання з Додатку 1.
3. Розробити алгоритм розв’язання індивідуального завдання.
4. Скласти програму на мовіС у відповідності з розробленим алгоритмом.
5. Виконати обчислення по програмі.
6. Одержати індивідуальне завдання з Додатку 2.
7. Розробити алгоритм розв’язання індивідуального завдання.
8. Скласти програму на мовіС у відповідності з розробленим алгоритмом.
9. Виконати обчислення по програмі при різних значеннях точності і порівняти отримані результати.
10. Одержати індивідуальне завдання з Додатку 3.
11. Скласти програму на мовіС для реалізації індивідуального завдання.
12. Підготувати та здати звіт про виконання лабораторної роботи.

**Завдання 1.** Використовуючи вищенаведені функції swap та qs\_sort, які реалізують алгоритм швидкого сортування масиву, написати програму мовою С для порівняння ефективності алгоритмів сортування масивів великих обсягів (наприклад, 100000 елементів). Програма повинна також реалізувати один з класичних алгоритмів сортування масиву згідно з варіантом індивідуального завдання. У програмі використати два одинакових масиви, які заповнити випадковими числами, здійснити перевірку впорядкованості елементів масиву, перевірку ідентичності масивів до і після сортування, а також за допомогою стандартної функції time, оцінити час виконання реалізованих алгоритмів сортування.

Сортування в порядку зростання методом вибору з пошуком мінімального елемента.

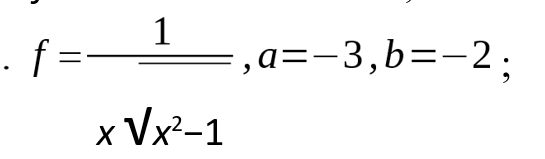
**Завдання 2.** Написати мовою С три функції, щоб протабулювати, задану згідно варіанту, функцію на проміжку [a , b] з кроком h, використавши:

а) для першої функції оператор циклу for;

б) для другої – оператор циклу while;

в) для третьої – – оператор циклу do…while.

Вибір способу табулювання реалізувати через вказівник на відповідну функцію.



**Завдання3.** Знайти найбільший елемент заданого одновимірного масиву, використовуючи рекурсивну функцію.

## Теоретичні відомості

**ТЕОРЕТИЧНІ ВІДОМОСТІ**

### 2.1 Вказівник на функцію

Як і дані, функція зберігається у пам'яті, а отже, вона має адресу. Тому на функцію можна встановити вказівник. Робиться це за допомогою такої конструкції:

*тип\_результату(\* ім’я\_вказівника)(список\_параметрів);*

Саме розміщення вказівника у круглих дужках і означає, що він посилається на функцію. Без круглих дужок значення конструкції зовсім інше – оголошувалася б функція, що повертає вказівник на дані зазначеного типу. Порівняйте:

*int (\*p)(int)* /\* оголошується вказівник на функцію, що повертає результат типу int \*/ *int \*p(int)* /\* оголошується функція, яка повертає вказівник на ціле число \*/

Оскільки в С ім’я функції є вказівником на першу комірку області пам'яті, де зберігається функція, то для ініціалізації слід просто присвоїти вказівнику ім’я функції. Наприклад, нехай ми маємо такі прототипи функцій:

*int f3 (int ); int f5 (int ); int f7 (int );*

Тоді можна описати такий вказівник на функцію, яка повертає результат типу *int* та має один формальний параметр також типу *int*:

*int (\*fst) (int );*

Після такого оголошення, оператор присвоєння *fst = f3;* присвоїть вказівнику *fst* адресу входу у функцію *f3*, оператор *fst = f5*; - адресу входу у функцію *f5* і, відповідно, *fst = f7;* - адресу входу у функцію *f7*. Після цього викликати кожну з функцій (*f3, f5, f7*) можна будь-яким оператором, записаним нижче (наприклад, для функції *f3*): *f3(a); // звертання до функції, використовуючи її ім'я.*

*(\*fst) (а); // виклик функції через вказівник. fst(a); // виклик функції також через вказівник.*

Останній варіант також правильний, тому що і *f3*, і *fst* - це адреси входу у функцію. Однак виклик *(\*fst) (а)* явно показує, що використовується вказівник на функцію, а не викликається функція з ім'ям *fst.*

Вказівники на функцію широко застосовуються в програмуванні:

* багато бібліотечних функцій як аргумент одержують вказівник на функцію;
* використання вказівників на функцію як аргументів дозволяє розробляти універсальні функції, наприклад функції чисельного рішення рівнянь, чисельного інтегрування й диференціювання;
* масиви вказівників на функції використовуються для організації меню.

