Лекция 11

Функции (продолжение)

Рекурсивные функции

Понятие рекурсии затруднено даже для программистов с опытом. Для вычислительных задач более эффективной, чем рекурсия, оказывается итерация. Как мы знаем, итерация – это повторяемое выполнение процесса до тех пор, пока не будет удовлетворено некоторое условие (while или do while). Рекурсивное описание алгоритмов обработки и их программная реализация, как правило, не экономичны – они требуют много памяти и времени. Следует избегать рекурсии там, где есть очевидное итерационное решение. Но для сложной структуры – рекурсия предпочтительней.

Рекурсия – это использование рекурсивных определений. Рекурсивное определение задает элементы множества с помощью других элементов этого же множества.

Рекурсивная задача разбивается на этапы. Для ее решения вызывается рекурсивная функция, которая знает, как решать только простейшую часть задачи – базовую задачу. Если эта функция вызывается для решения базовой задачи, то она просто вызывает результат. Если функция вызывается для выполнения более сложной задачи, то она делит эту задачу на две части: одну часть, которую функция умеет решать, и другую, которую функция решать не умеет. Чтобы сделать рекурсию выполнимой, последняя часть должна быть похожа на исходную задачу, но быть по сравнению с ней несколько проще или несколько меньше. Эта новая задача подобна исходной, поэтому функция вызывает новую копию самой себя, чтобы начать работать над меньшей проблемой. Это называется **шагом рекурсии**. Шаг рекурсии содержит ключевое слово return, так как в дальнейшем результат шага будет объединен с той частью задачи, которую функция умеет решать, и сформируется конечный результат, который будет передан в исходное место вызова, возможно в main.

***Рекурсивная функция* – функция, содержащая вызовы самой себя. Выполнение рекурсивной функции происходит в два этапа: первый этап завершается выполнением так называемого условия завершения рекурсии; второй этап – вычисление действий на основе рекурсивных соотношений.**

На первом этапе осуществляется построение рекурсивных соотношений и частичное вычисление (с задержкой выполнения действий, которые не могут быть выполнены на

данном отрезке). Задержка вычислений обуславливается тем, что в описании рекурсивной функции всегда присутствует обращение к той же самой функции.

Выделение таких фрагментов памяти происходит и при каждом вызове очередного «поколения» рекурсивной функции.

При вызове рекурсивной функции с некоторым значением аргумента ее выполнение не завершается до конца, **а порождает новый вызов с другим значением аргумента**. Каждый рекурсивный вызов порождает новое «поколение» рекурсивной функции, сопровождаемое, в частности, выделением стековой памяти под параметры-значения и локальные переменные. Любой локальной переменной X на разных уровнях рекурсии будут соответствовать различные ячейки памяти, которые могут иметь различные значения. Поэтому выполнение вычислений с большой глубиной рекурсии (числом поколений) нередко приводит к нехватке этой памяти (сегмента стека).

Формы рекурсивных функций

Определены следующие формы рекурсивных функций:

- **форма с выполнением действий на рекурсивном спуске** (до рекурсивного вызова);
- **форма с выполнением действий на рекурсивном возврате** (после рекурсивного вызова);
- **форма с выполнением действий, как на рекурсивном спуске, так и на рекурсивном возврате** (как до, так и после рекурсивного вызова).

Например, структура рекурсивной функции может иметь формы:

С выполнением действий на рекурсивном спуске:

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

С выполнением действий на рекурсивном возврате:

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

С выполнением действий как на рекурсивном спуске, так и на рекурсивном возврате:

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

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

Приведенная ниже рекурсивная функция работает бесконечно и бесконечно печатает известные строки (это форма с выполнением действий на рекурсивном спуске):

```
#include <conio.h>
void PopeAndDoge1();
int main ()
```

```
{
 PopeAndDoge1();
   _getch();
   return 0 ;
}
void PopeAndDoge1()
{
cout << " Y popa byla sobaka, on ee lubil."<< endl;
cout << "Ona s'ela kysok mjasa, on ee ubil."<< endl;
cout << " poxoronil i nadpic napical:"<< endl;
PopeAndDoge1();
}
```

Однако если оператор вызова функции поставить перед выводом текста (получить форму с выполнением действий на рекурсивном возврате), то программа ничего не напечатает, хотя будет работать также бесконечно, как и первая.

```
#include <conio.h>
void PopeAndDoge1();
int main ()
{
PopeAndDoge1();
  _getch();
  return 0 ;
}
void PopeAndDoge1()
{
PopeAndDoge1();
cout << " Y popa byla sobaka, on ee lubil."<< endl;
cout << "Ona s'ela kysok mjasa, on ee ubil."<< endl;
cout << " poxoronil i nadpic napical:"<< endl;
 }
```

В конце концов, работа и той и другой программ приведет к переполнению стека и возникновению ошибки времени выполнения.

