**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**

**НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ "ЛЬВІВСЬКА ПОЛІТЕХНІКА"**

Інститут **КНІТ**

Кафедра **ПЗ**

# ЗВІТ

До лабораторної роботи № 6

**На тему:** *“ОСОБЛИВОСТІ РОБОТИ З ФУНКЦІЯМИ В С.*

*ДИРЕКТИВИ ПРЕПРОЦЕСОРА.”*

**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 та основ роботи з препроцесором.

## 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.** Задано три матриці А,В,С цілих чисел розмірності *n*×*m* . Обчислити: ‖*A*‖+‖*B*‖+‖*C*‖

, де ‖⋅‖ - норма матриці, яка рівна сумі максимальних елементів

‖*A*+*B*+*C*‖

кожного рядка матриці.

**Завдання 2.** У функцію зі змінним числом параметрів надходять послідовності чисел Фібоначі, кінець списку – значення -1. Знайти і вивести перше натуральне число, яке більше за останнє число в цій послідовності.

1. **Завдання 3.** Використовуючи той самий ідентифікатор, оголосити числовий і символьний масиви. Для числового масиву знайти значення мінімального елемента. Для символьного масиву визначити символ, що найчастіше зустрічається. Використовувати умовну компіляцію.

## Теоретичні відомості

**ТЕОРЕТИЧНІ ВІДОМОСТІ**

**Передача масивів у функції**

У ролі аргументів функцій в С можуть використовуватися масиви. У цьому випадку достатньо вказати лише ім’я масиву без необхідності задавати квадратні дужки і розмірність масиву. Так, наприклад, якщо одновимірний масив array оголошений як int array[25];

то передати його у функцію сортування можна так: sort(array);

Прототип функції sort можна задати двома способами:

void sort(int []); або void sort(int \*);

С автоматично передає масиви у функції використовуючи механізм виклику функції з передачею адрес. Це означає, що у функцію передається лише адреса масиву (тобто адреса першого елемента масиву), а не його копія (пригадуємо, що у мові С ім’я масиву без індексу є вказівником на перший елемент масиву). Тому функції, яким передається масив, можуть змінювати значення елементів масиву, тобто будь-яка модифікація масиву в функції буде автоматично змінювати масив у точці виклику цієї функції. У тілі самої функції доступ до елементів масиву може здійснюватися або через стандартний механізм індексації, або через механізм вказівника. Наступний приклад програми демонструє ці можливості.

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

/\* передача масивів у функції та робота з ними всередині функції\*/

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

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

void sort(int [], int); // прототип функції з параметром типу масив void display(int \*, int); // прототип функції з параметром типу вказівник

void main()

{ const int n=10;

int array[n]={3,5,8,10,1,4,15,2,4,7}; printf("Array before sort: ");

display(array, n); // виклик функції з аргументом типу масив sort(array, n); // ще один виклик функції з аргументом типу масив printf("Array after sort: ");

display(array,n);

\_getch();

}

void display(int \*mas, int size)

{ // доступ до елементів масиву здійснюється через механізм вказівників for(int i=0; i<size; i++) printf("%d ", \*(mas+i)); printf("\n");

}

void sort(int mas[], int size)

{ // доступ до елементів масиву здійснюється через механізм індексації int k=1, temp;

for(int i=0; (i<size)&&(k); i++) { k=0;

for(int j=0; j<size-1; j++) if(mas[j]>mas[j+1]) { temp=mas[j]; mas[j]=mas[j+1]; mas[j+1]=temp; k=1;

}

}

}

Інколи виникають ситуації, коли потрібно заборонити модифікувати елементи масиву всередині функції. Оскільки масив передається за адресою, то контролювати самостійно всі зміни масиву в тілі функції може бути не простою задачею. Краще тут використати спеціальний специфікатор типу мови С, а саме специфікатор **const**. Коли параметр типу масив описується за допомогою *const*, то елементи такого масиву в тілі функції стають константами і будь-яка спроба їх модифікувати буде приводити до помилки (для прикладу спробуйте поміняти прототип та заголовок опису функції sort у попередній програмі на void sort(const int [], int); та void sort(const int mas[], int size), відповідно).

