Міністерство освіти і науки України

Національний університет “Львівська Політехніка”

Лабораторна робота №13-14

Виконав:
Студент групи АП-11
Білий Анатолій Іванович

Прийняв:

Чайковський І.Б.

Львів – 2024

“Структура функції. Локальні та глобальні змінні. Класи пам’яті”

Мета роботи: навчитися використовувати функції у процесі програмування, розуміти особливості використання локальних та глобальних змінних та специфікаторів різних класів пам’яті.

Теоретичні відомості

При програмуванні будь-яких задач, крім найпростіших, постійно виникає потреба виконувати в кількох різних місцях алгоритму одні й ті самі дії над різними значеннями. Наприклад, нехай потрібно обчислити площу трьох різних трикутників, перший з яких має сторони a_1, b_1, c_1, другий – a_2, b_2, c_2, третій – a_3, b_3, c_3. Це можна було б зробити в програмі наступним чином:

```
p1 = ( a1 + b1 + c1 ) / 2 ;
s1 = sqr t ( p1 * ( p1 - a1 ) * ( p1 - b1 ) * ( p1 - c1 ) ) ;
p2 = ( a2 + b2 + c2 ) / 2 ;
s2 = sqrt ( p2 * ( p2 - a2 ) * ( p2 - b2 ) * ( p2 - c2 ) ) ;
p3 = ( a3 + b3 + c3 ) / 2 ;
s3 = sqrt ( p3 * ( p3 - a3 ) * ( p3 - b 3 ) * ( p3 - c 3 ) ) ;
```

Зауважимо, що кожного разу писати одну й ту саму формулу незручно, оскільки при цьому витрачаються зусилля та час програміста на багатократне повторення, а не на творчу роботу. Збільшується текст програми, в якому стає важко орієнтуватись, що в свою чергу призводить до ще більших непродуктивних втрат часу.

Припустимо, що в першій з формул у програмі зроблено помилку (в процесі написання програми повністю уникнути помилок неможливо) і потім її скопійовано10 раз при повторних використаннях. Тоді і виправлення треба внести теж 10 раз. Це громіздка робота, оскільки потрібно продивлятися великий текст програми та вручну знаходити там всі випадки застосування цієї формули.

Отже, багаторазово описувати в програмі подібні між собою обчислення дуже незручно.

Потрібен засіб, який би дозволяв один раз і наперед описати деякий допоміжний алгоритм чи формулу, а потім по мірі потреби звертатися до нього з

основного алгоритму. Це дозволило б позбутися всіх вищеописаних труднощів. Для цього призначений потужний інструмент мови С, а саме функції.

Функцією в мові С є програмна одиниця, яка має ім 'я, деяку кількість аргументів визначених типів, містить послідовність операторів, що реалізують певний алгоритм, та може дати результат певного відомого типу.

Функції призначені для того, щоб з інших місць програми (тобто з інших функцій) їх можна було неодноразово викликати. Програма у мові С являє собою сукупність функцій, одна з яких функція main, з якої починається виконання програми (точка входу в програму) .

Для того, щоб мати змогу викликати функцію, транслятор повинен заздалегідь знати її ім'я,тип значення, кількість та типи аргументів. Ці характеристики разом складають прототип функції. Прототип нічого не каже про внутрішню будову функції, він описує лише її зовнішній вигляд (як до неї звертатися) .

Прототип – це оголошення функції, яке повинно бути відоме транслятору перед викликом функції.

Внутрішня будова функції, алгоритм, за яким вона працює та, зокрема, обчислює результат, визначається тілом функції. Нарешті, виклик функції складається з її імені та значень всіх аргументів.