Условия «правильного» рекурсивного определения:

- множество определяемых объектов частично упорядочено;
- каждая убывающая по этому упорядочению последовательность элементов заканчивается некоторым минимальным элементом;
- минимальные элементы определяются нерекурсивно; характерное свойство описания рекурсивной функции состоит в том, что в ней должна содержаться проверка значений аргументов для принятия решения о завершении функции;
- неминимальные элементы определяются с помощью элементов, которые меньше их по этому упорядочению.

Виды рекурсий

Существуют два вида рекурсий. **Прямая рекурсия** – функция содержит вызовы самой себя. **Косвенная рекурсия** – кольцо – функция обращается к себе через вызов в другой функции. **Глубина рекурсии** вызова функции – максимальное количество незаконченных вызовов. **Текущий уровень рекурсии** – число рекурсивных вызовов в данный момент времени. **Общее количество вызовов** – количество вызовов всего, порожденных вызовом рекурсивной функции. Следует учитывать одну особенность, характерную для рекурсивных программ. Например, как было вычислено, для нахождения n-го числа Фибоначчи количество рекурсивных вызовов будет порядка 2n. Это называется

экспоненциальной сложностью. Поэтому при написании программ рекурсию используйте, только убедившись, что она не вызовет очень большое число рекурсивных вызовов.

Рекурсивная функция вычисления факториала

Рекурсивное определение факториала:
n! = n(n-1)!, если n>0;
n! = 1, если n=0.

Рассмотрим программу, выполняющую вычисление факториала на рекурсивном возврате.

```
double  Rec_Fact_Up (int);                          // прототип функции
int main()
{ int i=5;
  double Fact;
  Fact= Rec_Fact_Up (i);                             //вызов функции
  cout << i << "!=   " <<  Fact << endl;
  _getch();
  return 0;
}
double  Rec_Fact_Up (int n)                         //определение функции
{
if (n<=1) return 1.0;
else return  Rec_Fact_Up(n-1) * n;  //вычисление на рекурсивном возврате
                                    //можно представить состоящим из двух действий
              // Mult=Rec_Fact_Up(n-1) - непосредственно рекурсивный вызов;
              // Mult *n - оператор накопления факториала
}
```

Таблица трассировки программы Rec_Fact_Up по уровням рекурсии выглядит следующим образом:

Текущий уровень рекурсии	Рекурсивный спуск	Рекурсивный возврат
0	Rec_Fact_Up(5) n==5	Вывод n! == 120
1	Mult=Rec_Fact_Up(4); n==4	Mult * 5 = =120;
2	Mult=Rec_Fact_Up(3); n==3	Mult * 4 == 24;
3	Mult=Rec_Fact_Up(2); n==2	Mult * 3 == 6;
4	Mult=Rec_Fact_Up(1); n==1	Mult * 2 == 2;
5	Mult=1;	Mult==1;

Многие задачи (в том числе и вычисление факториала) безразличны к форме используемой рекурсивной подпрограммы, однако есть классы задач, при решении которых программисту требуется сознательно управлять ходом работы рекурсивных функций.

Рассмотрим выполнение действий на рекурсивном спуске на примере все того же алгоритма вычисления факториала.

Введем в рекурсивную функцию дополнительно два параметра: *Mult* – для выполнения на спуске операции умножения накапливаемого значения факториала на очередной множитель; *m* - для обеспечения независимости рекурсивной функции от

имени конкретной глобальной переменной, то есть для повышения универсальности функции:

```
double  Rec_Fact_Dn (double, int, int);              // прототип функции
int main()
{ int i=5, n=i;
  double Fact;
  Fact= Rec_Fact_Dn (1.0, 1, n);                 //вызов функции
  cout <<i << "!  =  " <<  Fact << endl;
  _getch();
  return 0;
}

double  Rec_Fact_Dn(double Mult, int i, int m)
//определение функции
{
  Mult=Mult*i;
  if (i==m) return Mult; else return  Rec_Fact_Dn (Mult, i+1, m);
  }
```

Таблица трассировки значений параметров рекурсивной функции Rec_Fact_Dn по уровням рекурсии:

Текущий уровень рекурсии	Рекурсивный спуск			Рекурсивный возврат
0	Rec_Fact_Dn(1.0,1,5);			Вывод n! = 120
1	Mult= Mult *1 =1;	i ==1;	Rec_Fact_Dn(1.0,2,5);	Mult*5= 120;
2	Mult= Mult *2 = 2;	i ==2;	Rec_Fact_Dn(2.0,3,5);	Mult*5= 120;
3	Mult= Mult *3 =6;	i ==3;	Rec_Fact_Dn(6.0,4,5);	Mult*5= 120;
4	Mult= Mult*4 =24;	i ==4;	Rec_Fact_Dn(24.0,5,5);	Mult*5= 120;
5	Mult= Mult*5= 120;	i==5;	Rec_Fact_Dn=120.0;	Mult*5= 120;

