

## Методические указания

### Тематическое занятие 8

## Рекурсия.

### Содержание

|  |   |
|--|---|
| <b>Рекурсия</b> .....                          | 1 |
| <i>Понятие рекурсии</i> .....                  | 1 |
| <i>Разбор рекурсии на примере</i> .....        | 2 |
| <i>Глубина и уровень рекурсии</i> .....        | 3 |
| <i>Таблица трассировки</i> .....               | 3 |
| <i>Недостатки и достоинства рекурсии</i> ..... | 3 |
| <b>Формы рекурсивных функций</b> .....         | 4 |
| <i>Условие останова</i> .....                  | 4 |
| <i>Структуры рекурсивных функций</i> .....     | 4 |
| <b>Упражнения</b> .....                        | 5 |
| <i>Упражнение 8.1</i> .....                    | 5 |
| <i>Упражнение 8.2</i> .....                    | 5 |

## Рекурсия

### Понятие рекурсии

Рекурсивным называется объект, который частично определяется через самого себя. В программировании на языке С **рекурсия** – способ организации вычислительного процесса, при котором функция **вызывает сама себя**.

Рекурсивные определения широко используются во многих областях, особенно в математике.

Рассмотрим функцию факториала  $n!$ . Как правило, ее определяют как произведение первых  $n$  целых чисел:

$$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n$$

Такое произведение можно вычислить с использованием итеративной конструкции цикла (это решение приводилось ранее, см. **тематическое занятие 3**):

```
#include <stdio.h>
int main(void) {
    int n, i;
    long int fact=1;
    printf("Input n:");
    scanf("%d", &n);
```

```

    for (i=1; i<=n; i++)
        fact*=i;
    printf("Factorial n!=%ld\n", fact);
    return 0;
}

```

Однако существует также другое (рекурсивное) определение факториала, в котором используется **рекуррентная** формула:

|  |
|--|
| (1) $0! = 1$<br>(2) для любого $n > 0$ $n! = n \cdot (n - 1)!$ |
|--|

Этому определению факториала соответствует программа, использующая рекурсивную функцию, которая вызывает сама себя:

```

#include <stdio.h>

long int fact(int i);

int main(void) {
    int n;
    printf("Input n:");
    scanf("%d", &n);
    printf("Factorial n!=%ld\n", fact(n));
    return 0;
}

long int fact(int i) {
    if (i==1) /* условие останова */
        return 1;
    else
        return i*fact(i-1); /* рекурсивный вызов */
}

```

Следует обратить внимание, что здесь совсем **нет операторов цикла**. При этом операция умножения будет **повторяться  $n-1$  раз**, поскольку функция `fact` вызывает саму себя  $n-1$  раз.

Заметьте, что использование рекурсии позволяет легко (почти дословно) запрограммировать вычисления по рекуррентным формулам.

## Разбор рекурсии на примере

Разберем подробнее приведенный пример рекурсивного вычисления факториала  $n!$ .

При каждом **рекурсивном вызове** `fact(i-1)` выполнение текущей функции приостанавливается, но ее переменные не удаляются из памяти. Происходит новый вызов функции, для переменных которой также выделяется память, и так далее. Образуется последовательность прерванных процессов, из которых выполняется всегда последний. Передаваемый в функцию параметр каждый раз уменьшается на 1.

Когда параметр станет равным 1, выполнится **условие останова** `i==1`. Рекурсивные вызовы закончатся, и функция вернет значение равное 1 (`return 1;`). Текущий вызов функции окажется последним. Затем в обратном порядке последовательно закончат работу все остальные вызванные функции.

Последняя завершенная функция (она же – первая вызванная) вернет результат, который будет выведен на экран стандартной функцией `printf`.

## Глубина и уровень рекурсии

Максимальное число рекурсивных вызовов функции без возвратов называется **глубиной** рекурсии. Число рекурсивных вызовов в каждый конкретный момент времени называется **текущим уровнем** рекурсии.

В рассмотренном примере, если пользователь ввел число  $n=5$ , то общее количество вызовов функции `fact` равно 5, а глубина рекурсии (количество рекурсивных вызовов) равна 4. При первом вызове функции `fact` (в качестве одного из параметров функции `printf`) текущий уровень рекурсии считается равным нулю, т.к. функция еще не успела вызвать саму себя.

Поскольку имена параметров и локальных переменных не меняются при каждом рекурсивном вызове, то **воспользоваться значением некоторой переменной  $i$ -го уровня рекурсии можно, находясь только на этом  $i$ -м уровне**. В рассмотренном примере у функции `fact` нет локальных переменных, но есть параметр `i`, область видимости которого соответствует данному правилу.

## Таблица трассировки

Для демонстрации действий, выполняемых рекурсивной функцией, используют **таблицу трассировки** значений ее параметров по уровням рекурсии.