Для передачі двовимірних масивів у функції необхідно явно задавати розмірність другого виміру масиву. Це пов’язано з тим, що компілятор використовує це значення розмірності для правильного визначення комірок пам’яті елементів багатовимірного масиву, тобто компілятору потрібна інформація про те скільки елементів знаходиться в одному рядку двовимірного масиву для того, щоб правильно розбити матрицю на окремі рядки. Наступний приклад програми демонструє можливості передачі двовимірних масивів у функції з їх подальшою обробкою.

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

/\* передача 2D-масивів у функції та робота з ними всередині функції\*/

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

#include <stdio.h>

#include <conio.h>

#define n 3 #define m 4

void sort(int [], int); void display(int [][m], int);

void main()

{ int matr[n][m]={{3,5,8,10},{1,4,15,2}, {4,7,2,9}};

printf("Matrix before sort: \n"); display(matr, n); for (int i=0; i<n; i++) sort(matr[i], m);

printf("Matrix after sort: \n"); display(matr,n);

\_getch();

}

void display(int mas[][m], int size)

{ for(int i=0; i<size; i++) { for (int k=0; k<m; k++) printf("%d ", mas[i][k]);

printf("\n");

}

printf("\n");

}

void sort(int mas[], int size)

{ int k=1, temp;

for(int i=0; (i<size)&&(k); i++) { k=0;

for(int j=0; j<size-1; j++) if(mas[j]>mas[j+1]) { temp=mas[j]; mas[j]=mas[j+1]; mas[j+1]=temp;

k=1;

}

}

}

**Функції зі змінною кількістю параметрів**

У С крім функцій з *фіксованим* числом параметрів є можливість також використовувати функції зі *змінною* кількістю параметрів, під якими розуміють функції, в які можна передавати дані, не описуючи їх усіх у прототипі й, відповідно, у заголовку опису функції. Класичним прикладом є успадковані з “чистої” С функції введення/виведення scanf та printf. Список формальних параметрів для функцій із змінним числом параметрів складається з постійних параметрів та змінної частини, яка задається за допомогою конструкції еліпсису, що представляє собою три крапки … , причому спочатку перераховуються постійні параметри. У загальному випадку формат оголошення функції зі змінною кількістю параметрів має такий вигляд:

*[тип\_результату] ім’я\_функції(параметр1, параметр2, ...)***;** де поля *параметр1, параметр2* представляють постійну частину списку параметрів (хоча насправді тут може бути тільки один параметр або більше, ніж два параметри), а еліпсис … змінну частину, яка означає довільну кількість параметрів.

Для розуміння механізму роботи з функціями із змінним числом параметрів слід уважніше розглянути процес, який відбувається при виклику функції і передачі аргументів. Як відомо, передача аргументів у функції в С відбувається через спеціальну ділянку оперативної пам’яті, яка називається *стек*. У С/С++ фактичні параметри записуються в стек з кінця списку аргументів, причому ці дані поміщаються в стек відповідно до типу, який використовується при виклику функції. Оскільки принцип роботи зі стеком полягає в тому, що значення, яке було занесено до стеку останнім, буде використовуватися (витягуватися зі стеку) першим, то, наприклад, для такого виклику функції *f(x1,x2,x3)*, схематично вміст стеку можна зобразити так:

Diagram

Description automatically generated

Тепер стає зрозуміло, що маючи вказівник на початок (вершина стеку) і знаючи тип даних аргументів, ми можемо послідовно перебрати всі аргументи зі стеку (поки стек не стане порожнім), навіть не знаючи наперед їх кількість. Саме така організація передачі аргументів в С і робить можливим створення та використання функцій зі змінною кількістю параметрів.

Очевидно, що при передачі у функцію додаткових аргументів (тобто аргументів, що відповідають змінній частині списку формальних параметрів) відсутня інформація про їх тип даних на етапі компіляції програми. Тому контроль типів аргументів та параметрів, а також приведення аргументів до типу параметрів не відбувається. Уся відповідальність за правильну обробку типів таких аргументів покладається на програміста. ***Якщо передати функції зі змінним числом параметрів параметри не того типу, обробка якого запрограмована, наслідки будуть непередбачувані і, скоріше за все, сумні, адже компілятор у цьому випадку не перевірятиме ні кількість, ні типи параметрів***. Більше того, на програміста також покладається задача забезпечення доступу до додаткових аргументів, адже їхні імена відсутні. Видобути аргументи зі стеку у цьому випадку можна лише через вказівники. Причому тут існує два способи задання довжини змінного списку параметрів: в одному з постійних аргументів вказувати їх кількість або ж додати у кінець списку аргумент з унікальним значенням, що служитиме ознакою кінця списку. Наприклад: func1(5,xl,x2,x3,x4,x5); - тут зазначене число аргументів - 5; func2( xl,x2,x3,x4,0); - тут зазначена ознака кінця списку - 0.