В наведеному вище прикладі для обчислення площі трьох різних трикутників потрібна функція під назвою Geron (оскільки для обчислення площі використовується формула Герона), до якої можна звернутися, передавши їй три аргументи дійсного типу – довжини трьох сторін трикутника, і отримати від неї у відповідь дійсне число, значення площі цього трикутника. Наведемо приклад програмного тексту:

```
#include<stdio.h>
#include<math.h>
double Geron (double a,double b,double c);  /*оголошення функції- прототипу*/

double Geron (double a, double b, double c) { /*реалізація функції*/
        double p; /* локальна змінна */
        p= (a + b + c)/2;
        return sqrt(p*(p-a)*(p-b)*(p-c));/*обчислити вираз і повернути результат*/
}
int main ( ) {
```

```
        double u, v, w;
        double s;
        printf ("Vvedit storonu trikutnika");  /* у консолі введіть значення 2, 3, 4 */
        scanf ("%lf %lf %lf",&u, &v,&w);
        s=Geron(u,v,w);      /*викликається функція Geron. До аргументів a,b,c передаються значення  змінних u,v,w. Результат виклику присвоюється змінній s*/
        printf("Ploscha 1 trikutnika %lf\n",s);
        s=Geron (10.3,8.1,9.7); /* викликається функція Geron. До аргументів a,b,c передаються константи */
        printf("Ploscha 2 trikutnika %lf\n",s);
        s=Geron(u+10.3,v+w,w*1.7); /*  викликається функція Geron. До аргументів передаються значення виразів */
        printf("Ploscha 3 trikutnika %lf\n",s);
        return 0;
       }
```

Цей приклад ілюструє такі основні правила. Прототип функції закінчується крапкою з комою. Реалізація функції складається з заголовку, який повторює прототип, та тіла у фігурних дужках.

В тілі функції можуть оголошуватися локальні змінні – такі змінні доступні лише для цієї функції та існують лише під час її виконання. По оператору return робота функції завершується, а результатом виклику функції стає значення відповідного виразу. Три виклики функції Geron в головній функції ілюструють, що аргументи, які передаються при виклику, можуть бути значеннями змінних, константами, або взагалі значеннями довільних виразів. Значення, яке функція виробила за допомогою оператора return, може присвоюватися змінним.

Прототип починається з типу значення функції, потім йде ім'я функції, а далі в дужках перераховуються типи та (необов'язково) імена аргументів, розділені комою:

тип_знач ім ' я_функ (тип _ арг_1 , . . . , тип _ арг _ n) ;

Якщо програма невелика за обсягом та весь її текст розміщується в одному файлі, то зручніше за все всі прототипи зібрати на початку, одразу після директив #include та перед усіма реалізаціями функцій. Якщо програма складається з кількох модулів, прототипи треба розмістити у власноруч створених заголовочних файлах (такі файли мають розширення . h) та підключати їх директивою #include .

Нерідко виклик функції здійснюють не для того, щоб отримати від неї деяке обчислене значення, а для того, щоб виконати дії над пристроями. Така функція за своїм призначенням просто нічого не повинна повертати. Наприклад, функція, яка

приймає один аргумент, номер пункту меню, та друкує на екран текст розширеної підказки по даному пункту. Виникає питання,як позначити в прототипі функції відсутність значення, що повертається.

Нерідко також функція за своїм призначенням не повинна мати жодного аргументу (така функція, може брати потрібні їй дані з файлу, з клавіатури). Наприклад, функція, яка друкує на екран меню разом з запрошенням ввести команду, та повертає введений користувачем номер команди. Тоді виникає питання, як у прототипі позначити відсутність аргументів. Для цих цілей в мову С запроваджено спеціальний порожній тип void.

Тип – це множина допустимих значень (об'єктів даних) в поєднанні з сукупністю операцій, які над цими значеннями можна виконувати. Тип void виділяється тим, що в ньому немає жодного допустимого значення, та немає жодної операції.

