ЛЕКЦИЯ 11.

Вопросы:

- **Рекурсивные функции**
- **Указатели на функции**
- **Использование указателя на функцию в качестве параметра функции**

Рекурсивные функции

Под рекурсией понимается вызов подпрограммы из тела этой же подпрограммы. Подобные соотношения достаточно часто встречаются в математике. Так, вычисление факториала можно представить следующим образом:

А нахождение целой степени произвольного числа как:

```
int Fact(int n)
{
  if (n == 0)
    return 1;
  else
    return n * Fact(n-1);
}

float IntPower(float x; int n)
{
  if (n == 0)
    return 1;
  else
    return x * IntPower(x,n-1);
}
```

Пример. ***Ввести последовательность чисел, оканчивающуюся нулем и вывести их в обратном порядке.***

```
void Reverse()
{    int ch;
     scanf (”%d”,&ch);
     if (ch != 0)  Reverse();
     printf (”%d”,ch);
}
```

Формы рекурсивных процедур

Структура рекурсивной процедуры может принимать три разные формы:

1. Форма с выполнением действий до рекурсивного вызова **(с выполнением действий на рекурсивном спуске):**

```
void Rec()
{
     S;
     if (условие)  Rec();
}
```

2. Форма с выполнением действий после рекурсивного вызова **(с выполнением действий на рекурсивном возврате):**

```
 void  Rec()
{    if (условие) Rec();
     S;
}
```

3. Форма с выполнением действий как до, так и после рекурсивного вызова **(с выполнением действий как на рекурсивном спуске, так и на рекурсивном возврате):**

```
void  Rec()
{
  S1;
  if (условие) Rec();
  S2;
}
```

Максимальное число рекурсивных вызовов функции без возвратов, которое происходит во время выполнения программы, называется ***глубиной рекурсии.***

Число рекурсивных вызовов в каждый конкретный момент времени, называется ***текущим уровнем рекурсии.***

В общем случае любая рекурсивная функция включает в себя некоторое множество операторов S и один или несколько операторов рекурсивного вызова.

Безусловные рекурсивные функции приводят к бесконечным процессам, так как практическое использование функций с бесконечным **самовызовом невозможно**.

Такая невозможность вытекает из того, что для каждой копии рекурсивной функции необходимо выделять дополнительную область памяти, а бесконечной

памяти не существует.

Следовательно, 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 условие истинно, то рекурсивный спуск продолжается. Когда оно станет ложным, спуск заканчивается и начинается поочередный рекурсивный возврат из всех вызванных на данный момент копий рекурсивной функции .

В примере выполнения рекурсивной процедуры ввода чисел и вывода их в обратном порядке, действия выполняются до рекурсивного вызова, т.е. на рекурсивном спуске (оператор scanf) и на рекурсивном возврате (оператор prinf).

Пример. ***Вычисление наибольшего общего делителя двух натуральных чисел.***

Алгоритм Евклида реализован в виде рекурсивной функции.

```
int f_nod(int x, int y)
//рекурсивная  функция
{
  if (y>x)
    return f_nod(y,x);
  else
    if (y==0)
        return x;
   else
        return f_nod(y,x%y);
}
```

Примеры ***конечного*** рекурсивного алгоритма, определяющего ***бесконечные*** вычисления.

```
void Pech()
{   puts("Отыщи всему начало");
    puts("И ты многое поймешь! ");
    Pech();
}

void PechNO()
{   PechNO();
    puts("Отыщи всему начало");
    puts("И ты многое поймешь! ");
}
```

Задача о Ханойских башнях

*Даны три стержня A, B, C. На стержне A находятся **n** дисков разного диаметра, пронумерованных сверху вниз. Каждый меньший диск находится на большем. Требуется переместить эти диски на стержень C, сохранив их взаиморасположение. Перемещать можно только по одному верхнему диску и нельзя класть больший диск на меньший.*

*/*Алгоритм Задачи о Ханойских башнях :*
Если n=1 то
1. Переместить этот единственный диск с A на C и остановиться
иначе
2. Переместить верхние n-1 дисков с A на B, используя C как вспомогательный.
3. Переместить оставшийся нижний диск с A на C.
4. Переместить n-1 дисков с B на C, используя A как вспомогательный.
**/*

```
void Mov(int n, char A, char C, char B)
//n - число дисков
//A – исходный стержень, на котором находятся диски
//B – вспомогательный стержень
//C – стержень, на который надо переставить диски
{
   if (n==1)
    cout<< "Переставить диск 1 с " << A << " на " << C;
   else
   {
      Mov(n-1,A,B,C);
      cout<< "Переставить диск" << n
                  << "с "  << A << " на " << C;
      Mov(n-1,B,C,A);
   }
}
void main()
{
 int n;
 cout << "введите число дисков";
 cin >> n;
 cout<< "решение: ";
 Mov(n, 'A', 'C', 'B');
}
```

Пример, когда рекурсивный алгоритм становится крайне неэффективным.