Рекурсивная функция вывода на печать символов строки в обратном порядке

Рассмотрим вывод на печать символов введенной строки ‘HELLO’ в обратном порядке:

```
void  Reverse ();              // прототип функции
int main()
{ cout << "Input string:\n";
  Reverse();
  cout << endl;
  _getch();
  return 0;
}
void  Reverse ()
{int ch;
if (( ch= _getche())!='\r')
                  {Reverse(); _putch(ch);
                   }
  }
```

Таблица трассировки данной программы по уровням рекурсии выглядит следующим образом:

Текущий уровень рекурсии	*Рекурсивный спуск*	*Рекурсивный возврат*
0	Reverse;	
1	Ввод: ‘H’; ch !=’\n’; Reverse;	Вывод: ‘H’
2	Ввод: ‘E’; ch !=’\n’; Reverse;	Вывод: ‘E’
3	Ввод: ‘L’; ch !=’\n’; Reverse;	Вывод: ‘L’
4	Ввод: ‘L’; ch !=’\n’; Reverse;	Вывод: ‘L’
5	Ввод: ‘O’; ch !=’\n’; Reverse;	Вывод: ‘O’
6	Ввод ‘\n’; ch ==’\n’;	

Рекурсивная функция возведения вещественного числа X в целую степень N>=0

Программа реализует алгоритм возведения вещественного числа X в целую степень N>=0 за минимальное число операций умножения:

```
double rec_degree(double, int );
int main ()
{ double x, y;
  int  n;
  cout << " Input (X<= 10)   Input  (-90<=N<=90) ?: " << endl ;
  cin >> x >> n ;
  y=rec_degree(x, abs(n));
  cout <<x  << "   " << n << "  "   <<  ( n>0 ? y : 1/y) << endl;
_getch();
return 0 ;
}

double rec_degree(double x,  int n)
{ double r;
if  ( !n)  return 1;          //действия выполняются на рекурсивном возврате
if ( !( n% 2)) return r=rec_degree(x, n/2), r*r ;          //n - четное
                  else return x *rec_degree(x, n-1);       //n - нечетное
}
```

Рекурсивная функция печати числа в виде строки символов

Рекурсивная функция printd вызывает себя, чтобы напечатать все старшие разряды заданного числа, а затем печатает цифру последнего разряда.

```
void printd (int);
int main ()
{ int a=12345;
  printd(a) ;
  cout <<  endl ;
_getch();
return 0 ;
}
void printd (int n)
{ if  (n < 0)   {putchar ('-');
                        n=-n;
                       }
if ( n)
{printd (n/10);          //действия выполняются на рекурсивном возврате
  putchar (n%10 +'0');
}
}
```

Вычисление НОД через итерации и через рекурсивную функцию

```
int  NOD_iter (int, int);
 int  NOD_rec (int,  int);
int main ()
{  int m=32, n=16;
  cout << m <<"  " << n << "   "  <<  NOD_iter ( m, n) << endl ;
  cout << m <<"  " << n << "   "  <<  NOD_rec ( m, n) << endl ;
  _getch();
  return 0 ;
}

int  NOD_iter (int m, int n)
{ int r;
  do  {r= m % n;
       m=n;
       n=r;
  }while (r);
  return m;
}

int  NOD_rec (int m,  int n)
{     if (!(m % n))  return n;    //действия выполняются на рекурсивном спуске
     else  return NOD_rec(n, m %n);
}
```

Рекурсивная функция вычисления чисел Фибоначчи

```
unsigned long fib (unsigned long);
int main ()
{ unsigned long number;
   cout << " Input number:" << endl;
   cin >> number;
   cout << "Fib number (" << number << ") = " << fib (number) << endl ;
   _getch();
   return 0 ;
}

unsigned long fib (unsigned long n)
{
      if (n==0 || n==1) return n;
else return  fib(n-1)+fib(n-2); //действия выполняются на рекурсивном
возврате
}
```

Рекурсивная функция вычисления суммы элементов массива

```
double sum (int [],  int);
const int con =5;
int main ()
{
 int array[con]= {1,2,3,4,5};
 for (int i=0; i<con; i++)
      cout << setw(3) << array[i];
cout <<endl;
cout << sum (array, con) << endl ;
 _getch();
 return 0 ;
}
```

```
double sum (int s[], const int n)
{
 if (n==1) return s[0];
 else return sum(s, n-1) + s[n-1];
                 //действия выполняются на рекурсивном возврате
}
```

Указатель на функцию

Указатель на функцию используется для передачи имени функции в качестве параметра в другую функцию, а также для косвенного вызова функции.

