№

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ

Технологический институт

Федерального государственного образовательного учреждения высшего профессионального образования

“Южный федеральный университет”

ПРИОРИТЕТНЫЙ НАЦИОНАЛЬНЫЙ ПРОЕКТ “ОБРАЗОВАНИЕ”

(2006-2007 гг.)

Работа с подпрограммами в Си и Паскале на ПК и в микроконтроллере

Учебно-методическое пособие

по курсу

ИНФОРМАТИКА

Для студентов специальностей
210106 Промышленная электроника и
230201 Информационные системы и технологии

РТФ

Таганрог 2007

КАФЕДРА МИКРОПРОЦЕССОРНЫХ СИСТЕМ

УДК

Составитель А.Б. Клевцова

Учебно-методическое пособие “Работа с подпрограммами в Си и Паскале на ПК и в микроконтроллере” по курсу “Информатика”. – Таганрог: Изд-во Технологического института ЮФУ, 2007. – 44 с.

В учебно-методическом пособии приводятся основные приемы работы с подпрограммами на языках программирования Си и Паскаль. Рассматриваются правила определения функций и процедур, а также механизмы их вызова для последующего взаимодействия. Ориентация сделана как на изложение синтаксиса и семантики конструкций языка, так и на практическое использование описанных конструкций при решении типовых задач программирования.

Библиогр.: 7 назв.

Рецензент П.П. Клименко, канд.техн.наук, доцент кафедры РТС Технологического института ЮФУ.

СОДЕРЖАНИЕ

1. ФУНКЦИИ В ЯЗЫКЕ ПРОГРАММИРОВАНИЯ СИ

1.1. Функции и их определение

Функция представляет собой самодостаточную единицу программного кода, разработанную для решения конкретной задачи. Функция в языке Си играет ту же роль, какую играют функции и процедуры в других языках программирования, хотя в деталях эти роли могут быть различными. В результате выполнения некоторых функций происходит то или иное событие. Например, в результате выполнения функции printf() на вашем экране появляются конкретные данные. Другие функции возвращают значения для их последующего использования в программе. Например, функция strlen() сообщает программе длину заданной строки. В общем случае функция может одновременно выполнять действия и возвращать значения.

Почему вы должны пользоваться функциями? Во-первых, они снимают с вас обязанность многократного повторения в программе одних и тех же кодовых последовательностей. Если в программе приходится решать одну и ту же задачу несколько раз, вам достаточно написать соответствующую функцию всего лишь один раз. Программа использует эту функцию там, где необходимо, а вы можете использовать одну и ту же функцию в нескольких программах. Даже в тех случаях, когда задача в программе решается всего лишь один раз, использование функции целесообразно, поскольку при этом увеличивается уровень модульности, благодаря чему программа становится более понятной при чтении, к тому же в нее легче вносить изменения и исправления.

Многие программисты предпочитают думать о функции как о “черном ящике”, представленном в терминах информации, которая поступает на вход этого ящика (ввод), и значением или действием, которое он производит (вывод). Вас не должно интересовать, что происходит внутри черного ящика, если, конечно, вы не программист, занимающийся разработкой этой функции. Например, когда вы используете функцию printf(), вы знаете что ей нужно передать управляющую строку и, возможно, некото-

рые аргументы. Вы также знаете, какой выход должна генерировать функция printf(). Вам не нужно знать программный код реализации функции printf(). Такой подход к использованию функций позволяет сосредоточить все усилия на создании общей структуры программы и не отвлекаться на отдельные детали.

Тщательно продумайте, что должна выполнять функция и какое место она занимает в программе, прежде чем приступать к написанию ее программного кода.

Что вы должны знать о функциях?

Прежде всего, вы должны знать, как их правильно определять, как их вызывать для последующего использования и как наладить их взаимодействие.

1.2. Функции без параметров

```
/* Программа, использующая функцию без параметров*/
#include <stdio.h>
void good(void);
void fine(void);
        int main(void)
        {
           good();
           good();
           fine();
           return 0;
         }
void good(void)
{
   printf("Наш институт очень хороший!\n");
}
void fine(void)
{
   printf("Наш институт лучше всех!\n");
}
```

Выходные данные будут иметь следующий вид:

Наш институт очень хороший!

Наш институт очень хороший!
Наш институт лучше всех!

Функция good() трижды появляется в рассматриваемой программе. В первый раз она появляется в виде прототипа void good(void), передающего компилятору информацию о функциях, которые будут использованы в данной программе. Во второй раз она появляется в основной функции main() в форме вызова функции good(). Причем вызов осуществляется дважды. И, наконец, в данной программе представлено определение функции, которое является исходным кодом самой функции:

```
void good(void)
{
   printf("Наш институт очень хороший!\n");
}
```

Прототип – это форма объявления, которая уведомляет компилятор о том, что вы используете конкретную функцию. Он также определяет свойства этой функции. Например, первое ключевое слово void в прототипе функции good() указывает на то, что функция good() не имеет возвращаемого значения. В общем случае, функция может возвратить значение в вызывающую функцию для последующего его использования, но функция good() этого не делает. Второе void, то что находится в скобках за именем функции good(void), означает, что функция не принимает аргументов.

Обратите внимание, что ключевое слово void употребляется в смысле “пусто”, а не в смысле “неправильно”.

Функция fine() также трижды появляется в рассматриваемой программе. В основной функции main() она вызывается один раз.

Как же работает программа?

Сначала в основной функции main() осуществляется первое обращение к функции good(). Программа переходит к выполнению этой функции. В результате на экране появляется сообщение:

Наш институт очень хороший!

Затем осуществляется возврат в основную функцию main(), в которой осуществляется повторный вызов функции good(). В результате работы этой функции на экране появляется второй сообщение:

Наш институт очень хороший!