Зрозуміло, що void не можна назвати справжнім типом, скоріше це умовний спосіб позначити відсутність будь-якого типу. Щоб позначити, що функція не повертає значення, треба написати, що вона повертає значення типу void. Точно таким же чином, щоб позначити, що функція не має аргументів, треба вказати у прототипі, що вона має один аргумент типу void ; також функцію без аргументів можна позначити парою дужок, між якими немає аргументів:

```
void ff ( int , int ) ; /* не повертає значення */
int gg ( void ) ; /* без аргументів */
void hh ( ) ; /* не повертає , без арг. */
```

Якщо функція повертає значення, то в її тілі повинен бути реалізований алгоритм, який обчислює це значення, та оператор, що має форму *return вираз*; , а якщо функція не повертає значення (має тип результату void), то оператор повернення скорочується до return ; .

Коли виконання доходить до оператору *return вираз*; , то він обчислює значення виразу і одразу припиняє виконання функції. Значення виразу передається в те місце програми, з якого функцію було викликано.

Оператор return може стояти в будь-якому місці функції. В більшості випадків його розміщують наприкінці тіла функції, але нерідко, якщо алгоритм функції доволі складний і має логічну структуру: якщо виконується умова A_1, повернути

значення e_1, а якщо виконується умова A_2, повернути значення e_2, то застосовується кілька операторів повернення в складі умовних операторів.

Локальні та глобальні змінні. Змінні, оголошені в тілі функції, називаються локальними. Поведінка та всі основні властивості аргументів функції такі ж, як і у локальних змінних. Локальні змінні цілком належать тій функції, в якій оголошені. Інші функції цих змінних не бачать, тобто не можуть жодним чиномдо них звернутися : ні взяти, ні присвоїти значення. Тому кажуть, що областю видимості локальної змінної є та функція, в якій вона оголошена.

Крім того, час життя локальної змінної є час, поки виконується функція, в якій вона оголошена. В той момент, коли програма входить у функцію, її локальні змінні створюються, тобто під них виділяється пам'ять. Локальні змінні існують у пам'яті,

Змінні , оголошені в двох різних функціях, навіть коли мають однакові імена, жодним чином не взаємодіють, не перекриваються між собою.

```
void ff ( ) {
  int x ;
  . . .
}
void gg ( ) {
  int x ;
  . . .
}
```

Змінна x з функції ff та змінна x з функції gg – це дві зовсім різні змінні, між якими немає нічого спільного.

Змінні в мові С можуть оголошуватися не лише всередині функції, а ще й за межами тіл функції (зазвичай перед усіма тілами функцій) . Такі змінні називаються глобальними, на відміну від локальних, їх час життя – весь час виконання програми, а область видимості – тіла всіх функцій у тому ж модулі, що стоять після оголошення даної змінної.

Іншими словами, глобальну змінну бачать всі функції, і всі можуть їй присвоювати значення. Розглянемо нижченаведену програму.

```
int x ; /* глобальна змінна */
void f ( ) ;
void g ( ) ;
void main ( ) {
  x = 0 ;
```

```
  g ( ) ;
  f ( ) ;
  g ( ) ;
}
void f ( ) {
  x = 8 ;
}
void g ( ) {
  printf ( " % d \ n " , x );
}
```

Три функції (main, f, g) мають спільний доступ до змінної x. Функція main спочатку присвоює змінній x значення 0, потім викликає функцію g, яка друкує значення змінної, тобто 0. Далі функція main викликає функцію f, яка тій же змінній присвоює значення 8. Потім функція g друкує значення змінної x, яке тепер дорівнює 8.

Отже, якщо немає виняткової потреби у глобальних змінних, то краще їх не використовувати. Справа в тому, що спільний доступ кількох функцій до глобальної змінної призводить до сильного зв'язку між функціями, надмірно тісної взаємодії. Як наслідок, навіть мала зміна в одній з таких функцій може відбитися на всіх інших, втрачається можливість різним програмістам працювати над різними функціями окремо. Натомість, взаємодію між функціями слід здійснювати через передачу аргументів та повернення значення.

Класи пам'яті. Однією з головних переваг мови С є те, що вона дає змогу керувати ресурсами програми. До таких ресурсів відноситься і пам'ять програм.