Як і звичайні змінні, вказівники на функції можуть об’єднуватися в масиви. Наприклад, визначити й проініціалізувати масив вказівників на функції можна в такий спосіб. Нехай є такі функції:

*float funcl(float); float func2(float); float func3(float); float func4(float); float func5(float);*

Тоді масив вказівників на функції можна описати так:

*float (\*fparray [5])(float) = { funcl, func2, func3, func4, func5};*

Доступ до елементів масиву *fparray* виконується, як до звичайних елементів масиву. Наприклад:

*float х = 1; printf(“%f”, fparray[0](х)); // або printf(“%f”, (\*fparray[0]) (х);*

Вказівники на функції застосовуються у випадках, коли потрібно викликати різні функції залежно від певних умов.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Використання вказівників на функцію \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <stdio.h>

#include <stdlib.h>

#include <conio.h>

double suma(double,double); double dobutok(double,double);

void main()

{ double (\*p)(double,double); int n1, n2; char ch;

printf("Enter number 1\n");

scanf("%d", &n1); printf("Enter number 2\n");

scanf("%d", &n2);

printf("Sum or divide (s|another key)?\n");

scanf("%с", &ch); if(ch=='s') p = &suma; else p = dobutok;

printf("%lf", p(n1,n2));

\_getch();

}

double suma(double n1,double n2)

{ return (n1+n2);

}

double dobutok(double n1,double n2)

{ return (n1\*n2);

}

У даній програмі оголошено дві функції – suma та dobutok. Обидві функції приймають по два параметри типу double та повертають результат типу double. У тілі функції main() оголошено вказівник p на функцію, яка приймає два параметри типу double та повертає значення типу double: (double (\*p)(double,double)). Користувачеві пропонується ввести з клавіатури два числа, які будуть записані у змінні n1, n2, та вибрати дію над цими числами – додати чи помножити. Для додавання чисел користувачеві пропонується натиснути клавішу ‘s’, а для множення – довільну іншу клавішу. Якщо користувач натиснув ‘s’, вказівнику p присвоюється адреса функції suma оператором p = &suma; інакше у вказівник p записується ім’я функції dobutok (тобто вказівник на першу комірку адреси області пам’яті, де зберігається функція) за допомогою оператора p = dobutok; Обидва записи є коректними. Виклик вибраної функції, передача параметрів і вивід результатів на екран здійснюється так: printf("%lf", p(n1,n2)); На рис. 1 показані результати роботи програми для дії додавання та множення.

Зображення, що містить текст

Автоматично згенерований опис

*Рис. 1 Приклад роботи програми із застосуванням вказівника на функцію*

### 2.2. Рекурсивні функції

В алгоритмічній мові С функції можуть бути рекурсивними. ***Рекурсивною*** називається така функція, яка звертається сама до себе (викликає саму себе). У рекурсивній функції обчислювальний процес повинен бути організований так, щоб на кожному кроці рекурсії задача поступово спрощувалася до тих пір поки для неї не стане можливим не рекурсивне рішення. Саме тому рекурсивна функція повинна повертати конкретний результат тільки для найпростішої частини задачі, тобто тоді коли можливо обчислити результат безпосередньо. В усіх інших випадках функція викликає саму себе, але з передачею нового значення аргументу, при якому нова задача є більш простою для розв’язання підзадачею вихідної задачі, причому алгоритм розв’язування цієї нової задачі є аналогічним алгоритму розв’язування вихідної задачі.

Створюючи рекурсивну функцію слід дотримуватися таких правил:

1. при кожному виклику такої функції в неї повинні передаватися нові дані;
2. на якомусь етапі повинен бути припинений подальший виклик даної функції;
3. після завершення кожного виклику рекурсивної функції в точку повернення повиненпередаватися деякий результат для подальшого використання.

