



ОРГАНІЗАЦІЯ КОДУ ЗА ДОПОМОГОЮ ФУНКЦІЙ

Питання 3.4.

Поняття функції

- Функція — це іменований фрагмент коду, відділений від інших (Любанович).
 - Функція – це самодостатня одиниця коду програми, спроектована для виконання окремої задачі (Прага)
- З функцією можна зробити 2 речі:
 - означити (*function definition*);
 - викликати (*function call*).
- Імена функцій відповідають тим же правилам, що й назви змінних.
 - Повинні починатись з букви або _ та містити тільки букви, цифри або _
- Функції усувають потребу багатократного написання однакового коду.
 - Навіть функція з одноразовим кодом робить програму більш модульною, покращуючи її читабельність та спрощуючи внесення змін чи виправлень.

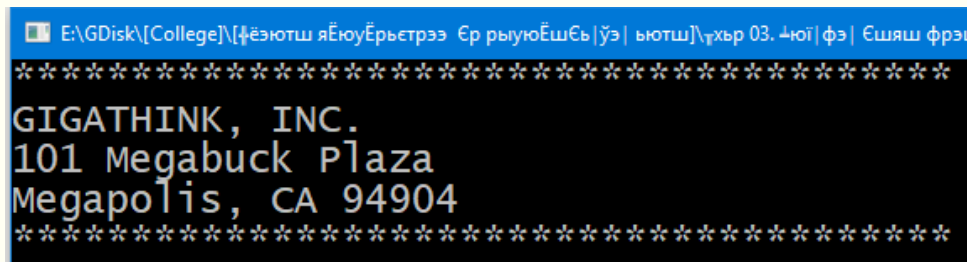
Приклад функції в мові С

```

1  /* lethead1.c */
2  #include <stdio.h>
3  #define NAME "GIGATHINK, INC."
4  #define ADDRESS "101 Megabuck Plaza"
5  #define PLACE "Megapolis, CA 94904"
6  #define WIDTH 40
7
8  void starbar(void); /* prototype the function */
9
10 int main(void)
11 {
12     starbar();
13     printf("%s\n", NAME);
14     printf("%s\n", ADDRESS);
15     printf("%s\n", PLACE);
16     starbar(); /* use the function */
17
18     return 0;
19 }
20
21 void starbar(void) /* define the function */
22 {
23     int count;
24
25     for (count = 1; count <= WIDTH; count++)
26         putchar('*');
27     putchar('\n');
28 }

```

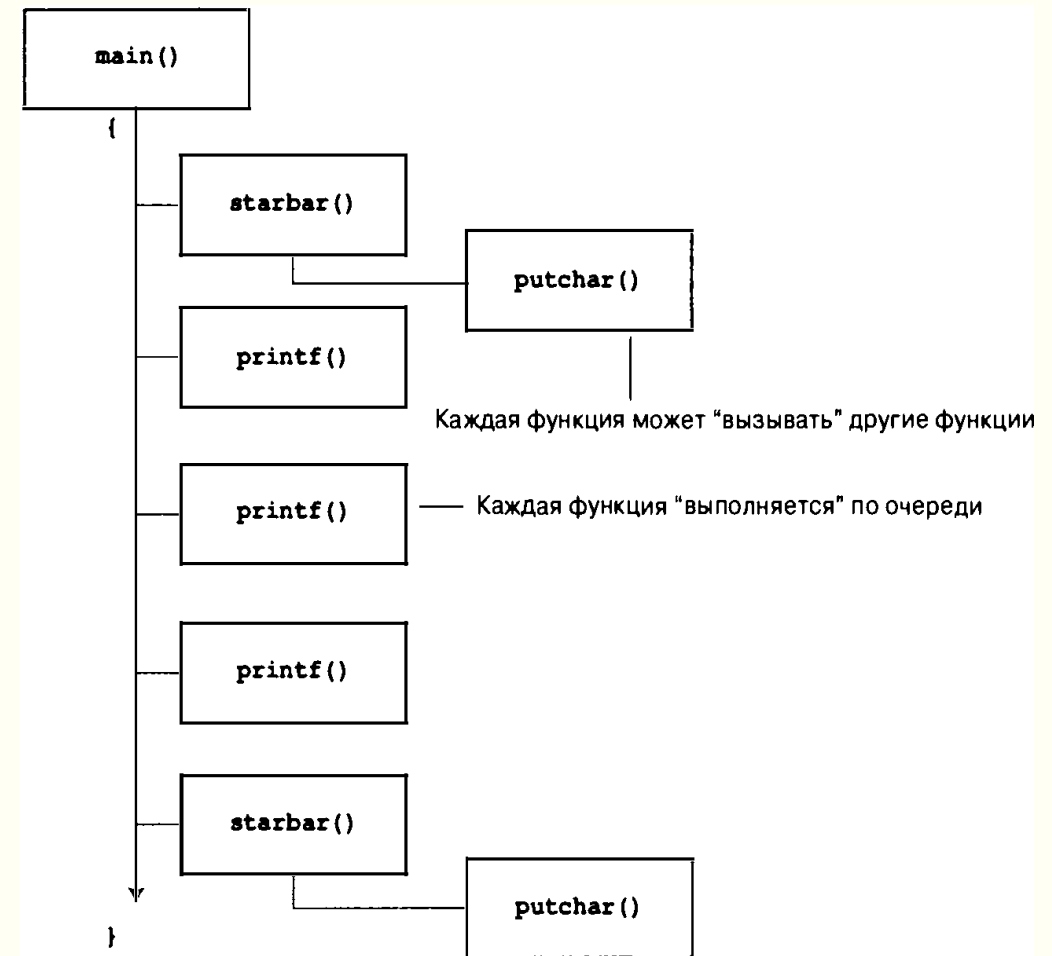
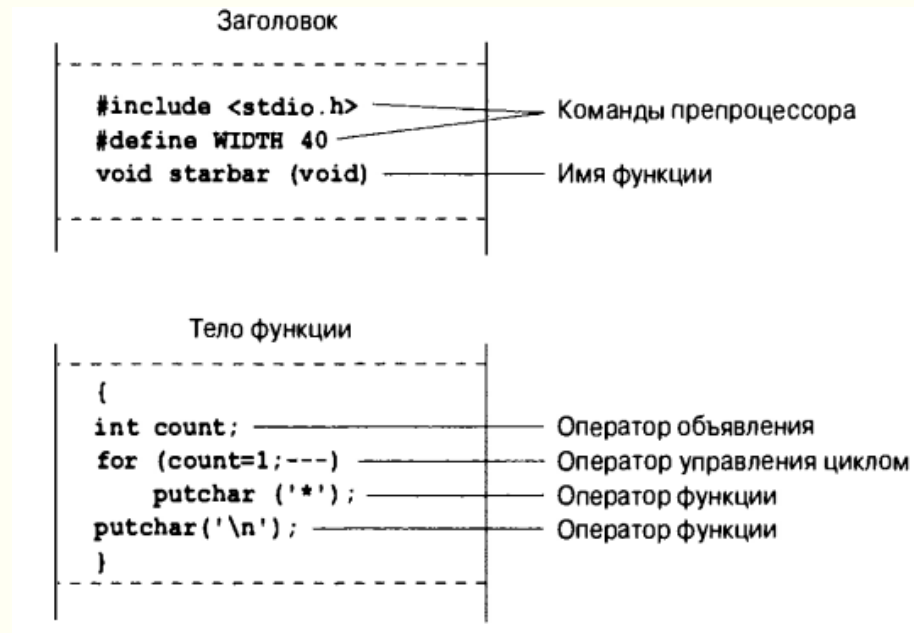
- Ідентифікатор `starbar` застосовується в 3 окремих контекстах:
 - У *прототипі функції*, який повідомляє компілятору різновид функції `starbar()` – оголошує її тип
 - У *виклику функції*, який приводить до виконання функції,
 - В *означенні функції*, де точно вказано все, що робить функція.
- Результати виводу:



lethead1.c

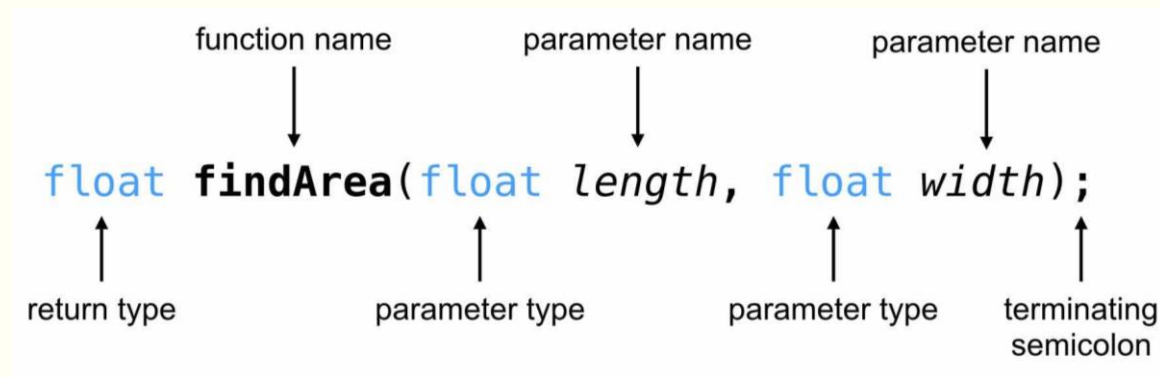
Аналіз потоку виконання програми

- Прототип вказує як тип значення, що повертається функцією, так і типи очікуваних нею аргументів.
 - Узагальнено цю інформацію називають *сигнатурою функції*



Функції в мові C

- Приймає будь-яку кількість параметрів та повертає один будь-який результат.