Кожна змінна в програмі належить до певного типу, який визначає скільки пам'яті необхідно виділити для її збереження. Крім того, кожна змінна належить до певного класу пам'яті, що визначає час її існування та область видимості. Час існування змінної - це період , протягом якого змінна існує в пам'яті, а область видимості (область дії) – це частина програми, в якій змінна може використовуватися.

В С визначено чотири специфікатори класу пам'яті:auto, register, extern і state.

Таблиця 1

Область дії та час існування змінних різних класів пам’яті

Клас пам’яті	**Ключове слово**	**Тривалість існування**	**Область дії**
Автоматичний	Auto	Тимчасово	Блок (локальна)
Регістровий	Register	Тимчасово	Блок (локальна)
Статичний	Static	Постійно	Блок (локальна)
Зовнішній статичний	Static	Постійно	Файл (один файл, глобальна)
Зовнішній	Extern	Постійно	Програма (всі файли, глобальна)

Клас пам'яті для функції завжди external. якщо перед її описом нестоїть специфікатор static. Клас пам`яті конкретної змінної залежить або від місця розташування її опису. або задається явно за допомогою спеціального специфікатору класу пам`яті. що розташовується перед описом функції. Усі змінні С можна віднести до одного з наступних класів пам`яті.

1) Auto. Автоматичні змінні, або ж **auto**, описуються всередині відповідної функції і ділянка їх дії лежить у межах даної функції, тобто змінна починає існування в момент активізації функції і зникає в момент завершення функції. Ще автоматичні змінні називають локальними. Спроба роботи з локальною змінною в інших функціях призводить до помилок. Оскільки за замовчуванням змінним призначається клас auto, то ключове слово auto можна не використовувати.

Два наведені нижче записи можна вважати еквівалентними.

float a, b;
auto float a, b;

Область видимості змінної автоматичного класу пам'яті починається з її визначення і завершується при виявленні кінця блоку, в якому ця змінна визначена. Доступ до таких змінних із зовнішнього блоку неможливий. Оскільки локальні змінні знищуються при виході з функції, в якій вони оголошені, то ці змінні не можуть зберігати значення між викликами функцій. Якщо не визначене місце для зберігання локальних змінних, то вони зберігаються в стеку. Пам'ять для автоматичних змінних відводиться динамічно під час виконання програми при вході

в блок, у якому описана відповідна змінна. При виході з блоку пам'ять, відведена під усі його автоматичні змінні, автоматично звільняється. Звідси й походить термін автоматичні змінні. Доступ до автоматичних змінних можливий лише з блоку, де змінні описані, оскільки до моменту входу в блок змінна взагалі не існує і пам'ять під неї не виділена.

2) Register. Регістрові змінні, або ж **register**, запам'ятовуються в регістрах центрального процесора, де доступ до них і робота з ними виконуються набагато швидше, ніж у пам'яті. В іншому регістрові змінні аналогічні автоматичним змінним.

При виділенні регістрової пам'яті запит програміста може бути не задоволений, оскільки регістри в поточний момент часу можуть бути недоступні. У цьому разі регістрова змінна стає простою автоматичною змінною.

Змінні регістрового класу пам'яті мають такі самі область видимості та час життя, як і автоматичні змінні. Не можна застосовувати register до глобальних змінних.

Приклад:

register int a = 5; /* а буде за можливістю розміщене в регістрі */

3) Static. Статичні змінні, або ж **static**, мають таку ж ділянку дії, як і автоматичні змінні, але вони не зникають, коли функція, що їх містить, закінчить свою роботу. Компілятор зберігає її значення від одного виклику функції до іншого. Локальна статична змінна при завершенні функції, в якій вона оголошена, не втрачає свого значення. Статична змінна залишається локальною для цієї функції. При черговому виклику функції вона зберігає своє колишнє значення.

```
int funct(void)
{
  static int value=20;
  ...
}
```

Статичні змінні можуть бути описані поза будь-якою функцією, тоді вони інтерпретуються як зовнішні змінні. Різниця між зовнішньою змінною та статичною зовнішньою змінною полягає в тому, що остання з них може бути використана тільки функціями того програмного файлу, в головній функції якого вона визначена, тобто доступна у власному файлі й більше ніде.