Известную в комбинаторике величину «число сочетаний из n элементов по k» можно вычислять как по рекурсивной, так и по нерекурсивной формуле

$$C_n^k = \begin{cases} C_{n-1}^k + C_{n-1}^{k-1}, \text{ если } k \neq 1, k \neq n, \\ 1 \quad \text{в противном случае} \end{cases}$$

$$C_n^k = \frac{n!}{k!(n-k)!} = \frac{(n-t+1)\cdot(n-t+2)\cdot\ldots\cdot n}{t\cdot(t-1)\cdot\ldots\cdot 1}, \quad \text{где} \quad t = \min(k, n-k)$$

```
int sochet1(int n, int k)
{
 if ((k==0)||(k==n))
  return 1;
 else
  return(sochet1(n-1,k)+ sochet1(n-1,k-1));
}

int sochet2(int n, int k)
int i, t, s;
{
 if ((k==0)||(k==n))
    return 1;
 else
  {
   if (n-k > k) t=k;
   else t=n-k;
   s = 1;
   for (i=1;i<=t;i++)
      s=s*(n-i+1))/i;
   return s;
  }
}
```

Упр. Сравнить время выполнения рекурсивного и нерекурсивного алгоритмов.

Указатели на функции

Каждая функция характеризуется типом возвращаемого значения, именем и сигнатурой. Сигнатура определяется количеством, порядком следования и типами параметров.

Иногда под сигнатурой функции подразумевают список типов ее параметров.

Имя функции без скобок и параметров выступает в роли указателя на эту функцию и его значением является адрес размещения функции в памяти.

Указатель на функцию определяется следующим образом:

тип_функции(*имя_указателя)(спецификация_ параметров);

Например: **int(*f1Ptr)(char)** определяет указатель **f1Ptr** на функцию с параметром типа char, возвращающую значение типа **int.**

Не путать с описанием

int* f1(char);

- это прототип функции с именем **f1** и параметром типа **char,** возвращающей значение указателя типа **int.**

char* (*f2Ptr)(char*, int);

Это определение указателя **f2Ptr** на функцию с параметрами типа указатель на **char** и типа **int**, возвращающую значение типа указатель на **char**.

В определении указателя на функцию тип возвращаемого значения и сигнатура должны совпадать с соответствующими типами и сигнатурами тех функций, адреса которых предполагается присваивать вводимому указателю при инициализации или с помощью оператора присваивания.

```
#include <iostream>
using namespace std;
void f1 (void)
{cout  << "выполняется f1()\n";}
void f2 (void)
{cout  << "выполняется f2()\n";}
void main()
{ setlocale(LC_ALL,".1251");
  void (*ptr)(void); // ptr - указатель на функцию
  ptr=f2;            // присваивается адрес f2()
  (*ptr)();          // вызов f2 по адресу
  ptr=f1;            // присваивается адрес f1()
  (*ptr)();          // вызов f1 по адресу
  ptr();             // вызов эквивалентен (*ptr)();
}
```

Результат выполнения:

```
выполняется f2()
выполняется f1()
выполняется f1()
```

В программе описан указатель **ptr** на функцию, и ему последовательно присваиваются адреса функций **f1** и **f2**.

Формат вызова функции:

```
(*имя_указателя)(список_фактических_параметров);
```

Значением **имени_указателя** служит адрес функции, а с помощью операции разыменования ***** обеспечивается обращение по адресу к этой функции. Ошибкой будет обращение

```
*ptr();
```

Ибо () имеют более высокий приоритет, чем ***.**

Указатель на функцию может инициализироваться при определении. При этом должны совпадать тип и сигнатура. Аналогично при присваивании значений указателю и при вызове функций с помощью указателей.

```
char f1(char){…} //определение функции
char f2(int){…}  //определение функции
void f3(float){…} //определение функции
int* f4(char*){…} //определение функции
char (*pt1)(int); //указатель на функцию
char (*pt2)(int); //указатель на функцию
void (*pt3)(float)=f3;
 //инициализированный указатель на функцию
…
 pt1=f1; //ош-несоотв сигнатур
 pt2=f3; //ош-несоотв типов и сигнатур
 pt1=f4; //ош-несоотв типов
 pt1=f2; //ok!
 pt2=pt1; //ok!
 char c=(*pt1)(44);//ok!
 c=(*pt2)('\t');// ош-несоотв сигнатур
```