В рассмотренном примере рекурсивного вычисления факториала действия выполняются на рекурсивном возврате, и при вводе пользователем  $n=5$  таблица трассировки:

| Текущий уровень рекурсии | Рекурсивный спуск                     | Рекурсивный возврат                  |
|--------------------------|---------------------------------------|--------------------------------------|
| 0                        | Ввод: $n=5$ <code>fact(5)</code>      | Вывод: $n!=120$                      |
| 1                        | $i=5$ <code>fact(4)</code>            | <code>return 5*24; (fact=120)</code> |
| 2                        | $i=4$ <code>fact(3)</code>            | <code>return 4*6; (fact=24)</code>   |
| 3                        | $i=3$ <code>fact(2)</code>            | <code>return 3*2; (fact=6)</code>    |
| 4                        | $i=2$ <code>fact(1)</code>            | <code>return 2*1; (fact=2)</code>    |
| 5                        | $i=1$ <code>return 1; (fact=1)</code> |                                      |

## Недостатки и достоинства рекурсии

Для каждого нового рекурсивного вызова функции выделяется память, которая освобождается только после достижения всей глубины рекурсии. Поэтому выполнение рекурсивных функций требует **значительно большего объема оперативной памяти** во время выполнения программы, чем нерекурсивных. Если глубина рекурсии очень велика, то это может привести к **переполнению памяти**.

К тому же рекурсивные алгоритмы не дают выигрыша в быстродействии, а, как правило, выполняются **медленнее**.

Однако код программы с рекурсией обычно компактнее, и его значительно легче писать и дорабатывать. Рекурсия удобна при работе с рекурсивно определенными алгоритмами и структурами данных.

# Формы рекурсивных функций

## Условие останова

Рекурсивное определение позволяет с помощью конечного выражения определить бесконечное множество объектов. А с помощью рекурсивного алгоритма можно определить **бесконечное вычисление**, причем алгоритм не будет содержать повторений фрагментов кода.

Рекурсивные функции, не содержащие **условия останова** – **прекращения рекурсивных вызовов**, – приводят к бесконечным процессам.

Например:

```
#include <stdio.h>

void endless(void);

int main(void) {
    endless();
    return 0;
}

void endless(void) {
    printf("Функция вызвана. Снова вызываем функцию:\n");
    endless();
}
```

Действие программы прервется, когда будет исчерпана вся свободная память. Использовать подобные рекурсивные функции **невозможно** из-за ограниченности оперативной памяти.

Поэтому **главное требование** к рекурсивным функциям заключается в том, что **вызов рекурсивной функции должен выполняться по условию, которое на каком-то уровне рекурсии станет ложным**.

Если это условие истинно, то **рекурсивный спуск** продолжается. А если условие становится ложным, то спуск заканчивается и начинается поочередный **рекурсивный возврат** из всех вызванных на данный момент копий рекурсивной функции.

## Структуры рекурсивных функций

Структура рекурсивной функции может принимать **три** различные формы. Рассмотрим их на примере функции `rec()` без параметров, которая не возвращает никакого значения.

### 1) Выполнение действий на рекурсивном спуске

```
void rec(void) {
    ...
    Операторы; /* вначале выполняются действия */
    ...
    if (Условие) rec(); /* затем происходит рекурсивный вызов */
}
```

### 2) Выполнение действий на рекурсивном возврате

```
void rec(void) {
    if (Условие) rec(); /* вначале происходит рекурсивный вызов */
    ...
    Операторы; /* затем выполняются действия */
    ...
}
```

### 3) Выполнение действий как на рекурсивном спуске, так и на рекурсивном возврате

```
void rec(void) {  
    ...  
    Операторы1; /* вначале выполняется 1-я часть действий */  
    ...  
    if (Условие) rec(); /* потом происходит рекурсивный вызов */  
    ...  
    Операторы2; /* затем выполняется 2-я часть действий */  
    ...  
}
```

или то же самое, но с проверкой условия вначале:

```
void rec(void) {  
    if (Условие) { /* проверка условия останова */  
        ...  
        Операторы1; /* 1-я часть действий */  
        ...  
        rec(); /* рекурсивный вызов */  
        ...  
        Операторы2; /* 2-я часть действий */  
        ...  
    }  
}
```

Многие задачи безразличны к тому, какая форма рекурсивных функций используется. Но существуют классы задач, при решении которых требуется сознательно управлять ходом работы рекурсивных функций.

## Упражнения

### Упражнение 8.1

Приведенную программу рекурсивного вычисления факториала дополнить выводом таблицы трассировки на экран. Обязательные поля таблицы: *текущий уровень рекурсии*, *значения на рекурсивном спуске*, *значения на рекурсивном возврате*.

### Упражнение 8.2

Составить программу, которая, используя рекурсивную функцию, находит значение данной функции для любых целых неотрицательных аргументов  $n$  и  $a$ :

$$R(n, a) = \underbrace{\sqrt{a + \sqrt{a + \dots + \sqrt{a}}}}_{n\text{-корней}}$$

Вывести на экран таблицу трассировки.