У стандарт мови С входять макроси для роботи зі списками параметрів змінної довжини. Ці макроси визначені у файлі stdarg.h. При їх використанні також доводиться вказувати у списку постійний параметр, оголосити та встановити на нього вказівник та переміщувати цей вказівник по списку. В кінці списку має бути NULL. Макроси мають наступний формат:

**void va\_start(va\_list prm, останній\_фіксований\_параметр);**

**тип va\_arg(va\_list prm, тип); void va\_end(va\_list prm);**

Тип вказівника повинен бути va\_list, наприклад: va\_list p;

Макрос va\_start встановлює вказівник типу va\_list на останній фіксований параметр. Макрос va\_arg переміщає вказівник на наступний параметр. Синтаксис наступний: va\_arg(вказівник\_типу\_va\_list, тип\_чергового\_параметра). У момент написання програми програміст повинен знати тип кожного з параметрів. Макрос va\_end встановлює вказівник в NULL.

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

/\* Функція для обчислення добутку довільної кількості дійсних чисел \*/

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

#include <stdio.h>

#include <conio.h>

#include <stdarg.h> double multiply(char \*, ...); // прототип функції зі змінним числом параметрів

int main(int argc, char \* argv[])

{ multiply("4.0\*6.0\*2.0 = ",4.0,6.0,2.0,-1.0); multiply("2.0\*3.0\*5.0\*0.5 = ",2.0,3.0,5.0,0.5,-1.0);

multiply("This product is ",-1.0);

\_getch(); return 0;

}

double multiply(char \*str, ...)

{ double product = 1, term;

va\_list arg\_list; // оголошуємо вказівник типу va\_list va\_start(arg\_list,str);// встановлюємо вказівник на початок змінної

// частини списку параметрів

int k = 0;

while((term = va\_arg(arg\_list, double))>0)

{ product \*=term;

k = 1;

}

if(!k) product = 0;

printf("%s%lf\n", str, product);

va\_end(arg\_list); // завершуємо обробку списку параметрів return product;

}

Функція multiply повертає добуток довільної кількості дійсних чисел, заданих другим та наступними параметрами функції. Ця функція викликається в тілі функції main тричі: перший раз з трьома параметрами, другий – з чотирма, а третій – з одним обов’язковим параметром, що є рядком символів, та одним параметром, що служить ознакою кінця списку. Поточне значення добутку зберігається у змінній product. Початкове значення цієї змінної встановлене в 1, а на кожній ітерації циклу воно множиться на значення параметра функції дійсного типу, оброблюваного у поточний момент. Для випадку, коли функції multiply не передано параметрів дійсного типу, слід змінну product встановлювати в 0. Для цього застосовується змінна k, що служить ознакою – чи виконався цикл принаймні один раз (тобто, чи має функція принаймні один параметр дійсного типу).

Параметри обробляються за допомогою макросів, визначених у файлі stdarg.h, тому у програмі присутня директива #include <stdarg.h>. Єдиним обов’язковим параметром функції multiply є параметр str типу char \*. Тому для перебору параметрів функції multiply створюємо вказівник arg\_list спеціального типу va\_list та спочатку встановлюємо його на параметр str (va\_start(arg\_list,str);). У циклі переміщаємося по параметрах за допомогою макросу va\_arg: term = va\_arg(arg\_list,double) – кожен такий запис посуває вказівник на 1 позицію, причому враховується, що оброблювані параметри функції є типу double, а поточний параметр записується у заздалегідь оголошену змінну term типу double. Цикл продовжуватиметься, поки term > 0. При виклику функції multiply їй в якості останнього параметра передаємо значення –1 або якесь інше від’ємне значення чи значення 0. Як тільки буде досягнуто останній параметр функції, умова циклу while стане хибною і тіло циклу більше не виконається.

Для обробки параметрів функції зі змінною кількістю параметрів можна обійтися і без стандартних засобів, лише з використанням вказівників, як це показано у наступному прикладі.

