

Таблица трассировки для рекурсивных функций создаётся также как и для обычных, однако, для каждого нового вызова рекурсивной функции необходима новая таблица трассировки. Рисунок 2 – это один из вариантов таблицы трассировки для рекурсивных вызовов функции.

В С функции могут вызывать сами себя. Функция является рекурсивной, если оператор в теле функции вызывает функцию, содержащую данный оператор. Иногда называемая круговым определением, рекурсия является процессом определения чего-либо с использованием самой себя.

Простым примером является функция factr(), вычисляющая факториал целого числа. Факториал числа N является произведением чисел от 1 до N. Например, факториал 3 равен 1\*2\*3 или 6. Как factr(), так и его итеративный эквивалент показаны ниже:  
  
/\* Вычисление факториала числа \*/  
  
int factr(int n) /\* рекурсивно \*/  
{  
int answer;  
if(n==1) return(1);  
answer = factr(n-1)\*n;  
return(answer);  
}  
  
/\* Вычисление факториала числа \*/  
  
int fact (int n)    /\* нерекурсивно \*/  
{  
int t, answer;  
answer == 1;  
for(t=1; t<=n; t++)  
answer=answer\*(t);  
return(answer);  
}

Действие нерекурсивной версии fact() должно быть совершенно очевидно. Она использует цикл, начиная с 1 и заканчивая указанным числом, последовательно перемножая каждое число на ранее полученное произведение.

Действие рекурсивной функции factr() немного более сложно. Когда factr() вызывается с аргументом 1, функция возвращает 1. В противном случае она возвращает произведение factr(n- 1) \* n. Для вычисления этого значения factr() вызывается с n-1. Это происходит, пока n не станет равно 1.

При вычислении факториала числа 2, первый вызов factr() приводит ко второму вызову с аргументом 1. Данный вызов возвращает 1, после чего результат умножается на 2 (исходное значение n). Ответ, таким образом, будет 2. Можно попробовать вставить printf() в factr() для демонстрации уровней и промежуточных ответов каждого вызова.

Когда функция вызывает сама себя, в стеке выделяется место для новых локальных переменных и параметров. Код функции работает с данными переменными. Рекурсивный вызов не создает новую копию функции. Новыми являются только аргументы. Поскольку каждая рекурсивно вызванная функция завершает работу, то старые локальные переменные и параметры удаляются из стека и выполнение продолжается с точки, в которой было обращение внутри этой же функции. Рекурсивные функции вкладываются одна в другую как элементы подзорной трубы.

Рекурсивные версии большинства подпрограмм могут выполняться немного медленнее, чем их итеративные эквиваленты, поскольку к необходимым действиям добавляются вызовы функций. Но в большинстве случаев это не имеет значения. Много рекурсивных вызовов в функции может привести к переполнению стека. Поскольку местом для хранения параметров и локальных переменных функции является стек и каждый новый вызов создает новую копию переменных, пространство стека может исчерпаться. Если это произойдет, то возникнет ошибка - переполнение стека.

Основным преимуществом применения рекурсивных функций является использование их для более простого создания версии некоторых алгоритмов по сравнению с итеративными эквивалентами. Например, сортирующий алгоритм Quicksort достаточно трудно реализовать итеративным способом. Некоторые проблемы, особенно связанные с искусственным интеллектом, также используют рекурсивные алгоритмы. Наконец, некоторым людям кажется, что думать рекурсивно гораздо легче, чем итеративно.