- Прототип: `void show_n_char(char ch, int num)`
 - Змінні `ch` та `num` називають *формальними параметрами*.
 - формальні параметри є локальними змінними для функції.
 - форма ANSI C вимагає, щоб **кожній змінній** передував її тип.
- Функції повинні оголошуватись із вказуванням типів.
 - Тип значення, яке функція повертає, повинен співпадати з оголошеним типом повернення (return type).
 - Функція без типу повернення повинна оголошуватись з типом `void`.

```

1  /* lethead2.c */
2  #include <stdio.h>
3  #include <string.h>          /* for strlen() */
4  #define NAME "GIGATHINK, INC."
5  #define ADDRESS "101 Megabuck Plaza"
6  #define PLACE "Megapolis, CA 94904"
7  #define WIDTH 40
8  #define SPACE ' '
9
10 void show_n_char(char ch, int num);
11
12 int main(void)
13 {
14     int spaces;
15
16     show_n_char('*', WIDTH);  /* using constants as arguments */
17     putchar('\n');
18     show_n_char(SPACE, 12);   /* using constants as arguments */
19     printf("%s\n", NAME);
20     spaces = (WIDTH - strlen(ADDRESS)) / 2;
21     /* Let the program calculate */
22     /* how many spaces to skip */
23     show_n_char(SPACE, spaces); /* use a variable as argument */
24     printf("%s\n", ADDRESS);
25     show_n_char(SPACE, (WIDTH - strlen(PLACE)) / 2);
26     /* an expression as argument */
27     printf("%s\n", PLACE);
28     show_n_char('*', WIDTH);
29     putchar('\n');
30
31     return 0;
32 }

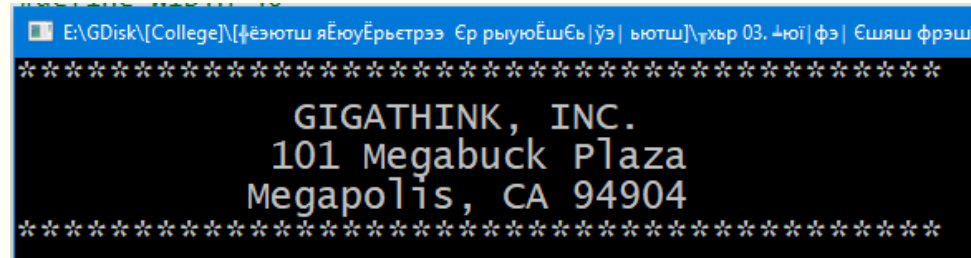
```

Приклад програми з 2 функціями (початок)

- Прототип функції (рядок 10) потрібно оголошувати до її застосування.
 - Він відображає кількість аргументів та їх типи.
 - За бажання їх назви можна не вказувати:
 - `void show_n_char(char, int);`
- Значення `ch` та `num` присвоюються за допомогою **фактичних аргументів** у виклику функції (рядки 16, 18, 23, 25, 28).
 - Для рядку 18 фактичні аргументи – `SPACE` та `12`.

Продовження коду та вивід програми

```
34 /* show_n_char() definition */
35 void show_n_char(char ch, int num)
36 {
37     int count;
38
39     for (count = 1; count <= num; count++)
40         putchar(ch);
41 }
```



```
E:\GDisk\College\Мієзютш яЁюуЁрьстрээ Ёр рыуюЁшёь|ўэ| ьютш]\_ґхьр 03. 4юї|фэ| Ёшяш фрэші
*****
GIGATHINK, INC.
101 Megabuck Plaza
Megapolis, CA 94904
*****
```



lethead2.c

```
int main(void)
{
    ...
    space(25);
    ...
}
```

Фактический аргумент — это значение 25,
переданное main() в функцию space()
и присвоенное number

Формальный параметр — это имя,
созданное определением функции

```
...
...
void space (int number)
{
    ...
}
```

Область видимості (scope)

- Описує частини програми, з яких можна отримати доступ до конкретного ідентифікатора.
- Для змінної в мові C характерна одна з наступних областей видимості:
 - в межах блоку,
 - в межах прототипу функції
 - в межах файлу.
- Визначена всередині блоку змінна має область видимості в межах блоку.
 - Її видно від точки визначення до кінця відповідного блоку.
 - Формальні параметри функції, які фактично знаходяться за межами блоку, мають в якості області видимості весь блок та належать блоку, що містить тіло функції.

Області видимості

```
double blocky(double cleo)
{
    double patrick = 0.0;
    ...
    return patrick;
}
```

■ Змінні `cleo` та `patrick` мають блок у якості області видимості.

- За традицією такі змінні оголошуються на початку блоку.
- Стандарт C99 дозволяє оголошувати змінні в будь-якому місці блоку.

```
for (int i = 0; i < 10; i++)
    printf("Возможность C99: i = %d", i);
```

```
double blocky(double cleo)
{
    double patrick = 0.0;
    int i;
    for (i = 0; i < 10; i++)
    {
        double q = cleo * i;    // начало области видимости переменной q
        ...
        patrick *= q;          // конец области видимости переменной q
    }
    ...
    return patrick;
}
```

Області видимості

- Область видимості в межах прототипу функції застосовується до назв змінних, які використовуються в прототипах функцій:
 - `int mighty(int mouse , double large);`
 - Область видимості в межах прототипу функції знаходиться від точки оголошення змінної до кінця оголошення прототипу.
- Змінна з визначенням за межами будь-якої функції має область видимості в межах файлу.
 - Така змінна видима від точки її оголошення до кінця файлу, який містить це оголошення.

```
#include <stdio.h>
int units = 0; /* переменная с областью видимости в пределах файла */
void critic(void);
int main(void)
{

}

void critic(void)
{

}
```

Оператор `return` та повернення значення з функції

- Перериває виконання функції та повертає управління викликаючій функції.
 - Ключове слово `return` призводить до того, що значення виразу після нього стає значенням, яке повертає функція

Режими передачі параметрів у функцію

▪ Call by value (передача за значенням)

- Зазвичай при передачі аргументу в функцію його значення копіюється у її відповідний формальний параметр.

```
int successor(int n) { return n+1; }
```

```
r = successor(5);
```

▪ Call by reference (передача за посиланням)

- Якщо тип формального параметру має &, такий параметр називають посилальним (*reference parameter*)
- При виклику функції потрібно передавати значення змінної, а функція отримає адресу цієї змінної.
- Функція напряду оперує змінною через адресу.

```
void successor(int& n) { n = n+1; }
```

```
int w = 5;  
successor(w);
```