Час життя статичних змінних глобальний: починається після визначення змінної і триває до кінця програми. Область видимості статичних змінних залежатиме від того, чи є вони зовнішніми чи внутрішніми.

4) Extern. Зовнішні змінні, або ж **extern**, описуються поза функцією, але можуть бути описані і всередині функції — обов'язково зі специфікатором extern. Ділянкою дії змінних є всі функції програмного комплексу, тому ці змінні інакше називають глобальними.

Якщо змінна визначена перед основною функцією програмного комплексу, то в інших функціях, що входять у програмний файл, її можна не описувати як зовнішню, вона й так діятиме в них. До них можна отримати доступ у будь-якому виразі, незалежно від того, в якій функції знаходиться даний вираз.

Опис змінної як extern всередині функції потрібний у тих випадках, коли цю змінну необхідно використати, а вона визначена або в функції, яка активізується пізніше, або в іншому програмному файлі. Наприклад:

```
funk ()
{ extern double pi;
....
}
double pi = 3.14159;
void main ()
{...
}
```

Функції main () і funk () розташовані в одному початковому файлі, але визначення зовнішньої змінної pi знаходиться після функції funk (), у цьому разі потрібне оголошення зовнішньої змінної pi в функції funk зі специфікатором extern. Оголошення зовнішніх змінних інформує компілятор, що така змінна вже існує і пам'ять для неї вже виділена.

У процесі написання програмних комплексів необхідно сполучати різні види змінних за використанням пам'яті. Найекономніше витрачається пам'ять при роботі з автоматичними змінними. Тому треба прагнути там, де можна, використовувати цей клас пам'яті, а інші застосовувати в тих випадках, коли це необхідно, а саме:

- зовнішні змінні слід використовувати, коли потрібно організувати передачу великих обсягів даних між кількома функціями;

- статичні змінні в тому разі, коли необхідно неодноразово звертатися до функції і в процесі кожного звертання використовувати результати попереднього звертання;
- регістрові змінні використовувати з метою прискорення розрахунків.

Виклик функції. Розглянемо детальніше, як відбувається виклик функції. Нехай функція f має аргументи x_1, . . . ,x_k і локальні змінні a_1, . . . ,a_l. Функція g має аргументи y_1 , . . . , y_m, а також локальні змінні b_1, . . . ,b_n. Припустимо, для простоти, що типи всіх змінних, аргументів та значень цілі.

Нехай в тілі функції f є виклик функції g з виразами e_1 , . . . , e_m на місці аргументів.Такий виклик сам є виразом, зокрема може бути правою частиною оператору присвоювання або входити до складу інших виразів.

Нарешті, нехай у тілі функції g є оператор return e, де e – вираз.

```
int f ( int x _ 1 , . . . , int x _ k ) ;
int g ( int y _ 1 , . . . , int y _m ) ;
int f ( int x _ 1 , . . . , int x _ k ) {
   int a _ 1 , . . . , a _ l ;
   . . .
    g ( e _ 1 , . . . , e _ m ) . . .
   . . .
}
int g ( int y _ 1 , . . . , int y _m ) {
   int b _ 1 , . . . , b _ n ;
   . . .
   return e ;
}
```

Нехай виконується тіло функції f , і управління дійшло до виразу g (e_1 , . . . , e_m) . Тоді послідовність дій обчислювальної машини така:

1 . Значення виразів e_i обчислюються та запам'ятовуються;

2. Локальні змінні a_j та аргументи x_j функції f перестають бути видимими. Вони продовжують існувати в пам'яті машини, але, так би мовити, в замороженому стані;

3. Створюються локальні змінні b_j та аргументи y_j функції g – для них виділяється пам'ять;

4. Значення виразів, обчислені на кроці 1 , поміщаються у змінні y_j ;

5. Розпочинається виконання операторів, що складають тіло функції g;

6. Виконання доходить до оператору return e, значення виразу e обчислюється та запам'ятовується;

7. Знищуються локальні змінні b_j та аргументи y_j – пам'ять, яку вони займали, звільняється;

8. Локальні змінні a_j та аргументи x_j функції f, що були заморожені на кроці 2, відновлюються;

9. Управління повертається до функції f, до того самого місця, звідки викликано функцію g. При цьому значенням виразу g (e_1 , . . . , e_m) стає значення, яке було обчислено на кроці 6;

10. Продовжується виконання тіла функції f.