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

/\*Функція обчислення середнього арифметичного довільної кількості дійсних чисел \*/

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

#include <stdio.h> #include <conio.h> double f(double , ...);// прототип функції зі змінним чисом параметрів

int main()

{ printf("Seredne aryfmetychne 2,4,3 = %lf\n", f(2.0,4.0,3.0,0.0));

printf("Seredne aryfmetychne 2,14,8,16 = %lf\n", f(2.0,14.0,8.0,16.0,0.0));

\_getch(); return 0;

}

double f(double n, ...)

{

double \*p = &n; // вказівник на початок списку параметрів double sum = 0; int count = 0;

while (\*p) // поки в списку є ненульовий елемент

{ sum+=(\*p);

p++; // переходимо на наступний параметр зі списку count++;

} return ((sum)?sum/count:0); }

Робота функції f базується на тому, що її останній параметр обов’язково має бути 0. Встановлюємо вказівник p на перший параметр функції n. Поки значення, що знаходиться за адресою, на яку посилається вказівник, не дорівнює нулю (тобто, до досягнення останнього елемента) поточний оброблюваний параметр функції додаємо до суми sum та збільшуємо лічильник count на 1. Функція f повертає sum/count або ж 0, якщо сума рівна 0. При цьому останній параметр (0) є виключно службовим – він не застосовується при розрахунку середнього арифметичного, оскільки при досягненні цього параметра умова циклу while стає хибною і тіло циклу більше не виконується.

Довжину змінної частини списку аргументів можна також задавати за допомогою фіксованого параметру, що демонструє такий приклад.

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

/\*Функція обчислення середнього арифметичного довільної кількості дійсних чисел \*/

/\* Варіант 2 \*/

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

#include <stdio.h>

#include <conio.h>

#include <stdarg.h> double f(int , ...);

int main()

{ printf("%lf\n", f(3,2.0,4.0,3.0)); printf("%lf\n", f(4,2.0,14.0,8.0,16.0));

printf("%lf\n", f(10,1.5,1.5,2.8,3.4,2.0,14.0,8.0,16.0,4.5,2.3));

\_getch(); return 0;

}

double f(int n, ...)

{

double sum=0, term; int count = 0;

va\_list p; va\_start(p, n); for (int i=0;i<n;i++)

{ term = va\_arg(p, double);

printf("term #%d = %lf\t", (i+1), term); sum+=term; count++;

} va\_end(p);

printf("%\nAverage = "); return ((sum)?sum/count:0); }

У даному випадку фіксований параметр n функції f містить число параметрів, які будуть фактично передані функції при виклику. Вказівник p типу va\_list спочатку вказує на єдиний обов’язковий параметр n функції f. У циклі з лічильником і за допомогою макросу va\_arg “перебираємо” усі параметри, записуючи поточний параметр у змінну term, причому враховуємо, що очікувані параметри мають тип double. Для наочності, щоб перевіряти кожен поточний параметр, виводимо його на екран. Змінну term додаємо у змінну sum та збільшуємо значення лічильника count. У тілі функції main функцію f викликаємо тричі – з 3, 4 та 10 параметрами дійсного типу. Результат показано на рис. 1:

Calendar

Description automatically generated

Рис. 1 Виклик функції з різною кількістю параметрів

**Аргументи функції main**

Мова С передбачає можливість передачі аргументів головній функції, запущеній на виконання програми – функції maіn() – за допомогою використання командного рядка. Відомо, що функція main() служить диспетчером, що організовує взаємодію між усіма іншими функціями у програмі. Зазвичай ця функція застосовується без параметрів, однак може викликатися з рядом параметрів, які передаються з командного рядка. Список формальних параметрів функції main() є фіксованим, однак за його допомогою можна задавати список аргументів, кількість яких обмежується лише можливостями операційної системи. Аргументи функції main() повинні вказуватися у командному рядку одразу після імені виконуваного модуля (файла з розширенням .exe). У ролі аргументів функції maіn() доцільно передавати імена файлів, функцій, текст, що задає режим роботи програми, а також самі дані (числа)

С підтримує три параметри функції maіn(). Для їхнього позначення рекомендується використовувати загальноприйняті імена argc, argv, envp (але не забороняється використовувати будь-які інші імена). Заголовок функції maіn() при використанні командного рядка має вигляд:

*іnt maіn(іnt argc, char \*argv[], char \*envp[])*

Перший параметр argcє цілим числом, що задає кількість аргументів, вказаних у командному рядку. Ім’я програми вважається першим аргументом. Тому у випадку, якщо користувач не задав жодного аргументу, значення argcстановитиме 1.