▪ Call by pointer (передача за вказівником)

- Оскільки при передачі за посиланням передається адреса пам'яті, насправді це буде вказівник.
- Можна явно передавати вказівники та працювати з ними.

```
void successor(int* n) { *n = *n+1; }
```

```
int w = 5;  
successor(&w);
```

Взаємодія функцій з масивами

- Нехай потрібна функція, яка повертає суму елементів масиву.
 - Нехай `marbles` — назва масиву значень типу `int`.
 - `total = sum(marbles);` // можливий виклик функції
 - `int sum(int * ar);` // відповідний прототип
- Перший варіант реалізації:

```
int sum(int * ar)          // соответствующее определение
{
    int i;
    int total = 0;
    for( i = 0; i < 10; i++) // предполагается наличие 10 элементов
        total += ar[i];     // ar[i] – то же самое, что и *(ar + i)
    return total;
}
```

Взаємодія функцій з масивами

- Гнучкіший підхід передбачає передачу в другому аргументі розміру масиву:

```
int sum(int * ar, int n)    // более общий подход
{
    int i;
    int total = 0;
    for( i = 0; i < n; i++) // используются n элементов
        total += ar[i];    // ar[i] – то же самое, что и *(ar + i)
    return total;
}
```

- У контексті прототипу або заголовку визначення функції (і тільки в цьому контексті) замість `int * ar` можна підставити `int ar[]`:
 - `int sum (int ar[] , int n);`
 - Форма `int * ar` завжди означає, що `ar` є типом вказівника на `int`.
 - Форма `int ar[]` також означає, що `ar` — тип вказівника на `int`, проте лише тоді, коли він застосовується для оголошення формальних параметрів.

Взаємодія функцій з масивами

- Оскільки в прототипах дозволено опускати ім'я, всі нижче наведені прототипи еквівалентні:
 - `int sum(int *ar, int n) ;`
 - `int sum(int *, int);`
 - `int sum(int ar[], int n) ;`
 - `int sum(int [], int);`
- У визначеннях функцій імена опускати не можна, тому наступні 2 форми визначення еквівалентні:
 - `int sum(int *ar, int n) {...}`
 - `int sum(int ar [], int n) {...}`

Взаємодія функцій з масивами

```
1 // use %u or %lu if %zd doesn't work
2 #include <stdio.h>
3 #define SIZE 10
4 int sum(int ar[], int n);
5 int main(void)
6 {
7     int marbles[SIZE] = {20,10,5,39,4,16,19,26,31,20};
8     long answer;
9
10    answer = sum(marbles, SIZE);
11    printf("The total number of marbles is %ld.\n", answer);
12    printf("The size of marbles is %zd bytes.\n",
13           sizeof marbles);
14
15    return 0;
16 }
17
18 int sum(int ar[], int n)    // how big an array?
19 {
20     int i;
21     int total = 0;
22
23     for( i = 0; i < n; i++)
24         total += ar[i];
25     printf("The size of ar is %zd bytes.\n", sizeof ar);
26
27     return total;
28 }
```

The size of ar is 8 bytes.
The total number of marbles is 190.
The size of marbles is 40 bytes.

- Якщо ваш компілятор не підтримує специфікатор `%zd`, для виводу значень функції `sizeof` використовуйте специфікатор `%u` або, можливо, `%lu`.



sum_arr1.c

Використання параметрів типу вказівників

```
1 #include <stdio.h>
2 #define SIZE 10
3 int sump(int * start, int * end);
4 int main(void)
5 {
6     int marbles[SIZE] = {20,10,5,39,4,16,19,26,31,20};
7     long answer;
8
9     answer = sump(marbles, marbles + SIZE);
10    printf("The total number of marbles is %ld.\n", answer);
11
12    return 0;
13 }
14
15 /* use pointer arithmetic */
16 int sump(int * start, int * end)
17 {
18     int total = 0;
19
20     while (start < end)
21     {
22         total += *start; // add value to total
23         start++;         // advance pointer to next element
24     }
25
26     return total;
27 }
```

E:\GDisk\[College]\[фёзютш яЁюуЁрьстрээ Ёр рыуюЁшёь|ўэ| ьютш]_хьр 03. 4юї|фэ|

The total number of marbles is 190.

- Інший спосіб опису масиву передбачає передачу функції 2 вказівників (початок і кінець масиву).
 - Вираз `total += *start` додає до `total` значення першого елемента (20).
 - Потім вираз `start++` вказує на наступний елемент в масиві.

Коментарі до коду

- Для перевірки завершення циклу застосовується другий вказівник:
 - `while (start < end)`
 - За такою умовою `end` насправді вказує на наступну після кінцевого елемента масиву комірку.
 - Мова C гарантує, що вказівник на першу комірку після кінця масиву буде допустимим.
 - Тоді маємо лаконічний виклик функції: `answer = sump(marbles, marbles + SIZE);`
 - Оскільки індексація починається з 0, то `marbles + SIZE` вказує на елемент після кінця масиву.
 - Якби `end` вказував на останній елемент масиву, довелось би писати:
`answer = sump (marbles, marbles + SIZE - 1);`
- Тіло циклу можна стиснути до одного рядка: `total += *start++;`
 - Унарні операції `*` та `++` мають однаковий пріоритет, проте асоціацію справа наліво.
 - Тобто операція `++` застосовується до `start`, а не до `*start`.