Для кращого розуміння даної послідовності кроків на рис. 1 представлено схему взаємодії цих кроків: з функції main викликається функція f , а з неї, в свою чергу, функція g.

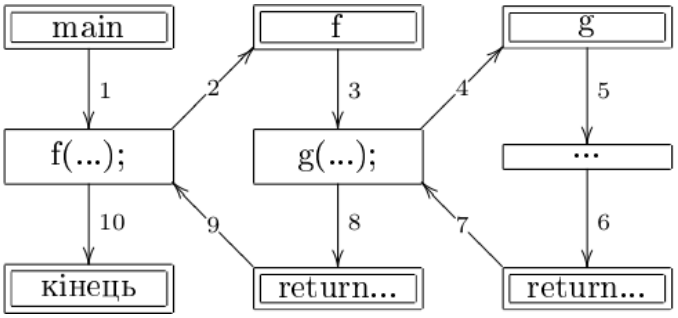

Рис. 1. Процес виконання програми з викликами функцій

Розглянемо процес виклику функцій на конкретному прикладі. Наведена нижче програма вводить з клавіатури два цілих числа, m та n, і друкує на екран значення: $\frac{(m+1+n+1)!}{(m+1)\cdot(n+1)}$. Введення вихідних значень та друк результату здійснюється у функції main, обчислення значення дробу $\frac{(x+y)!}{x\cdot y}$ (при x = m + 1, y = n + 1) винесено у функцію fract, яка, в свою чергу, викликає функцію factorial для обчислення факторіалу.

```
#include<stdio.h>
#include<math.h>
double fract(int,int);
double factorial(int);
int main( ) {
   int m,n,s;
```

```
    scanf ("%d %d",&m,&n);
    printf ("%lf\n",fract(m+1,n+1));
    return 0;
}
double fract(int x, int y) {
    double t;
    t=factorial(x+y)/(x*y);
    return t;
}
double factorial (int n) {
    int i;
    double p;
    p=1;
    for(i=1;i<=n;++i)
        p*=i;
    return p;
}
```

Опишемо детально кожен крок виконання програми, припустивши, що користувач вводить числа 1 та 3. На момент безпосередньо перед викликом функції fract в рядку з номером 7 машина має стан пам'яті: змінні функції main ~~m:1 n:3~~ .

Далі обчислюються значення виразів m+1 та n+1 , які стоять на місці аргументів, відповідно, числа 2 та 4. Локальні змінні m та n функції main приховуються, а натомість створюються змінні x, y, t. Обчислені перед цим значення 2 та 4 одразу присвоюються змінним x, y, а значення змінної t поки що невизначене (в ній може знаходитися будь-яке непередбачуване число, яке випадково опинилося в пам'яті на тому місці, куди потрапила змінна t).

Отже, маємо стан пам'яті: змінні функції fract x:2 y:4 t:- ; заморожені з функції main : m:1 n:3 .

Виконання тіла функції fract одразу ж призводить до виклику функції factorial. Діючи за наведеними правилами, маємо, що спочатку обчислюється та запам'ятовується значення виразу x+y, воно дорівнює 6. Змінні x, y, t функції fract приховуються, а створюються змінні n, i, p функції factorial. Значенням змінної n одразу ж стає обчислене число 6, значення двох останніх змінних поки що невизначені.