Имя функции в С++ – это указатель-константа на функцию, равный адресу точки входа функции (адресу первой машинной команды). Помимо констант, возможно описание и указателей-переменных на функции:

return_type (*name) (arg_list);

где return_type – тип возвращаемого функцией значения; name – имя переменной-указателя на функцию; arg_list – список типов аргументов, передаваемых функции при ее вызове по значению указателя.

Указатели на функцию используются в случаях, перечисленных ниже:

1. Многие библиотечные функции в качестве аргумента получают указатель на функцию. Например, функция сортировки qsort() получает четвертым аргументом указатель, на составленную пользователем функцию, выполняющую сравнение сортируемых элементов.
2. Использование указателей на функцию в качестве аргументов позволяет разрабатывать универсальные функции, например численного интегрирования и т.д.

Рассмотрим программу, использующую указатель на функцию для доступа к функциям difference() и sum():

```
int dif (int, int);   //прототипы функций
int sum(int, int);

int main()
{
  int (*func_ptr) (int, int);              //определение указателя на функцию
  int var1=20, var2=5, ret;

  func_ptr=dif;                          //инициализация указателя на функцию
  ret=(*func_ptr)(var1, var2);           //вызов функции dif
  cout << ret << endl;

 func_ptr=sum;                         //инициализация указателя на функцию
  ret=(*func_ptr)(var1, var2);         //вызов функции sum
  cout << ret << endl;
  _getch();
  return 0;
}
int dif(int a, int b)
{return a-b;}

int sum(int a, int b)
{return a+b;}
```

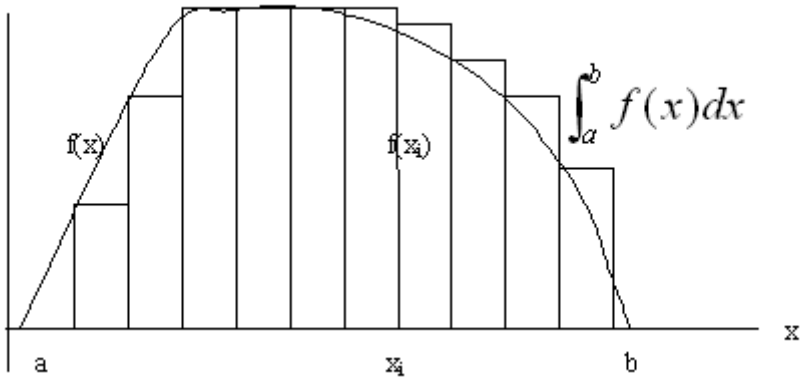
Обобщенные алгоритмы

Нередко алгоритмы описываются как некоторые *«обобщенные» алгоритмы*, применяемые не только к разным данным, но и инвариантные относительно разных действий. Обобщенные алгоритмы описываются в виде подпрограмм, имеющих в качестве параметров функции (в C++ такие параметры являются указателями на функцию). При обращении к таким подпрограммам в качестве аргумента должно быть задано имя функции, имеющей ту же сигнатуру (количество и тип параметров), что и формальный параметр. В качестве примера обобщенного алгоритма рассмотрим вычисление интеграла.

Вычисление интеграла по методу левых прямоугольников

На отрезке [a, b] с заданной точностью eps методом левых прямоугольников вычислить интегралы двух функций:

$$\int_a^b \frac{\cos(x)}{\sqrt[3]{x}}dx \qquad \int_a^b \frac{\cos(x^3)}{\sqrt[4]{x}}dx$$



Вычислить интеграл для заданной функции f(x) на заданном отрезке [a, b] – это значит вычислить площадь фигуры, ограниченной кривой f(x) и осью x. Самый простой способ – метод левых прямоугольников – предполагает разбиение отрезка [a, b] на n отрезков длины h и вычисление требуемой площади фигуры как суммы площадей полученных прямоугольников $f(x_i)* h$.

Вычисление интеграла предполагает последовательное выполнение действий:

1. вычисление суммы S1 площадей прямоугольников с шагом h;
2. вычисление суммы S2 площадей прямоугольников с шагом h/2;
3. если |S1 - S2| >= eps, то S1=S2; и переход на выполнение очередной итерации (выполнение п.2 с новым шагом); завершение вычислений – при |S1 - S2| < eps.

Мы представим весь процесс вычисления интеграла в виде последовательных вызовов функций:

функция main() обеспечивает: ввод-вывод данных и вызов функции двойного пересчета, которой в качестве параметров передаются **границы отрезка интегрирования**, требуемая **точность вычисления** интеграла, **подынтегральная**

функция, две **переменные (для вычисленного значения интеграла** и **количества** совершаемых при вычислении **итераций);**

Процедура вычисления интеграла по критерию двойного пересчета предполагает реализацию п.3. (см. выше алгоритм вычисления интеграла) и будет представлена в виде функции **Vych_Int_lpram,** которой передается набор соответствующих параметров, и которая имеет прототип:

void **Vych_Int_lpram**(double **a**, double **b**, double **eps**, double (***pf**)(double), double ***I**, int ***k**);

Здесь **a**,**b** – отрезок интегрирования; **eps** – требуемая точность вычислений; **pf** – указатель-переменная на функцию заданной сигнатуры и возвращаемого значения; I и k – переменные, предназначенные для получения из функции результата (I – значение интеграла, k – количество итераций, потребовавшихся для достижения заданной точности вычисления).