Функції та багатовимірні масиви

- Одна з можливостей: використати цикл for для застосування функції обробки одновимірних масивів до кожного рядка двовимірного масива.

```
int junk[3][4] = { {2,4,5,8}, {3,5,6,9}, {12,10,8,6} };
int i, j;
int total = 0;
for (i = 0; i < 3 ; i++)
    total += sum(junk[i], 4); // junk[i] - одномерный массив
```

- Проте втрачається можливість відстеження інформації про рядки та стовпці.
- Варіанти вирішення:
 - *Оголосити формальний параметр правильного виду:*
 - `void somefunction (int (*pt) [4]) ;`
 - Альтернатива (тільки якщо pt є формальним параметром функції):
 - `void somefunction (int pt[][4]);`
 - Порожні квадратні дужки ідентифікують pt в якості вказівника.

Функції та багатовимірні масиви

```
1 #include <stdio.h>
2 #define ROWS 3
3 #define COLS 4
4 void sum_rows(int ar[][COLS], int rows);
5 void sum_cols(int ar[][COLS], int ); // ok to omit names
6 int sum2d(int (*ar)[COLS], int rows); // another syntax
7 int main(void)
8 {
9     int junk[ROWS][COLS] = {
10         {2,4,6,8},
11         {3,5,7,9},
12         {12,10,8,6}
13     };
14
15     sum_rows(junk, ROWS);
16     sum_cols(junk, ROWS);
17     printf("Sum of all elements = %d\n", sum2d(junk, ROWS));
18
19     return 0;
20 }
21
22 void sum_rows(int ar[][COLS], int rows)
23 {
24     int r;
25     int c;
26     int tot;
27
28     for (r = 0; r < rows; r++)
29     {
30         tot = 0;
31         for (c = 0; c < COLS; c++)
32             tot += ar[r][c];
33         printf("row %d: sum = %d\n", r, tot);
34     }
35 }
```

```
37 void sum_cols(int ar[][COLS], int rows)
38 {
39     int r;
40     int c;
41     int tot;
42
43     for (c = 0; c < COLS; c++)
44     {
45         tot = 0;
46         for (r = 0; r < rows; r++)
47             tot += ar[r][c];
48         printf("col %d: sum = %d\n", c, tot);
49     }
50 }
51
52 int sum2d(int ar[][COLS], int rows)
53 {
54     int r;
55     int c;
56     int tot = 0;
57
58     for (r = 0; r < rows; r++)
59         for (c = 0; c < COLS; c++)
60             tot += ar[r][c];
61
62     return tot;
63 }
```



array2d.c

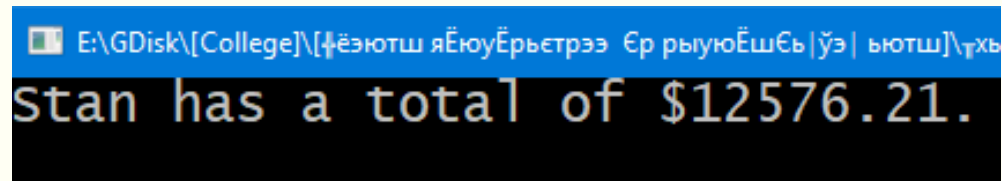
```
E:\GDisk\[College]\[Результаты ЕГЭ по информатике] Ер рыуюЕшЕь|ўэ|
row 0: sum = 20
row 1: sum = 24
row 2: sum = 36
col 0: sum = 17
col 1: sum = 19
col 2: sum = 21
col 3: sum = 23
Sum of all elements = 80
```

Коментарі до коду

- Зверніть увагу, що `ar` застосовується в тій же манері, як і `junk` у функції `main()`, оскільки вони мають однаковий тип.
- Наступне оголошення не буде працювати потрібним чином:
 - `int sum2(int ar[][], int rows);` // помилкове оголошення
 - Щоб компілятор міг оцінити такий вираз, він повинен знати розмір об'єкта, на який вказує `ar` (у зв'язку з переходом до вказівників).
 - `int sum2(int ar[][4], int rows);` // допустиме оголошення
 - `ar` вказує на масив з 4 значень `int` (16 байтів у нашій системі), тому `ar+1` означає "додати 16 байтів до адреси".
- Можна також включити розмір `i` в іншу пару квадратних дужок, проте компілятор його проігнорує:
 - `int sum2(int ar[3][4], int rows);` // допустиме оголошення, 3 ігнорується
 - Це зручно при використанні `typedef`:
 - `typedef int arr4[4];` // масив `arr4` з 4 значень `int`
 - `typedef arr4 arr3x4[3];` // масив `arr3x4` з 3 масивів `arr4`
 - `int sum2(arr3x4 ar, int rows);` // те ж, що і наступне оголошення
 - `int sum2(int ar[3][4], int rows);` // те ж, що і наступне оголошення
 - `int sum2(int ar[][4], int rows);` // стандартна форма

Передача структури в якості аргументу

```
1 #include <stdio.h>
2 #define FUNDLLEN 50
3
4 struct funds {
5     char    bank[FUNDLLEN];
6     double  bankfund;
7     char    save[FUNDLLEN];
8     double  savefund;
9 };
10
11 double sum(struct funds moolah); /* argument is a structure */
12
13 int main(void)
14 {
15     struct funds stan = {
16         "Garlic-Melon Bank",
17         4032.27,
18         "Lucky's Savings and Loan",
19         8543.94
20     };
21
22     printf("Stan has a total of $%.2f.\n", sum(stan));
23
24     return 0;
25 }
26
27 double sum(struct funds moolah)
28 {
29     return(moolah.bankfund + moolah.savefund);
30 }
```



E:\GDisk\[College]\[фёютш яёюёрьстрээ ёр ьуюёшёь|ўэ| ьютш]\тхь
Stan has a total of \$12576.21.