Параметр argv є вказівником на масив символьних вказівників, які посилаються на аргументи командного рядка. Слід мати на увазі, що всі аргументи командного рядка є рядками, а тому при потребі їх слід перетворювати у числа за допомогою відповідних функцій. Самі аргументи в командному рядку повинні розділятися пробілами або знаками табуляції, коми, крапки та інші символи не розглядаються як розділювачі. Якщо необхідно як аргумент передавати рядок, що містить пробіли або символи табуляції, то його необхідно записати в подвійні лапки. Останнім елементом масиву вказівників є нульовий вказівник (NULL).

Параметр env є масивом вказівників на рядки оточення операційної системи. Його реалізація залежить від конкретного середовища. Отримати вказівники на рядки оточення можна за допомогою функції getenv(), а задати їх – за допомогою функції putenv(). Обидві ці функції визначені у заголовному файлі stdlib.h.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ /\* Застосування аргументів функції main \*/

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

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

int main(int argc, char\* argv[])

{ printf("There are %d arguments\n", argc);

if ( !(argc==4) )

{ printf("Error - not enough or too many arguments\n");

\_getch(); return 1;

}

solvelin(atof(argv[2]), atof(argv[3]));

\_getch(); return 0;

}

int solvelin(double var1, double var2)

{

printf("The equation to be solved is:\t%lf\*x = %lf\n", var1, -var2);

if(var1) printf("Result: %lf", -var2/var1); else printf("There are no solutions");

return 0;

}

У даній програмі в функції main виводиться на екран кількість аргументів, введених з командного рядка. Якщо кількість аргументів не рівна 4, то виводиться повідомлення про те, що кількість аргументів або недостатня, або надмірна. У цьому випадку робота функції main припиняється оператором return. Інакше викликається користувацька функція solvelin, що приймає два параметри var1 і var2 типу double і служить для розв’язання лінійного рівняння var1\*x = –var2, за умови, що var1 ≠ 0 (інакше виводиться повідомлення про те, що розв’язків немає). Функції solvelin передаються параметри atof(argv[2]) та atof(argv[3]). Передбачається, що користувач з командного рядка вводить назву виконуваного модуля lab9.exe (це буде значення argv[0]), назву функції solvelin (це буде argv[1]) та представлення двох дійсних чисел (вони і визначатимуться значеннями argv[2] та argv[3]). Оскільки всі задані з командного рядка аргументи є рядками символів, то необхідне перетворення типів – argv[2] та argv[3] приводяться до дійсного типу за допомогою функції atof. На рис. 2 показане вікно програми cmd.exe (запущеної через Start=>Run), де викликається виконуваний модуль lab9.exe з різними аргументами. При цьому вказується повний шлях до файлу lab9.exe. Як видно з рис. 2, спочатку lab9.exe викликається з “правильною” кількістю аргументів – 4. Результатом є виклик функції solvelin. Далі lab9.exe викликається спочатку з недостатньою, а тоді з надмірною кількістю аргументів, про що видається відповідне повідомлення. Зрештою, lab9.exe викликається з чотирма аргументами, але argv[2] = 0 – розв’язку немає.

Text

Description automatically generated

Рис. 2 Виклик функції main з командного рядка з різними аргументами

Одним з критеріїв успішності виконання програми для оцінки операційною системою є код повернення програми (код помилки). Це ціле число, яке вказує на наявність певної проблеми або її відсутність при виконанні програми. Якщо функція main() описується з типом повернення int, то значення при операторі return в цій функції передаватиметься операційній системі як результат виконання програми (0 – успішний). Якщо ж функція main() описана з типом повернення void, то компілятор генерує код, який завжди вказуватиме операційній системі на успішне виконання програми.

**2.4. Препроцесор. Основні директиви препроцесора та компілятора.**

Першим етапом підготовки програми при перетворенні компілятором вихідного тексту у виконуваний код є обробка її *препроцесором* (або *макропроцесором*). Препроцесор обробляє текст програми і повертає його нову редакцію. У більшості систем програмування *препроцесор* сполучений з *транслятором*, і для програміста його робота прозора. Хоча при потребі препроцесор можна викликати окремо без компіляції тексту. Препроцесор виконує лише обробку тексту. Це означає, що він "не розуміє" операторів мови програмування, "не знає" про змінні, константи програми, а розрізняє тільки константи та змінні і виконує тільки оператори *макромови* – вирази у програмі, що адресовані макропроцесору. При цьому у проміжному тексті програми (результат роботи препроцесора) останні будуть відсутні і для подальших етапів обробки "не видні". Так, якщо препроцесор замінив у програмі деякий текст A на текст B, то транслятор уже бачить тільки текст B, і не знає, був цей текст написаний програмістом "своєю рукою" чи підставлений препроцесором.