Рекурсія не є властивістю функції, це властивість опису функції. Про це свідчить такий простий приклад. Нехай потрібно знайти найбільше серед двох заданих чисел. Виявляється, що ця проста задача може бути розв’язана також і рекурсивним способом, як це показано у наступній програмі.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*функцію max можна зробити рекурсивною\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include<stdio.h> #include<conio.h> int max(int, int);

void main()

{ int a,b;

printf("Enter a,b:"); scanf("%d%d", &a, &b); printf("Max is %d", max(a,b));

\_getch();

}

int max(int a, int b)

{

if (a<b) max(b,a); // рекурсія в дії

else return a;

}

У функцію max передаються значення двох цілих чисел. Функція повертає результат лише в тому випадку коли більше число задане першим аргументом. У протилежному випадку функція max звертається до самої себе, однак параметри а і b у виклику міняються місцями, тобто параметру а у функції передається більше значення (b), а параметру b - менше значення (a). Якщо ж спочатку умова а < b не виконується, то відразу ж буде мати місце не рекурсивний вихід.

Рекурсія нерідко входить у визначення математичних функцій. Найтиповішим прикладом є обчислення факторіала за допомогою рекурсії. Розглянемо на цьому прикладі детальніше механізм рекурсивних функцій. Програма мовою С, яка реалізує задачу обчислення факторіала від заданого числа за допомогою рекурсії може мати такий вигляд.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* обчислення факторіала рекурсивно \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include<stdio.h> #include<conio.h>

unsigned long factorial(unsigned long);

void main()

{

for(int i=1; i<=15; i++)

printf("%d != %d\n", i, factorial(i));

\_getch();

}

unsigned long factorial(unsigned long number)

{

if(number==1) return 1; else

return number\*factorial(number-1);

}

Пояснимо тепер процес виконання цієї програми. Функція main викликає функцію factorial, яка має параметр number. Якщо виконується умова number==1, то повернеться значення 1, інакше відбувається рекурсивний виклик функції з передачею нового значення аргумента number-1. Цей рекурсивний процес буде тривати до виклику factorial(1), який поверне значення 1 в останню точку виклику.

Порядок виконання кожного виклику функції factorial і кожного повернення через оператор return для обчислення 5! схематично зображено на рис2.