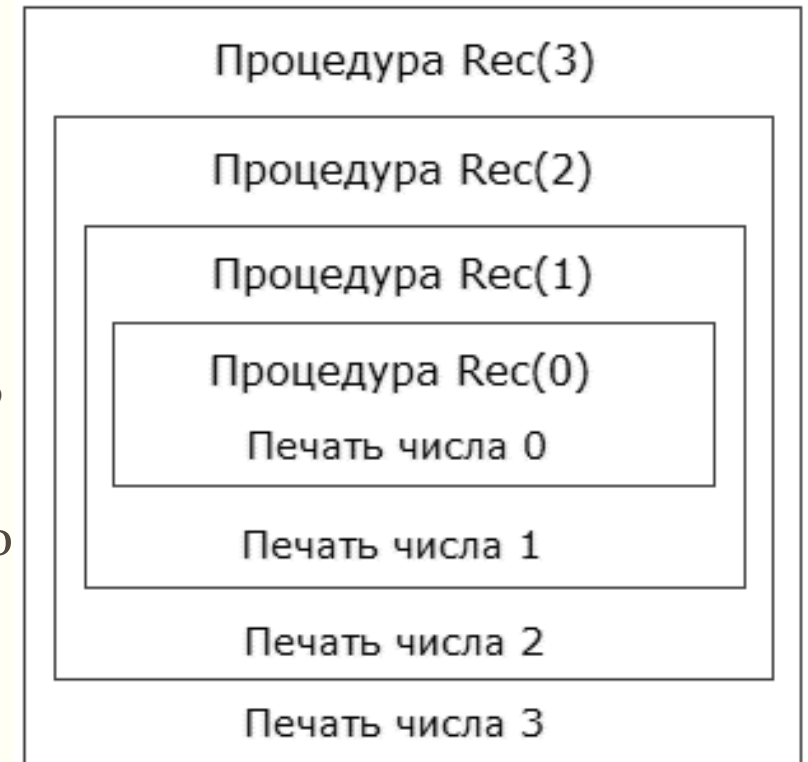
- При виклику `sum()` створюється автоматична змінна `moolah`, узгоджена з шаблоном `funds`.
 - Потім члени цієї структури ініціалізуються копіями значень відповідних членів структури `stan`.
 - Оскільки `moolah` є структурою, в програмі застосовується `moolah.bankfund`, а не `moolah->bankfund`.

Поняття рекурсії

- Рекурсія – це властивість об’єкта наслідувати (подражати) самому собі.
 - Об’єкт являється рекурсивним, якщо його частини виглядають так же, як і весь об’єкт.
- Рекурсія дуже широко застосовується в математиці та програмуванні:
 - структури даних:
 - граф (зокрема дерева і списки) можна розглядати як сукупність окремого вузла і підграфа;
 - Рядок складається з першого символу та підрядка;
 - шаблони проектування, наприклад декоратор.
 - Об’єкт декоратора може включати в себе інші об’єкти, які також є декораторами.
 - рекурсивні функції (алгоритми) виконують виклик самих себе.
- *Reduction* - найбільш поширений підхід при проектуванні алгоритмів.
 - зведення задачі X до іншої задачі Y означає створення алгоритму для X, який використовує алгоритм задачі Y як чорний ящик або підпрограму.
- Рекурсія – вид редукції, який спрощено описують так:
 - Якщо даний екземпляр задачі достатньо малий, розв’яжіть її.
 - Інакше зменшуйте (reduce) задачу до одного або кількох екземплярів тієї ж задачі.

