**Практическое занятие №9**

**"Рекурсивные функции"**

**Теория**

Язык C (и все Си-подобные языки) допускает создание и использование рекурсивных функций. Это означает, что внутри функции можно обращаться к самой функции. Однако чтобы такое обращение имело смысл, функция должна быть организована должным образом, т.е. реализовала бы именно рекурсивный алгоритм. Например, составим функцию, вычисляющую сумму элементов одномерного массива. Если использовать известный прием накопления суммы, предусматривающий вычисление суммы в цикле, то такая функция может иметь следующий вид:

//нерекурсивный вариант функции вычисления суммы элементов //массива

#include "stdio.h"

float sum(float a[],int n);

int main()

{ const int nn=20;

float a[nn],s;

int i,n;

printf("wwedite kol-wo elementow\n");

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

printf("\nwwedite massiw\n");

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

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

s=sum(a,n);

printf("\nsum=%f",s);

return 0;

}

float sum(float a[],int n)

{float s=0;

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

s+=a[i];

return s;

}

Для построения рекурсивного варианта подпрограммы следует сформулировать **рекурсивное утверждение и условие окончания рекурсии**. В нашем случае рекурсивное утверждение будет заключаться в том, что сумма элементов массива из n элементов может быть вычислена как сумма (n-1)-го начальных элементов и последнего элемента, то есть Sn =Sn-1 + an (n>0). Условие окончания рекурсии S0 = a0. Рекурсивный вариант функции приведен ниже

// рекурсивный вариант функции вычисления суммы элементов массива

float sum(float a[],int n) //n – количество элементов массива

{ float s;

if (n==1)

s=a[n-1];

else

s=a[n-1]+sum(a,n-1);

return s;

}

Рекурсивное описание обычно является более компактным и выглядит более наглядно, если **природа самого алгоритма рекурсивна**. Однако надо принимать во внимание и то, как реализуется рекурсивный алгоритм в ЭВМ. Например, при n=4 (массив состоит из четырех элементов) производится первое обращение к функции со значениями параметров массива a и n=4. При этих значениях выполняется оператор s=a[3]+sum(a,3), затем обращение к функции со значениями a и 3, т.е. sum(a,3). Следом выполняется обращение к функции со значениями a и 2, т.е. sum(a,2). При выполнении этого обращения к функции производится еще одно обращение: sum(a,1). На последнем шаге при обращении к функции с этими параметрами будет выполнен оператор s=a[0]. Далее процесс развивается в обратном направлении, т.е. выполнится оператор S=a[1]+sum(a,1), что даст результат s=a[1]+a[0], затем выполнится оператор s=a[2]+sum(a,2) с результатом s=a[2]+a[1]+a[0]. На последнем шаге выполнится оператор s=a[3]+sum(a,3), дающий окончательный результат. Рассмотрев описанный процесс, можно придти к выводу, что рекурсивная подпрограмма требует больших затрат машинного времени и памяти. Это объясняется необходимостью выполнения повторных обращений к подпрограмме и необходимостью хранения (в стеке) фактических параметров при обращениях к подпрограмме.

Рекурсия может быть и косвенной. В этом случае программа обращается сама к себе опосредованно, т.е. первая программа обращается ко второй, которая, в свою очередь, содержит обращение к первой.

1. float pr1(float a, float b)
2. {
3. . . . . .
4. s=a+pr2(a,b);
5. . . . . .
6. }
7. float pr2(float c, float d)
8. {
9. . . . . .
10. s=d+pr1(c,d);
11. . . . . .
12. }

Согласно правилам языка C каждый идентификатор должен быть предварительно объявлен, поэтому, чтобы разрешить использование косвенной рекурсии достаточно объявить функцию pr2 до определения функции pr1 или поместить объявление функций (прототипы) до функции main, а определения функций расположить (как и положено в языке) после главной функции.

Далее приведен пример программы, в которой реализована косвенная рекурсия.

// косвенная рекурсия

#include "stdio.h"

#include "conio.h"

float f1(float a,int n);

float f2(float a,int n);

int main()

{ int n;

float sum;

scanf\_s(“%d”,&n);

sum=f1(2.0,n);

printf("sum=%f",sum);

\_getch();

return 0;

}

float f1(float a,int n)

{ float s;

if (n==1) s=5;

else s=n+a+f2(2.0,n-1);

return s;

}

float f2(float a,int n)

{ float s;

if (n==1) s=10;

else s=n+a+f1(1.0,n-1);

return s;

}

**Практика**

Пример 1. Вычислить факториал с помощью итеративной функции и рекурсивной функции. **(lab9primer1.cpp)**

Пример 2. Реализовать поиск числа из ряда Фиббоначи по введенному порядковому номеру числа с помощью рекурсивной функции. **(lab9primer2.cpp).** Напишите итеративную функцию поиска числа Фиббоначи. Подумайте, как можно записать рекурсивную функцию более упрощенно.

Пример 3. Вычислить сумму элементов массива с помощью итеративной функции и рекурсивной функции. **(lab9primer3.cpp).**

Пример 4. Использование переменной типа static. При рекурсивном вызове программы может сложиться ситуация, когда нам нужно сохранять некоторые данные при очередном вызове функции. Для этого можно создавать элементы типа static. Отличие статических элементов заключается в том, что они создаются статической области программы и их значения сохраняются после завершения функции. **(lab9primer4.cpp).**

**Задание.** Создать рекурсивную функцию, которая на вход получает целое число, ищет простой множитель для этого числа, если находит, то выводит этот множитель на экран, после чего рекурсивно вызывает себя для числа, разделённого на найденный множитель. И после завершения вызванной программы заканчивает работу. Если простого множителя не найдено, то функция выводит исходное число и заканчивает работу. Прототип функции: **void prime\_factor(int n);**

**Задание.** Создать проект. Написать и отладить рекурсивную функцию печати целого числа в двоичном формате. Прототип: **void bit\_element(int n);** Если входной элемент равен 0, то указанная функция завершает работу и начинается обратный ход, если нет, то рекурсивно вызывается **bit\_element(n/2);**