*Директиви* (команди) препроцесора зазвичай використовуються для полегшення внесення змін до початкової програми, а також для полегшення її компіляції для різних середовищ виконання.

Всі директиви препроцесора в мові С починаються із символу #. Він повинен бути першим непорожнім символом в рядку (символи пробілу і табуляції ігноруються). Для зручності читання коду директиви препроцесора прийнято починати писати з першої колонки рядка.

Нижче наведено основні команди препроцесора та їх короткий опис.

#include #ifndef #endif

#define #if #error

#undef #else #pragma

#ifdef #elif

Директива #include має наступний формат:

#include "filename" або

#include <filename>

Використовується для вставлення усього вмісту деякого файлу з ім’ям (шляхом) filename у місце виклику директиви. Якщо ім’я файлу взяте в подвійні лапки, то пошук файлу відбувається в локальних папках проекту (задаються в параметрах компілятору, за замовчуванням – папка, в якій розміщений опрацьовуваний файл). Якщо ж ім’я файлу взяте в кутові дужки, препроцесор шукатиме його в папці для стандартної бібліотеки (залежить від налаштувань компілятора). Директива #include, як правило, використовується для підключення заголовкових файлів стандартної бібліотеки (таких як stdio.h, stdlib.h) та користувацьких заголовних файлів для великих програм, які складаються з декількох (багатьох) файлів, що компілюються разом в одну програму.

Директива #define має наступний формат:

#define IDENTIFIER text або

#define IDENTIFIER(parаms) text

Використовується для заміни часто вживаних в тексті програми констант, ключових слів, операторів або виразів деякими ідентифікаторами. В останніх двох випадках такі ідентифікатори називаються макровизначеннями або просто макросами. Макроси можуть приймати параметри. Формат макроса проілюстовано другим. Директива #define заміняє всі наступні входження ідентифікатора (в ілюстрації – IDENTIFIER) на вказаний текст (в ілюстрації – text). Текстом може бути будь-який фрагмент програми на мові С або пустий фрагмент (відсутність тексту як такого). Для задання багаторядкового тексту заміни можна використати символ \ вкінці кожного рядка, окрім останнього.

Директива #undef має наступний формат:

#undef IDENTIFIER

Використовується для скасування дії директиви #define для вказаного ідентифікатора. Після такого рядка всі наступні входження IDENTIFIER не будуть замінюватися.

Препроцесор С містить декілька вбудованих макроімен (ідентифікаторів), серед яких варто виділити такі:

\_\_LINE\_\_ – номер рядка у вихідному файлі (заміняється номером, де розміщений),

\_\_FILE\_\_ – ім’я опрацьовуваного файлу,

\_\_DATE\_\_ – дата початку опрацювання файлу препроцесором, \_\_TIME\_\_ – час початку процесу опрацювання.

Вбудовані макроімена не можна оголошувати в #define і скасовувати у #undef.

**Директиви умовної компіляції**

Директиви умовної компіляції використовуються для поділу тексту програми на частини, одні з яких будуть компілюватися, а інші пропускатимуться компілятором залежно від деяких умов.

Формат:

#if CONST\_EXPRESSION

#ifdef IDENTIFIER

#ifndef IDENTIFIER

Для #if обчислюється константний вираз CONST\_EXPRESSION, і якщо результат обчислень – істина (ненульовий), то компілюються всі рядки програми аж до директиви #else, #elif або #endif. Для #ifdef та #ifndef компілюється відповідна частина коду, якщо визначено (не визначено) ідентифікатор IDENTIFIER.

У константному виразі директиви #if може бути використана операція defined для перевірки, чи був визначений зазначений ідентифікатор. Обернену умову перевіряє операція ! defined. Ранні версії препроцесора не мають операції defined, а працюють тільки з #ifdef та #ifndef. Ці директиви були збережені для сумісності.

Вказані вище директиви можуть мати розділ "інакше", який визначається в одній з двох форм:

#else

#elif CONST\_EXPRESSION2

Для #else відповідна послідовність коду компілюється в тому випадку, якщо одна з вимог, вказана в #if не виконується.