В свою очередь, функция **Vych_Int_lpram** вызывает функцию **Sum**, вычисляющую **одноразово** требуемую площадь по методу левых прямоугольников. При этом функции передается набор следующих параметров:

```
void Sum(double a, double b, double h, double (*pf) (double), double *S);
```

//прототип функции вычисления суммы по методу левых прямоугольников

Здесь **a**,**b** – отрезок интегрирования; **h** – шаг интегрирования; **pf** – указатель-переменная на функцию заданной сигнатуры и возвращаемого значения; S – переменная, предназначенная для получения из функции значения площади.

Чтобы вычислить значение подынтегральной функции f(x) в требуемой точке $\mathbf{x_i}$, функция **Sum** организует вызов подынтегральной функции, передавая ей в качестве параметра значение этой точки. Прототип подынтегральной функции выглядит так:

```
double  f1 (double);
```

Запишем теперь текст программы:

```
        //прототип функции вычисления интеграла по критерию двойного пересчета
void Vych_Int_lpram(double a, double b, double eps, double (*pf) (double),
double *I,  int *k);
           //прототип функции вычисления суммы по методу левых прямоугольников
void Sum (double a, double b, double h, double (*pf) (double), double *S);
double  f1 (double);                    //прототипы подынтегральных функций
double  f2 (double);

int main ()
{ double  a, b, eps;                //отрезок интегрирования,  точность
  double Int;          //переменная, в которой возвращается значение интеграла
  int K_iter;          //переменная, в которой возвращается количество итераций

  cout << "Input a, b, eps\n";
  cin >> a >> b >> eps;

  Vych_Int_lpram(a, b, eps, &f1, &Int, &K_iter);  //вычисление интеграла для f1
                             // или  Vych_Int_lpram(a, b, eps, f1, &Int, &K_iter);

  cout << "Integral    for f1  =" << Int << "    K_iter= "<< K_iter << endl;

  Vych_Int_lpram(a, b, eps, &f2, &Int, &K_iter);   //вычисление интеграла для f2
                            // или  Vych_Int_lpram(a, b, eps, f2, &Int, &K_iter);

 cout << "Integral  for f2  =" << Int << "   K_iter = " << K_iter << endl;
 _getch();
 return 0;
}
```

```
void Vych_Int_lpram(double a, double b, double eps, double (*pf) (double),
                                                    double *I,  int *k)
 {int n=20;                 //инициализация начального количества разбиений
 double h=(b-a)/n;          //определение шага интегрирования
 double S1=0, S2=0;         //переменные для значений сумм с шагом h и с шагом h/2

                             //вызов функции Sum1 с шагом h: в S1 возвращается сумма
 Sum (a, b, h, (*pf), &S1);  //вызов функции для запуска процесса двойного пересчета
                            //или Sum (a, b, h, pf, &S1);
 *k=0;
 do
    { S2= S1;
      n *=2;                //увеличение количества отрезков разбиения и
      h=(b-a)/n;            //уменьшение шага интегрирования в 2 раза
                                  //вызов функции Sum1 с шагом h=h/2
      Sum (a, b, h, pf, &S1);     //повторное вычисление суммы в S1
                              //или Sum (a, b, h, (*pf), &S1);
      *k +=1;
 }while (fabs(S1-S2) > eps) ;
 *I=S1;
}

void Sum(double a, double b, double h, double (*pf) (double), double *S)
 { double x, sum;
   x=a;
   sum=0;
   while (x<b)
           {
             sum=sum+(*pf)(x);    //накопление суммы высот
                                  //или sum=sum+pf(x);
             x=x+h;
    }
    *S=h*sum;                     //вычисление площади
   }

 double  f1 (double x)
{return cos(x)/exp(0.333333*log(x));}

 double  f2 (double x)
{return  cos(x*x*x)/sqrt(sqrt(x)); }
```

Если в функции main определить переменную типа «указатель на функцию», то вызов функций будет возможно организовать иначе. Для улучшения читаемости программы с помощью ключевого слова **typedef**[1] зададим синоним для описания типа «указатель на функцию»:

```
typedef double (*pfunc) (double);//pfunc–синоним типа «указатель на функцию»
```

Тогда имеем:

```
typedef  double (*pfunc) (double x); //pfunc - синоним типа «указатель на функцию»
```