Стан пам'яті утворюється такий:змінні функції factorial n:6 i: - p: - ; заморожені функції з fract : x:2 y:4 t: - ; заморожені з функції main: m:1 n:3 .

Зверніть увагу, що в даному прикладі змінна (точніше, аргумент) n з функції main та змінна n з функції factorial – це дві зовсім різні змінні.

Виконання тіла функції factorial за кілька кроків призведе до такого стану пам'яті : змінні функції factorial : | n: 6 | i : 7 | p: 720 | ; заморожені з функції fract: | x:2 | y:4 | t: - |; заморожені з функції main: | m:1 | n:3 | .

У цьому стані виконується оператор return p. Обчислити значення виразу p означає просто взяти значення змінної, число 720. Локальні змінні та аргумент функції factorial знищуються, а змінні функції fract відновлюються. Продовжується перерване, або призупинене виконання тіла функції fract .

Виклик функції factorial стояв у функції fract у складі виразу factorial (x + y) / (x * y).

Значенням підвиразу factorial (x + y) стає число 720, отже значенням усього виразу є число $\frac{720}{2 \cdot 4} = 90$, яке й присвоюється змінній t. В результаті виходить стан пам'яті: змінні функції fract | x:2 | y: 4 | t: 90 |; заморожені з функції main: | m:1 | n:3 |.

У цьому стані виконується оператор return t, отже функція fract повертає значення 90, яке (після знищення локальних змінних функції fract, відновлення змінних та продовження виконання функції main) передається як аргумент до функції printf та друкується на екран.

Рекурсія. Логічна структура програми в мові С зводиться до того, як функції викликають одна одну. Кожна функція відповідає певній підзадачі, тому виклик функції g з тіла функції f означає, що для вирішення задачі f потрібно вирішити допоміжну підзадачу g.

Рекурсивною називають таку функцію, яка викликає сама себе. Це означає, що для вирішення деякої 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 функція f безпосередньо викликає сама себе, а непряма – в тому, що функція f_1 викликає деяку функцію f_2, та в свою чергу викликає функцію f_3 і так далі, нарешті функція f_n викликає функцію f_1.

Рекурсивні функції добре підходять для реалізації методу математичної індукції та для програмного моделювання математичних понять, які мають рекурентні означення. Спрощено, рекурентним означенням деякого поняття А називають таке означення, в якому використовується саме це поняття А.

Простим прикладом рекурсивної функції є factr (), яка обчислює факторіал цілого невід'ємного числа. Факторіалом числа n (позначається n!) називається добуток всіх цілих чисел, від 1 до n включно (для 0, за визначенням, факторіал дорівнює 1.). Наприклад, 3! - це 1 × 2 × 3, або 6.

Недоліками рекурсивних функцій в мові С є порівняно великі затрати пам'яті та часу. Мова С більше пристосована до операторів циклу. Хоча для багатьох задач рекурсивне рішення виглядає настільки елегантнішим та простішим, що ці додаткові витрати цілком виправдані.

Хід роботи

1. Ознайомитися з теоретичними відомостями.

2. Здійснити виконання прикладів, представлених у теоретичних відомостях, після чого представити скріни їх коду та результати виконання у звіті.

3. Написати програму з використанням функції, яка друкує визначену кількість символів рядка. Уточнення: дана функція повинна приймати рядок символів і ціле число, яке визначатиме кількість символів, що слід надрукувати. Скрін коду програми та результати її виконання представити у звіті.

4. Оформити звіт.