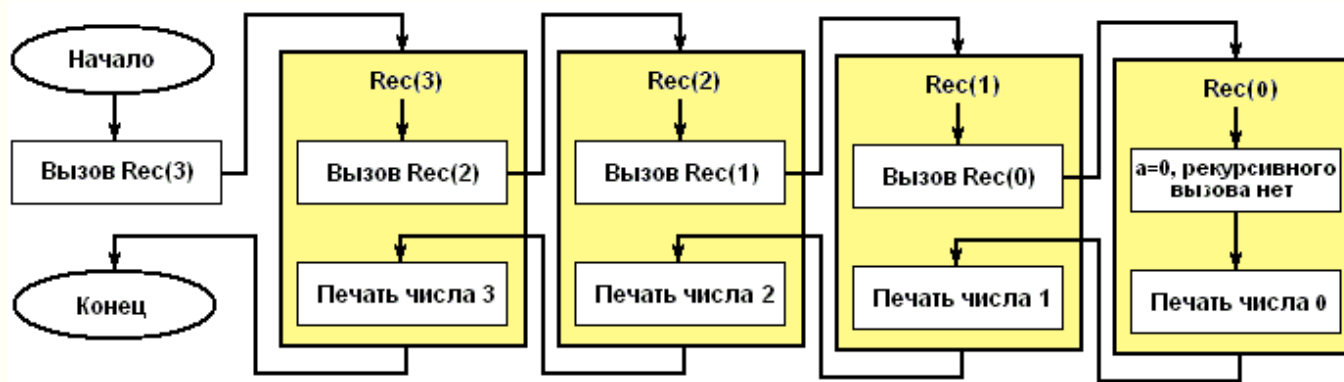
Сутність рекурсії

- Процедура Rec викликається с параметром $a = 3$.
 - У ній міститься виклик процедури Rec з параметром $a = 2$.
 - Попередній виклик ще не завершився.
- Процес виклику закінчується, коли параметр $a = 0$.
 - У цей момент одночасно виконуються 4 екземпляри процедури.
 - Кількість одночасно виконуваних процедур називають *глибиною рекурсії*.
- Четверта викликана процедура (Rec(0)) надрукує число 0 та закінчить свою роботу.
 - Після цього управління повертається до процедури, яка її викликала (Rec(1)) і друкується число 1.
 - І т.д., поки не завершаться всі процедури.
 - Виведеться: 0, 1, 2, 3.



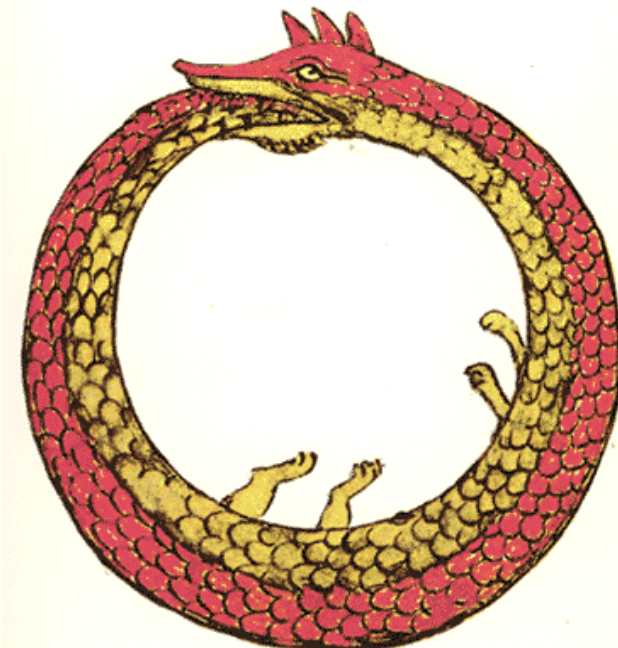
Сутність рекурсії

- процедура може викликати саму себе.



```
procedure Rec(a: integer);  
begin  
    if a>0 then  
        Rec(a-1);  
    writeln(a);  
end;
```

- Зверніть увагу, що рекурсивний виклик стоїть всередині умовного оператора.
 - Це необхідна умова для завершення рекурсії.
- Сама себе процедура викликає з іншим параметром, ніж той, з яким вона викликала.
 - Якщо глобальні змінні не використовуються, це потрібно, щоб рекурсія не продовжувалась нескінченно.



Рекурсія в дії



26.11.2019

@Марченко С.В., ЧДБК, 2019

Рекурсивний алгоритм

- **Рекурсивний алгоритм** – це алгоритм, в описі якого прямо або опосередковано міститься звернення до самого себе.
- Розробці рекурсивних алгоритмів передуює рекурсивна тріада:
 - **Параметризація** – з постановки задачі виділяються параметри, які описують первинні дані. Надалі можливе введення додаткових параметрів, які не передбачені в умові, проте використовуються при встановленні залежностей.
 - **Виділення бази** – передбачає знаходження в задачі, яка розв’язується, тривіальних випадків, результат яких очевидний. Вірно обрана база рекурсії забезпечує завершеність рекурсивних звернень, які в кінці зводяться до базового випадку.
 - **Декомпозиція** – зведення загального випадку до більш простих підзадач, які відрізняються від початкової задачі набором вхідних даних. Декомпозиційні залежності описують не лише зв’язок між задачею та підзадачами, але й характер зміни значень параметрів на черговому кроці.

```

1  #include <stdio.h>
2  long fact(int n);
3  long rfact(int n);
4  int main(void)
5  {
6      int num;
7
8      printf("This program calculates factorials.\n");
9      printf("Enter a value in the range 0-12 (q to quit):\n");
10     while (scanf("%d", &num) == 1)
11     {
12         if (num < 0)
13             printf("No negative numbers, please.\n");
14         else if (num > 12)
15             printf("Keep input under 13.\n");
16         else
17         {
18             printf("loop: %d factorial = %ld\n",
19                    num, fact(num));
20             printf("recursion: %d factorial = %ld\n",
21                    num, rfact(num));
22         }
23         printf("Enter a value in the range 0-12 (q to quit):\n");
24     }
25     printf("Bye.\n");
26
27     return 0;
28 }