Пример вызова функций с помощью указателей:

```
#include <iostream>
using namespace std;
//функции одного типа с одинаковыми //сигнатурами
int add (int n, int m) {return n+m;}
```

```
int divi(int n, int m) {return n/m;}
int mult(int n, int m) {return n*m;}
int subt(int n, int m) {return n-m;}

void main()
{   setlocale(LC_ALL,".1251");
    int (*par)(int, int);
    // указатель на функцию
    int a=6, b=2;
    char c='+';
    while (c!='0')
    {  cout<<"знак операции или 0?\n";
       cin>>c;
      switch (c)
         {case '0': break;
          case '+': par=add;  break;
          case '-': par=subt; break;
          case '*': par=mult; break;
          case '/': par=divi; break;
          default : cout <<
          "неверная операция" << endl;
           c  ='1'; break;
         }
         if ((c!='1')&&(c!='0'))
         cout << a << c << b << "="<< (*par)(a,b)<< endl;
                                        //вызов по адресу
    }
}
```

Вызывается функция по адресу, содержащемуся в указателе **par** следующим образом:

(*par)(a,b); **(1)**

Альтернативная форма вызова через указатель подобна нормальному вызову функции:

par(a,b); **(2)**

Однако предпочтительнее использовать вариант 1.

Указатели на функции часто используют в качестве аргументов функций.

Полезным приложением указателей на функции является выбор одной из нескольких функций. Ранее рассмотрели использование switch.

В следующем примере создадим массив указателей на функции.

```
#include <stdio.h>
#include <iostream>
using namespace std;
void f1(int);
void f2(int);
void f3(int);
int main()
{
 setlocale(LC_ALL,".1251");
 void (*f_array[])(int)={f1,f2,f3};
 int vibor;
 do
 {
        printf("Введите номер функции:1,2,3
                 или 0 - для финиша\n");
        scanf("%d", &vibor);
        if (vibor > 0 && vibor < 4)
            (*f_array [vibor-1])(vibor);
        else
         if (vibor == 0) break;
         else{
               printf("ошибка\n");
               vibor = 1;
             }
  }
  while (vibor > 0 && vibor < 4);
  printf("финиш\n");
 }
void f1(int a)
 {printf("Введен номер %d\n", a);}
void f2(int b)
 {printf("Введен номер %d\n", b);}
void f3(int c)
 {printf("Введен номер %d\n", c);}
```

Использование указателя на функцию в качестве параметра функции

Пример. ***Нахождение корня уравнения f(x)=0 на отрезке [a,b] методом деления отрезка пополам с заданной точностью.***

Напишем рекурсивную функцию для решения этой задачи.

```
#include <iostream>
#include <cmath>
using namespace std;
double recRoot(double (*f)(double),
                  double a,double b, double eps)
//double recRoot( double f(double),
//  double a, double b, double eps)
{   const double MyNull=1e-12;
    double fa=f(a), fb=f(b), c, fc;
    if (fa*fb>0)
    {   cout << "неверный отрезок\n";
        return -999.0;
    }
    c=(a+b)/2.0;
    fc=f(c);
    if ((fabs(fc)<=MyNull)||(b-a<=eps))
        return c;
    return
    (fa*fc<=0.0)?recRoot(f,a,c,eps):recRoot(f,c,b,eps);
}
int main()
 { setlocale(LC_ALL,".1251");
   double root,
   a=0.1, //левая граница отрезка
   b=3.5, //правая граница отрезка
   eps=1e-5; //точность вычислений
   double fun(double);//прототип функции
   root=recRoot(fun,a,b,eps);
   cout<<"\nкорень="<<root<<"\n";
   return 0;
 }
 double fun(double x) //  функция
 {   return(2.0/x*cos(x/2.0)); }
```

Прямая и косвенная рекурсия

Рекурсия бывает ***прямой*** и ***косвенной***. При косвенной рекурсии функция обращается к себе опосредовано, путем вызова другой функции, которая содержит обращение к первой.

КОНЕЦ ЛЕКЦИИ