---

[1] Пример использования typedef: **объявить массив из N указателей на функцию, возвращающую указатель на функцию, возвращающую указатель на char:** char *(*(*a[N])()) ();

```
typedef char *pc;          // pc – указатель на char
typedef pc  fpc();         // fpc – функция, возвращающая  pc (указатель на char)
typedef fpc *pfpc;         // pfpc – указатель на fpc (функцию, возвращающую указатель на char)
typedef pfpc fpfpc();      // fpfpc – функция, возвращающая pfpc (указатель на функцию, возвращающую указатель на char)
typedef fpfpc *pfpfpc;     // pfpfpc – указатель на функцию, возвращающую указатель на функцию, возвращающую указатель на char
pfpfpc  a[N];              // определяем массив из N элементов типа pfpfpc
```

```
                    //прототип функции вычисления интеграла по критерию двойного пересчета
void Vych_Int_lpram(double a,double b,double eps, pfunc pf,double *I,int *k);
                   //прототип функции вычисления суммы по методу левых прямоугольников
void Sum (double a, double b, double h, pfunc pf, double *S);

double  f1 (double);                        //прототипы подынтегральных функций
double  f2 (double);

int main ()
{ double  a, b, eps;                 //отрезок интегрирования,  точность
  double Int;               //переменная, в которой возвращается значение интеграла
  int K_iter;               //переменная, в которой возвращается количество итераций

  cout << "Input a, b, eps\n";
  cin >> a >> b >> eps;
  pfunc  pf=f1;              //определение указателя на функцию pf и его инициализация
                             //или         pfunc  pf=&f1;

  Vych_Int_lpram(a, b, eps, (*pf), &Int, &K_iter);   //вычисление интеграла для
f1
                        // или  Vych_Int_lpram(a, b, eps, pf, &Int, &K_iter);

 cout << "Integral    for f1  =" << Int << "    K_iter= "<< K_iter << endl;
 pf=f2;                   // присваивание указателю на функцию нового значения
                          //или         pf=&f2;

  Vych_Int_lpram(a, b, eps, (*pf), &Int, &K_iter); //вычисление интеграла для f2
                       //или  Vych_Int_lpram(a, b, eps, pf, &Int, &K_iter);

 cout << "Integral for f2  =" << Int << "   K_iter = " << K_iter << endl;
 _getch();
 return 0;
}

void Vych_Int_lpram(double a,double b,double eps, pfunc pf, double *I,int *k)
 {int n=20;                  //инициализация начального количества разбиений
 double h=(b-a)/n;            //определение шага интегрирования
 double S1=0, S2=0;       // переменные для значений сумм с шагом h и с шагом h/2

                           //вызов функции Sum1 с шагом h: в S1 возвращается сумма
 Sum (a, b, h, (*pf), &S1); //вызов функции для запуска процесса двойного пересчета
                            //или Sum (a, b, h, pf, &S1);
 *k=0;
 do
   { S2= S1;
    n *=2;                    //увеличение количества отрезков разбиения и
    h=(b-a)/n;                       //уменьшение шага интегрирования в 2 раза
                                     //вызов функции Sum1 с шагом h=h/2
    Sum (a, b, h, (*pf), &S1);         //повторное вычисление суммы в S1
                                     //или Sum (a, b, h, pf, &S1);
   *k +=1;
 }while (fabs(S1-S2) > eps) ;
 *I=S1;
}

void Sum(double a, double b, double h, pfunc pf, double *S)
 { double x, sum;
    x=a;
    sum=0;
    while (x<b) {
             sum=sum+(*pf)(x);     //накопление суммы высот
                                   //или sum=sum+pf(x);
             x=x+h;
```

```
     }
    *S=h*sum;                    //вычисление площади
   }

 double  f1 (double x)
{ return cos(x)/exp(0.333333*log(x));}

 double  f2 (double x)
{return  cos(x*x*x)/sqrt(sqrt(x)); }
```

Как мы видим в рассмотренных случаях, в вызове функции **Vych_Int_lpram** всегда присутствует переменная **pf**, но содержит она разные функции.

Вычисление интеграла по методу правых прямоугольников

Рассмотрим вычисление интеграла **по методу правых прямоугольников (**функция **Vych_Int_rpram).**

Организуем иначе и саму процедуру вычисления интеграла. Откажемся от выделения процедуры одноразового вычисления площади в виде отдельной функции Sum и «растворим» эту процедуру в теле функции **Vych_Int_rpram,** наряду с процедурой двойного пересчета**.** Оптимизируем процесс вычисления площади. Проанализируйте приведенную ниже функцию и ответьте, за счет чего здесь происходит оптимизация:

```
// функция вычисления интеграла по методу правых прямоугольников
// с элементами оптимизации вычисления суммы
 void Vych_Int_rpram(double a,double b,double eps,pfunc pf,double *I, int *k)
{double h, S, S1, sum=(*pf)(b);
 int  n=1, m=0;
*k=0;
 S=sum* (b-a);     // очень «грубое» вычисление интеграла для входа в цикл
do
   {S1= S;
    * k +=1;                //количество итераций
    n=n*2; h=(b-a)/n;       // увеличение количества отрезков разбиения и
                             // уменьшение шага интегрирования
    int m=1;
    while (m<n )
           {sum=sum+(*pf)(b-h*m); // накопление суммы высот
                    // без повторного вычисления высот в «старых» точках!!!
            m=m+2;
    }
    S=h*sum;                     // вычисление площади
}while (fabs(S-S1)>eps);
*I=S;
}
```