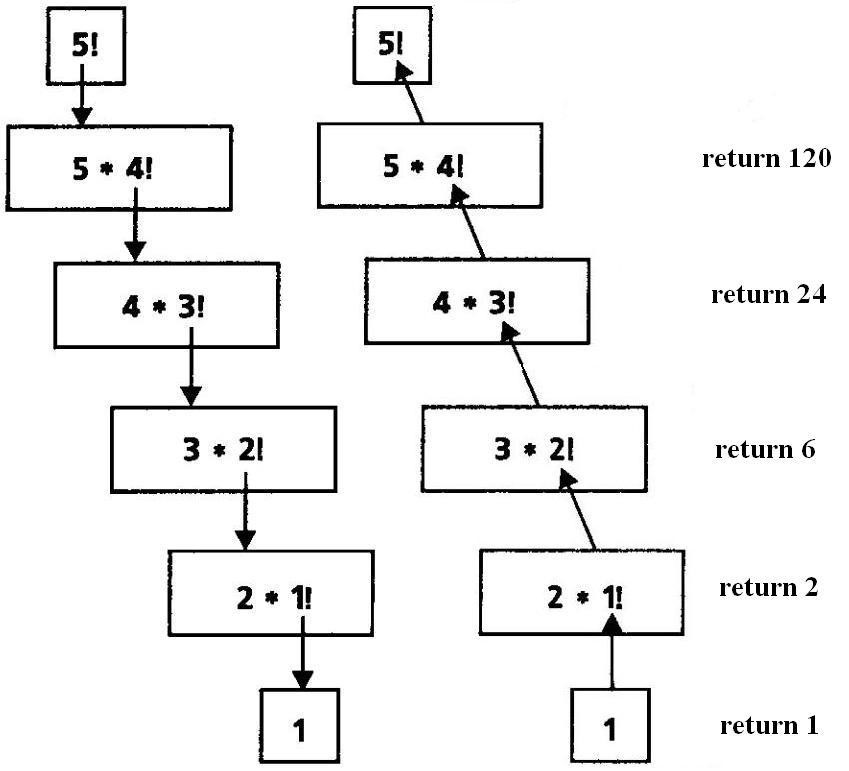


Рис.2. Порядок виконання рекурсивної функції обчислення факторіалу

З цього рисунку добре видно, що в процесі виконання рекурсивної функції спочатку відбувається послідовне звертання функції до самої себе до настання найпростішого випадку обчислення цієї функції, коли стає можливим просте обчислення значення функції. Після цього моменту відбувається зворотній хід, тобто послідовне повернення результату кожного рекурсивного виклику в те місце функції, де він відбувся, до самого першого виклику (у цю точку виклику і буде повернено остаточний результат.

Іншим класичним прикладом застосування рекурсії є обчислення чисел, що належать послідовності Фібоначі. Текст програми мовою С, яка розв’язує цю задачу подано нижче.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*рекурсивне обчислення чисел Фібоначі \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include<stdio.h> #include<conio.h>

unsigned long fibo(unsigned long);

void main()

{

unsigned long n;

printf("Enter n:"); scanf("%lu", &n);

printf("Fibo number: %d\n", fibo(n));

\_getch();

}

unsigned long fibo(unsigned long number)

{

if(number==1) return 1; else if (number==0) return 0; else

return fibo(number-1) + fibo(number-2);

}

Аналогічно до попереднього випадку, порядок роботи рекурсивного механізму виклику функції можна зобразити схематично (див. рис.3).

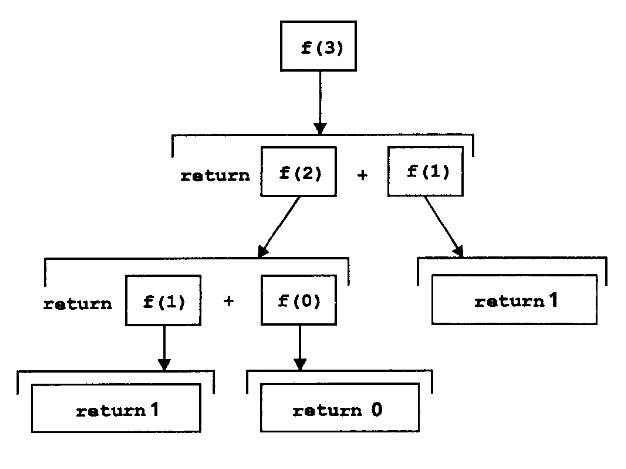


Рис.3. Порядок виконання рекурсивної функції обчислення чисел Фібоначі

Ще один характерний приклад застосування рекурсії - це швидке сортування елементів масиву (сортування Хоара). Основна ідея методу Хоара базується на принципі “розділяй і володій”. Для заданого масиву вибирається елемент (який отримав назву ”розділяючий елемент”), що розбиває масив на дві частини, які потім сортуються незалежно один від одного. Очевидно, що це можна зробити лише в тому випадку, коли розділяючий елемент буде знаходитися на своєму місці, тобто всі елементи масиву зліва від нього будуть не перевищути його, а всі елементи справа від нього- навпаки не менші за розділяючий елемент (звісно коли мова йде про сортування у порядку зростання). Тоді та ж процедура може бути рекурсивно застосовуватися й до двох отриманих підмножин елементів масиву до тих пір поки в обох підмножинах залишається більше двох елементів. У протилежному випадку рекурсивний алгоритм швидкого сортування завершується.

Ефективність алгоритму швидкого сортування в значній мірі залежить від правильного вибору розділяючого елемента. Очевидно, що для найкращого результату, у ролі розділяючого елементу слід вибирати медіану масиву (тобто той елемент, який у відсортованому масиві буде знаходитися посередині масиву). Однак пошук медіани масиву на кожному кроці рекурсії приведе до значних обчислювальних затрат, і як наслідок, до суттєвої втрати в швидкодії самого алгоритму (і тоді той алгоритм важко буде називати швидким). Тому на практиці, у ролі розділяючого елементу вибирають медіану між першим, останнім та середнім елементами масиву, що сортується. Обчислювальні затрати на таку операцію будуть завжди постійними і виражатимуться не більше ніж трьома операціями порівняння. Іншим можливим підходом є вибір випадкового елемента масиву у ролі розділяючого елементу.