Приклад 1

```
#include<stdio.h>
#include<math.h>
double Geron (double a,double b,double c);
double Geron (double a, double b, double c) {
double p;
p= (a + b + c)/2;
return sqrt(p*(p-a)*(p-b)*(p-c));
}
int main ( )
{ double u, v, w;
double s;
printf ("Vvedit storonu trikutnika");
scanf ("%lf %lf %lf",&u, &v,&w);
s=Geron(u,v,w);
```

```
printf("Ploscha 1 trikutnika %lf\n",s);
s=Geron (10.3,8.1,9.7);
printf("Ploscha 2 trikutnika %lf\n",s);
s=Geron(u+10.3,v+w,w*1.7);
printf("Ploscha 3 trikutnika %lf\n",s);
return 0;
}
```

```
Vvedit storonu trikutnika 15 15 15
Ploscha 1 trikutnika 97.427858
Ploscha 2 trikutnika 36.928095
Ploscha 3 trikutnika 307.460239
```

Приклад 2

```
#include<stdio.h>
#include<math.h>
double fract(int,int);
double factorial(int);
int main( ) {
 int m,n,s;
 scanf ("%d %d",&m,&n);
 printf ("%lf\n",fract(m+1,n+1));
 return 0;}
double fract(int x, int y) {
 double t;
 t=factorial(x+y)/(x*y);
 return t;}
double factorial (int n) {
 int i;
 double p;
 p=1;
 for(i=1;i<=n;++i)
 p*=i;
 return p;}
```

```
1 2
20.000000
```

Приклад 3

```
#include <stdio.h>
#include <windows.h>
int func(int x);
char str[50];
void main(){
	SetConsoleCP(65001);
	SetConsoleOutputCP(65001);
	int a = 0;
	printf("Введіть строку: ");
	scanf("%s",&str);
```

```
    printf("Введіть скільки символів повинно вивести: ");
    scanf("%d",&a);
    func(a);
}
int func(int x){
    for(int i=0;i<x;i++){
        printf("%c",str[i]);
    }
}
```

```
Введіть строку: Programmer
Введіть скільки символів повинно вивести: 3
Pro
```

Контрольні питання

1. Дайте визначення поняття функція в мові С.
   Функція в мові С - це фрагмент програмного коду, який виконує певну задачу і може бути викликаний з інших частин програми. Вона може приймати аргументи, виконувати певні операції та повертати значення.

2. Поясніть призначення типу void.
   Тип void у мові програмування С використовується для вказівки на те, що функція не повертає жодного значення. Він може бути використаний у випадках, коли функція виконує певні операції, але не має значення, яке потрібно повернути.

3. Чим локальні змінні відрізняються від глобальних?
   Локальні змінні обмежені областю видимості в межах блоку коду, у якому вони визначені. Вони доступні лише всередині цього блоку коду та недоступні ззовні. Глобальні змінні, натомість, визначаються на рівні файлу і можуть бути доступні з будь-якого місця в програмі.

4. Яку функцію називають рекурсивною?
   Рекурсивною функцією називають таку функцію, яка викликає саму себе протягом свого виконання. Це потужний метод для розв'язання певних задач, таких як обхід дерева або обчислення факторіалу, але він також може призвести до переповнення стеку, якщо не використовувати умови виходу.

5. Назвіть переваги і недоліки використання рекурсивної функції.
   Переваги рекурсивних функцій:
   - Зручність та простота реалізації деяких алгоритмів, які можуть бути виражені рекурсивно.
   - Деякі задачі можуть бути більш чітко виражені за допомогою рекурсії.
   Недоліки рекурсивних функцій:
   - Може призвести до переповнення стеку пам'яті при надмірній глибині рекурсії.
   - Рекурсивні виклики можуть бути менш ефективними за ітеративні рішення через додаткові витрати на кожний виклик функції.
   Назвіть специфікатори класів пам’яті, які використовуються у мові С.
6. У мові С використовуються такі специфікатори класів пам'яті:

- auto: Змінна, що зберігається в стеку та автоматично ініціалізується при кожному входженні в блок.
- register: Підказка компілятору для збереження змінної у регістрах процесора, якщо це можливо.
- static: Змінна, яка зберігається в області даних, а не на стеку, і існує протягом усього життєвого циклу програми.
- extern: Вказує на те, що змінна визначена в іншому файлі програми.

Висновок: Під час виконання даної лабораторної роботи я навчився використовувати функції у процесі програмування, зрозумів особливості використання локальних та глобальних змінних та специфікаторів різних класів пам’яті.