Использование одинаковых вызовов функции для вычисления интеграла разными методами и для разных функций

Зададим синоним для описания типа «указатель на функцию, **тип возвращаемого значения и сигнатура которой** соответствуют **типу возвращаемого значения и сигнатуре функции вычисления интеграла по заданному методу**:

```
typedef  void (*pfunc_metod)(double,double,double, pfunc, double *, int *);
```

Функция main тогда может иметь вид:

```
          // pfunc – синоним типа «указатель на функцию (под интегралом)»
typedef  double (*pfunc) (double x);
```

```
                  // pfunc_metod – синоним типа «указатель на функцию-метод»
typedef  void (*pfunc_metod)(double,double,double, pfunc, double *, int *);
           //прототип функции вычисления интеграла по критерию двойного пересчета
void Vych_Int_lpram(double a, double b, double eps, pfunc pf, double *I,  int
*k);
         //прототип функции вычисления суммы по методу левых прямоугольников
void Vych_Int_rpram(double a,double b,double eps,pfunc pf,double *I, int *k);
        //прототип функции вычисления суммы по методу правых прямоугольников
void Sum (double a, double b, double h, pfunc pf, double *S);
double  f1 (double);              //прототипы функций «под интегралом»
double  f2 (double);

int main ()
{ double  a, b, eps;      //отрезок интегрирования, точность, значение суммы
   double Int;        //переменная, в которой возвращается значение интеграла
  int K_iter;         //переменная, в которой возвращается количество итераций
  pfunc  pf=f1;       // инициализация указателя адресом функции  f1
  pfunc_metod   pmet = Vych_Int_lpram;
                      // инициализация указателя адресом функции Vych_Int_lpram
  cout << "Input a, b, eps\n";
  cin >> a >> b >> eps;
  (*pmet) (a,b,eps,(*pf),&Int,&K_iter); //вызов Vych_Int_lpram  для f1
  cout << "lpram    for f1  =" << Int << "    K_iter= "<< K_iter << endl;

  pf=f2;                      // присваивание указателю на функцию значения f2
  (*pmet) (a,b,eps,(*pf),&Int,&K_iter); //вызов Vych_Int_lpram  для f2
  cout << "lpram    for f2  =" << Int << "     K_iter = " << K_iter << endl;

  pmet =Vych_Int_rpram;  // указателю присваивается адрес новой функции-метода
(*pmet)(a, b, eps,(*pf),&Int,&K_iter); //вызов Vych_Int_rpram для f2
  cout << "rpram    for f2  =" << Int << "     K_iter = " << K_iter << endl;

  pf=f1;                // инициализация указателя на функцию значением f1
  (*pmet) (a, b, eps,(*pf), &Int, &K_iter); //вызов Vych_Int_rpram для f1
  cout << "rpram    for f1  =" << Int << "     K_iter = " << K_iter << endl;
_getch();
return 0;
}

void Vych_Int_lpram(double a, double b, double eps, pfunc pf, double *I,  int
*k)
 {int n=20;               //инициализация начального количества разбиений
 double h=(b-a)/n;        //определение шага интегрирования
 double S1=0, S2=0;       // переменные для значений сумм с шагом h и с шагом h/2

                         //вызов функции Sum1 с шагом h: в S1 возвращается сумма
 Sum (a, b, h, (*pf), &S1);  //вызов функции для запуска процесса двойного пересчета
                          //или Sum (a, b, h, pf, &S1);
 *k=0;
 do
 { S2= S1;
   n *=2;               //увеличение количества отрезков разбиения и
   h=(b-a)/n;           //уменьшение шага интегрирования в 2 раза
                           //вызов функции Sum1 с шагом h=h/2
   Sum (a, b, h, (*pf), &S1);         //повторное вычисление суммы в S1
                            //или Sum (a, b, h, pf, &S1);
  *k +=1;
 }while (fabs(S1-S2) > eps) ;
 *I=S1;
}

void Vych_Int_rpram(double a, double b, double eps, pfunc pf, double *I,  int
*k)
```

```
{double h, S, S1, sum=(*pf)(b);
 int  n=1, m=0;
*k=0;
 S=sum* (b-a);              // очень «грубое» вычисление интеграла для входа
в цикл
do
        {S1= S;
         * k +=1;                          //количество итераций
      n=n*2; h=(b-a)/n;      // увеличение количества отрезков разбиения и
                             // уменьшение шага интегрирования
      int m=1;
           while (m<n )
            {sum=sum+(*pf)(b-h*m); // накопление суммы высот
             m=m+2;
      }
      S=h*sum;                                              // вычисление площади
}while (fabs(S-S1)>eps);
*I=S;
}

void Sum(double a, double b, double h, pfunc pf, double *S)
 { double x, sum;
    x=a;
    sum=0;
    while (x<b) {
            sum=sum+(*pf)(x);    //накопление суммы высот
                                 //или sum=sum+pf(x);
            x=x+h;
     }
    *S=h*sum;                                       //вычисление площади
  }

 double  f1 (double x)
{ return cos(x)/exp(0.333333*log(x));
}
 double  f2 (double x)
{return  cos(x*x*x)/sqrt(sqrt(x));
}
```