Реалізація вище описаного варіанту алгоритму швидкого сортування елементів масиву у порядку зростання наведена нижче у вигляді двох функції: функції обміну двох елементів масиву swap і рекурсивної функції qs\_sort(int array[], long start, long end), яка сортує підмножину масиву задану індексами start та end.

void swap(int array[], long pos1, long pos2)

{ long tmp; tmp=array[pos1]; array[pos1]=array[pos2]; array[pos2]=tmp; }

void qs\_sort(int array[], long start, long end)

{

long head=start, tail=end-1, tmp;

long diff=end-start; long pe\_index;

// якщо залишилося менше двох елементів – кінець рекурсії

if (diff<1) return; if (diff==1)

if (array[start]>array[end]) { swap(array, start, end);

return; }

// пошук індексу розділяючого елементу pe\_index

long m=(start+end)/2; if (array[start]<=array[m]) { if (array[m]<=array[end]) pe\_index=m;

else if (array[end]<=array[start]) pe\_index=start;

else pe\_index=end;

} else {

if (array[start]<=array[end]) pe\_index=start; else if(array[end]<=array[m]) pe\_index=m;

else pe\_index=end; }

long pe=array[pe\_index]; // сам розділяючий елемент

swap(array,pe\_index,end);

while (1) {

while(array[head]<pe)

++head;

while(array[tail]>pe && tail>start) --tail;

if (head>=tail) break; swap(array,head++, tail--); }

swap(array,head,end); long mid=head;

qs\_sort(array, start, mid-1); // рекурсивний виклик для 1-ої підмножини qs\_sort(array, mid+1, end); // рекурсивний виклик для 2-ої підмножини

}

## Код програми

Назва файлу: **Lab7Task1.c**

#include <stdio.h>

#include <time.h>

#include <stdlib.h>

int arr1[100000], arr2[100000];

void swap(int array[], long pos1, long pos2)

{

    long tmp;

    tmp = array[pos1];

    array[pos1] = array[pos2];

    array[pos2] = tmp;

}

void qs\_sort(int array[], long start, long end)

{

    long head = start, tail = end - 1, tmp;

    long diff = end - start;

    long pe\_index;

    // якщо залишилося менше двох елементів – кінець рекурсії

    if (diff < 1) return;

    if (diff == 1)

        if (array[start] > array[end]) {

            swap(array, start, end);

            return;

        }

    // пошук індексу розділяючого елементу pe\_index

    long m = (start + end) / 2;

    if (array[start] <= array[m]) {

        if (array[m] <= array[end]) pe\_index = m;

        else if (array[end] <= array[start]) pe\_index = start;

        else pe\_index = end;

    }

    else {

        if (array[start] <= array[end]) pe\_index = start;

        else if (array[end] <= array[m]) pe\_index = m;

        else pe\_index = end;

    }

    long pe = array[pe\_index]; // сам розділяючий елемент

    swap(array, pe\_index, end);

    while (1) {

        while (array[head] < pe)

            ++head;

        while (array[tail] > pe && tail > start)

            --tail;

        if (head >= tail) break;

        swap(array, head++, tail--);

    }

    swap(array, head, end);

    long mid = head;

    qs\_sort(array, start, mid - 1); // рекурсивний виклик для 1-ої підмножини

    qs\_sort(array, mid + 1, end); // рекурсивний виклик для 2-ої підмножини

}

void choice\_sort(int array[], long start, long end)

{

    for (int i = start; i <= end; i++)

    {

        int minIndex = i;

        for (int j = i + 1; j <= end; j++)

        {

            if (array[j] < array[minIndex])

                minIndex = j;

        }

        swap(array, minIndex, i);

    }

}

void testSorting(int\* f1(int[], long, long), int\* f2(int[], long, long), double\* time1, double\* time2)