```

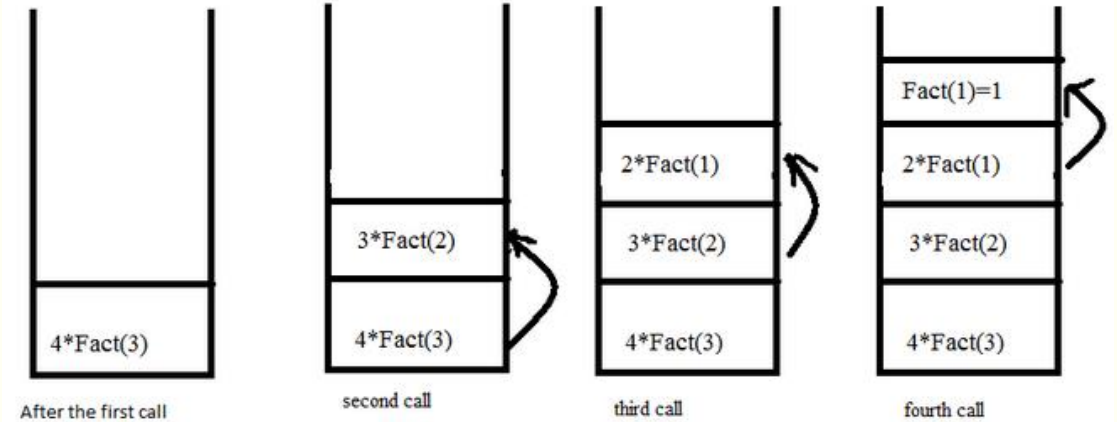
- Факторіал цілого числа — це добуток усіх цілих чисел, починаючи з 1 та закінчуючи заданим числом.
 - Наприклад, факторіал 3 ($3! = 1 \cdot 2 \cdot 3 = 6$).
 - $0!$ береться рівним 1, а для від'ємних чисел факторіали не визначені.
- Тестова програма обмежує вхідні дані цілими значеннями в діапазоні від 0 до 12.
 - Значення $12!$ трохи менше пів мільярда, тому результат $13!$ не поміститься в тип long у нашій системі.


```

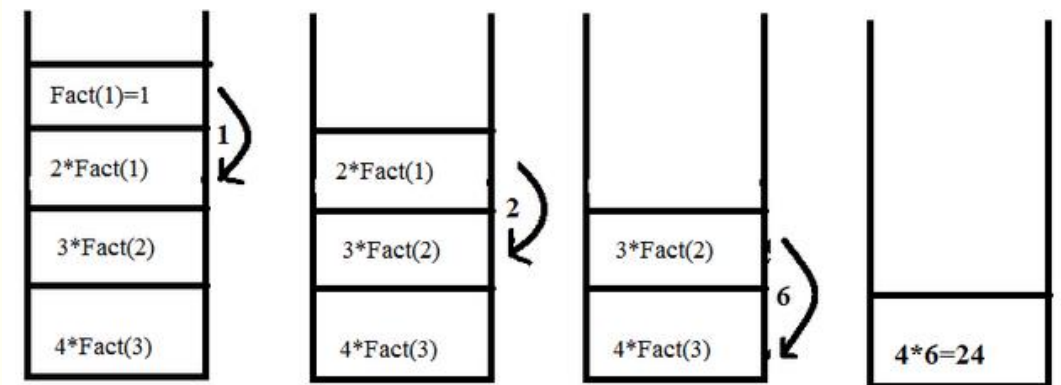
30 long fact(int n)    // loop-based function
31 {
32     long ans;
33
34     for (ans = 1; n > 1; n--)
35         ans *= n;
36
37     return ans;
38 }
39
40 long rfact(int n)    // recursive version
41 {
42     long ans;
43
44     if (n > 0)
45         ans = n * rfact(n-1);
46     else
47         ans = 1;
48
49     return ans;
50 }

```

When function call happens previous variables gets stored in stack



Returning values from base case to caller function



Рекурсія vs ітерація

КРИТЕРІЙ ПОРІВНЯННЯ	РЕКУРСІЯ	ІТЕРАЦІЯ
Основа	Інструкція в тілі функції викликає саму функцію.	Дозволяє набору інструкцій повторювано виконуватись.
Формат	У рекурсивній функції задається лише умова переривання (базовий випадок).	Містить ініціалізацію, умову, виконання інструкцій у циклі та оновлення лічильника.
Переривання	Умовний оператор включено в тіло функції, щоб змусити її до виходу без виконання рекурсивного виклику.	Ітераційні інструкції повторно виконуються, поки не досягнуто виконання певної умови.
Умова	Якщо функція не зводиться до базового випадку, це призведе до нескінченної рекурсії.	Якщо управляюча умова в ітераційному операторі МП ніколи не стане false, це призведе до нескінченної ітерації.
Нескінченне повторення	Нескінченна рекурсія може crash систему.	Нескінченний цикл використовує повторювано CPU-цикли.
Застосування	Рекурсія завжди застосовується до функцій.	Ітерація завжди використовується до інструкцій або «циклів».
Стек	Стек використовується для зберігання набору нових локальних змінних та параметрів при кожному виклику функції.	Не використовує стек.
Накладні витрати	На повторні виклики функцій.	Немає накладних витрат на повторні виклики функцій.
Швидкість	Повільно виконується.	Швидко виконується.
Розмір коду	Зменшує розмір коду.	Робить код довшим.

Переваги та недоліки рекурсії

- Перевага: пропонує найпростіший розв'язок цілого ряду задач програмування.
- Недолік: деякі рекурсивні алгоритми можуть швидко вичерпати ресурс пам'яті комп'ютера.
- Недолік: рекурсію складно документувати та супроводжувати.

```
unsigned long Fibonacci(unsigned n)
{
    if (n > 2)
        return Fibonacci(n-1) + Fibonacci(n-2);
    else
        return 1;
}
```

- У представленій функції використовується подвійна рекурсія.
 - На кожному рівні кількість змінних подвоюється, тобто їх об'єм росте експоненційно!



ДЯКУЮ ЗА УВАГУ!