Использование при вычислении интеграла многофайловой компиляции

```
// integral.h:
        // pfunc – синоним типа «указатель на функцию (под интегралом)»
typedef  double (*pfunc) (double x);
       // pfunc_metod – синоним типа «указатель на функцию-метод»
typedef void (*pfunc_metod)(double, double, double, pfunc, double *, int *);
         //прототип функции вычисления интеграла по критерию двойного пересчета
void Vych_Int_lpram(double a,double b,double eps,pfunc pf,double *I, int *k);
        //прототип функции вычисления суммы по методу левых прямоугольников
void Vych_Int_rpram(double a,double b,double eps,pfunc pf,double *I,int *k);
        //прототип функции вычисления суммы по методу правых прямоугольников
void Sum (double a, double b, double h, pfunc pf, double *S);
double  f1 (double);       //прототипы подынтегральных функций
double  f2 (double);

// stdafx.h :
#pragma once

#define WIN32_LEAN_AND_MEAN
#include <stdio.h>
#include <tchar.h>
```

```
// TODO: reference additional headers your program requires here
#include <iostream>
#include <conio.h>
#include <math.h>
#include <iomanip>
#include "integral.h"
using namespace std;

// f_int.cpp :
#include "stdafx.h"

double  f1 (double x)
{ return cos(x)/exp(0.333333*log(x));}

 double  f2 (double x)
{return  cos(x*x*x)/sqrt(sqrt(x)); }



// f_metod_int.cpp :
#include "stdafx.h"


int main ()
{ double  a, b, eps;
  double Int;
  int K_iter;
  pfunc  pf=f1;
  pfunc_metod   pmet =Vych_Int_lpram;

  cout << "Input a, b, eps\n";
  cin >> a >> b >> eps;
  (*pmet) (a, b, eps, pf, &Int, &K_iter);  // Vych_Int_lpram  для f1
  cout << "lpram    for f1  =" << Int << "    K_iter= "<< K_iter << endl;

  pf=f2;
  (*pmet) (a, b, eps, pf, &Int, &K_iter); // Vych_Int_lpram  для f2
  cout << "lpram    for f2  =" << Int << "     K_iter = " << K_iter << endl;

 pmet =Vych_Int_rpram;
(*pmet) (a, b, eps, pf, &Int, &K_iter); // Vych_Int_rpram для f2
  cout << "rpram    for f2  =" << Int << "     K_iter = " << K_iter << endl;

pf=f1;
  (*pmet) (a, b, eps, pf, &Int, &K_iter); // Vych_Int_rpram для f1
  cout << "rpram    for f1  =" << Int << "     K_iter = " << K_iter << endl;
_getch();
return 0;
}


void Vych_Int_lpram(double a, double b, double eps, pfunc pf, double *I,  int
*k)
 {int n=20;
 double h=(b-a)/n;
 double S1=0, S2=0;
 Sum (a, b, h, pf, &S1);
*k=0;
 do
 { S2= S1;
   n *=2;
   h=(b-a)/n;
```

```
   Sum (a, b, h, (*pf), &S1);
   *k +=1;
 }while (fabs(S1-S2) > eps) ;
 *I=S1;
}

void Vych_Int_rpram(double a, double b,double eps, pfunc pf,double *I,int *k)
{double h, S, S1, sum=(*pf)(b);
 int  n=1, m=0;
*k=0;
 S=sum* (b-a);
do
      {S1= S;
       * k +=1;
      n=n*2; h=(b-a)/n;
      int m=1;
      while (m<n )
            {sum=sum+(*pf)(b-h*m);
             m=m+2;
      }
      S=h*sum;
}while (fabs(S-S1)>eps);
*I=S;
}

void Sum(double a, double b, double h, pfunc pf, double *S)
 { double x, sum;
    x=a;
    sum=0;
    while (x<b) {
             sum=sum+(*pf)(x);
             x=x+h;
     }
    *S=h*sum;
   }

// stdafx.cpp :
#include "stdafx.h"
```