{

    srand(time(NULL));

    for (int i = 0; i < 100000; i++)

    {

        int element = rand();

        arr1[i] = element;

        arr2[i] = element;

    }

    for (int i = 0; i < 100000; i++)

    {

        if (arr1[i] != arr2[i])

        {

            printf("Arrays not identical before sorting!\n");

            return;

        }

    }

    clock\_t arr1\_before = clock();

    f1(arr1, 0, 99999);

    clock\_t arr1\_after = clock();

    clock\_t arr2\_before = clock();

    f2(arr2, 0, 99999);

    clock\_t arr2\_after = clock();

    for (int i = 0; i < 100000; i++)

    {

        if (arr1[i] != arr2[i])

        {

            printf("Arrays not identical after sorting!\n");

            return;

        }

    }

    \*time1 = (arr1\_after - arr1\_before)\*1000/ CLOCKS\_PER\_SEC;

    \*time2 = (arr2\_after - arr2\_before)\*1000/ CLOCKS\_PER\_SEC;

}

int main(void)

{

    double time1 = 0, time2 = 0;

    testSorting(qs\_sort, choice\_sort, &time1, &time2);

    printf("Quick sorting time is %lf milliseconds, choice sorting time is %lf milliseconds.", time1, time2);

}

Назва файлу: **Lab7Task2.c**

#include <stdio.h>

#include <math.h>

#include <stdbool.h>

bool compFloat(float x, float y)

{

    if (fabs(x - y) < 0.001)

        return 1;

    return 0;

}

void tabulateFunction\_for(float a, float b, float h)

{

    for (float x = a; x < b || compFloat(x,b); x += h)

    {

        if (x \* sqrt(x \* x - 1) == 0)

        {

            printf("Function doesn't exist with x = %f", x);

            return;

        }

        float y = 1 / (x \* sqrt(x \* x - 1));

        printf("x = %f, y = %f\n", x, y);

    }

}

void tabulateFunction\_while(float a, float b, float h)

{

    float x = a;

    while (x < b || compFloat(x,b))

    {

        if (x \* sqrt(x \* x - 1) == 0)

        {

            printf("Function doesn't exist with x = %f", x);

            return;

        }

        float y = 1 / (x \* sqrt(x \* x - 1));

        printf("x = %f, y = %f\n", x, y);

        x += h;

    }

}

void tabulateFunction\_doWhile(float a, float b, float h)

{

    float x = a;

    do

    {

        if (x \* sqrt(x \* x - 1) == 0)

        {

            printf("Function doesn't exist with x = %f", x);

            return;

        }

        float y = 1 / (x \* sqrt(x \* x - 1));

        printf("x = %f, y = %f\n", x, y);

        x += h;

    } while (x < b || compFloat(x, b));

}

double\* test\_function(double\* d)

{

    return d;

}

int main(void)

{

    int (\*func) (float, float, float);

    printf("Choose tabulation function:\n1 - for\n2 - while\n3 - do while\n");

    int choice;

    scanf\_s("%i", &choice);

    switch (choice)

    {

    case 1:

        func = tabulateFunction\_for;

        break;

    case 2:

        func = tabulateFunction\_while;

        break;

    case 3:

        func = tabulateFunction\_doWhile;

        break;

    default:

        printf("Choice incorrect!\n");

        return;

    }

    float a, b, h;

    printf("Enter a: ");

    scanf\_s("%f", &a);

    printf("Enter b: ");

    scanf\_s("%f", &b);

    printf("Enter h: ");

    scanf\_s("%f", &h);

    func(a, b, h);

}

Назва файлу: **Lab7Task3.c**

#include <stdio.h>

int max(int a, int b)

{

    if (a > b)

        return a;

    return b;

}

int maxArray(int arr[], int start, int end)

{

    if (end == start)

        return arr[start];

    return max(arr[start], maxArray(arr, start + 1, end));

}

int main(void)

{

    int arr[] = { 5,7,34,78,9 };

    printf("%i", maxArray(arr, 0, 4));

}

## Висновки

На лабораторній роботі розглянуто особливості роботи з рекурсивними функціями та вказівниками у мові C. Для завдань створені програми мовою C.