#elif аналогічна #else за винятком того, що має виконуватися умова, яка задається константним виразом CONST\_EXPRESSION2.

Вказані вище директиви мають закінчуватися директивою #endif.

Директива #error зупиняє роботу компілятора і видає повідомлення про помилку.

Директива #line константа "ім'я файлу" змушує компілятор вважати, що константа задає номер наступного рядка вихідного файлу, і поточний вхідний файл іменується ідентифікатором. Якщо ідентифікатор відсутній, то ім'я файлу не змінюється.

Директива #pragma – це директива препроцесора, яка реалізує можливості компілятора. Ці особливості можуть бути пов'язані з типом компілятора. Різні типи компіляторів можуть підтримувати різні директиви. Загальний вигляд директиви:

#pragma команда компілятора

## Код програми

Назва файлу: **Lab6Task1.c**

#include <stdio.h>

double norm(int row, int col, int matrix[][100])

{

double sum = 0;

for (int i = 0; i < row; i++)

{

int max = matrix[i][0];

for (int j = 0; j < col; j++)

{

if (matrix[i][j] > max)

max = matrix[i][j];

}

sum += max;

}

return sum;

}

void addThreeMatrix(int m1[][100],int m2[][100], int m3[][100],int row, int col,int result[][100])

{

for (int i = 0; i < row; i++)

{

for (int j = 0; j < col; j++)

{

result[i][j] = m1[i][j] + m2[i][j] + m3[i][j];

}

}

}

int main(void)

{

int n, m;

int a[100][100], b[100][100], c[100][100], res[100][100];

printf("Enter n: ");

scanf\_s("%i", &n);

printf("Enter m: ");

scanf\_s("%i", &m);

for (int i = 0; i < n; i++)

{

for (int j = 0; j < m; j++)

{

printf("Enter element %i in row %i in matrix A: ", j, i);

scanf\_s("%i", &a[i][j]);

}

}

for (int i = 0; i < n; i++)

{

for (int j = 0; j < m; j++)

{

printf("Enter element %i in row %i in matrix B: ", j, i);

scanf\_s("%i", &b[i][j]);

}

}

for (int i = 0; i < n; i++)

{

for (int j = 0; j < m; j++)

{

printf("Enter element %i in row %i in matrix C: ", j, i);

scanf\_s("%i", &c[i][j]);

}

}

addThreeMatrix(a, b, c, n, m, res);

double answer = (norm(n, m, a) + norm(n, m, b) + norm(n, m, c)) / norm(n, m, res);

printf("\nAnswer: %lf", answer);

## }

Назва файлу: **Lab6Task2.c**

#include <stdio.h>

#include <stdarg.h>

int fibonacci(char c, ...)

{

va\_list arg\_list;

va\_start(arg\_list, c);

int num=va\_arg(arg\_list,int), res = -1;

while (num!=-1)

{

res = num + 1;

num = va\_arg(arg\_list, int);

}

return res;

}

int main()

{

printf("%i", fibonacci('a', 1, 1, 2, 3, 5, -1));

return 0;

}

Назва файлу: **Lab6Task3.c**

#include <stdio.h>

int main()

{

#define arr arrInt

int n;

int arr[100];

printf("Enter quantity of elements: ");

scanf("%i", &n);

for (int i = 0; i < n; i++)

{

printf("Enter element %i of int array: ", i);

scanf("%i", arr + i);

}

int min = arr[0];

for (int i = 0; i < n; i++)

{

if (min > arr[i])

min = arr[i];

}

printf("Min: %i\n", min);

#undef arr

char arr[100];

int cnt[100];

for (int i = 0; i < n; i++)

{

printf("Enter element %i of char array: ", i);

scanf(" %c", arr + i);

getchar();

cnt[i] = 1;

}

for (int i = 0; i < n; i++)

{

for (int j = 0; j < n; j++)

if (arr[i] == arr[j])

cnt[j]++;

}

int max = cnt[0], maxIndex = 0;

for (int i = 0; i < n; i++)

{

if (max < cnt[i])

{

max = cnt[i];

maxIndex = i;

}

}

printf("Most frequent character: %c", arr[maxIndex]);

}

## Висновки

На лабораторній роботі розглянуто особливості роботи з функціями у мові C. Окрім цього розглянуто директиви препроцесора. Для завдань створені програми мовою C.