После возврата в основную функцию main() происходит вызов функции fine(), в результате работы которой на экране появляется сообщение:

Наш институт лучше всех!

Еще один момент, на который следует обратить внимание, заключается в том, что на практике рекомендуется все программы начинать с выполнения функции main(), так как она играет роль базового каркаса.

Упражнения по программированию

1. Напишите программу, которая выдает следующие выходные данные:

Улыбайся! Улыбайся! Улыбайся!
Улыбайся! Улыбайся!
Улыбайся!

В программе должна быть определена функция, которая отображает строку *Улыбайся!* один раз, в то же время программа может использовать эту функцию столько раз, сколько надо.

2. Напишите программу, которая вызывает функцию с именем one_three(). Эта функция должна напечатать слово *один* в одной строке, вызвать функцию two(), а затем напечатать слово *три* в одной строке. Функция two() должна отобразить слово *два* в одной строке. Функция main() должна вывести фразу *начать сейчас:* перед вызовом функции one_three() и напечатать *порядок!* после ее вызова.

Таким образом выходные данные должны иметь следующий вид:

начать сейчас:
один
два
три

порядок!

1.3. Функции с параметрами

1.3.1. Функции, возвращающие одно целое значение

/* Программа, использующая функцию, возвращающую одно значение*/

```
#include <stdio.h>
#include <conio.h>
int sign(int a);
int main(void)
{
   int x,y,z;
   clrscr();
   printf("Введите x=");
   scanf("%d",&x);
   printf("Введите y=");
   scanf("%d",&y);
   z=sign(x)+sign(y)+sign(x+y);
   printf("z=%d",z);
   getch();
   return 0;
   }
 int sign(int a)
 {
 int ss;
 if (a<0)
 ss=-1;
 else
 if (a=0)
 ss=0;
 else
 ss=1;
 return ss;
 }
```

В данной программе вычисляется выражение: z(x)=sign(x)+sign(y)+sign(x+y).

Переменные x и y вводятся с клавиатуры. Функция sign() определяется следующим выражением:

$$\text{sign(a)}=\begin{cases} -1, если & a<0 \\ 0, если & a=0 \\ 1, если & a>0 \end{cases}$$

Определение функции начинается с описания ее прототипа: int sign(int a). Эта строка сообщает компилятору, что функция sign() имеет один аргумент целого типа int a. Этот аргумент называется входным формальным параметром. Подобно переменным, определенным внутри функции, формальные параметры представляют собой локальные переменные, действующие в рамках только данной функции.

Когда функция принимает аргументы, ее прототип задает их количество и типы, при этом употребляются списки по типам, в которых имена переменных отделяются друг от друга запятыми. Например, double func(double a, double b, int c). При желании имена переменных можно опустить в прототипе. Например в нашем случае, можно записать: int sign(int).

Первое ключевое слово int в прототипе int sign(int a) означает, что значение переменной, которое функция возвращает в основную функцию main() целого типа. Возврат осуществляется в точку вызова функции.

Значение переменной **a** назначается с помощью фактического аргумента в вызове функции sign().

В строчке программы z=sign(x)+sign(y)+sign(x+y) три раза происходит обращение к функции sign(). При первом обращении фактическим аргументом является переменная **x**, при втором – **y**, при третьем – **x+y**. При каждом обращении значению формального параметра присваивается значение фактического параметра, т.е. при первом обращении переменной **a** присваивается значение **x**, при втором переменной **a** присваивается значе-

ние **y,** при третьем обращении переменной **a** присваивается значений **x+y.**

Таким образом передается информация из вызывающей функции в вызываемую. В данной программе из основной функции main() информация передается в функцию sign(). Чтобы передать информацию в противоположном направлении используется возвращаемое значение, которое передается из вызываемой функции в вызывающую с помощью оператора return. Ключевое слово return обеспечивает то, что следующее за ним значение переменной становится возвращаемым значением функции. Возвратить с помощью return можно только одно значение переменной. В рассматриваемом случае функция sign() возвращает значение, присвоенное переменной **ss**. Поскольку **ss** имеет тип int, следовательно, функция sign() также имеет этот тип.

1.3.2. Функции, возвращающие одно нецелое значение

В предыдущих примерах функции либо вообще не возвращали результирующих значений (void), либо возвращали значения типа int. А как быть, когда результат функции должен иметь другой тип? Многие вычислительные функции, как, например, sqrt, sin и cos, возвращают значения типа double; другие специальные функции могут выдавать значения еще каких-то типов. Чтобы проиллюстрировать, каким образом функция может возвратить нецелое значение, напишем функцию atof(s), которая переводит строку s в соответствующее число с плавающей точкой двойной точности. Она имеет дело со знаком (которого может и не быть), с десятичной точкой, а также с целой и дробной частями, одна из которых может отсутствовать. Наша версия не является высококачественной программой преобразования вводимых чисел; такая программа потребовала бы заметно больше памяти. Функция atof входит в стандартную библиотеку программ: ее описание содержится в заголовочном файле <stdlib.h>.

Прежде всего отметим, что объявлять тип возвращаемого значения должна сама atof, так как этот тип не есть int. Указатель типа задается перед именем функции.

```
#include <ctype.h>
/*atof: преобразование строки s в double */
double atof (char s[])
{
   double val, power;
   int i, sign;

  for (i = 0; isspace(s[i]); i++) ; /* игнорирование левых символов-разделителей */
   sign = (s[i] == '-') ? –1 : 1;
   if (s[i] =='+' || s[i] =='-')
      i++;
   for (val = 0.0; isdigit (s[i]); i++)
      val = 10.0 * val + (s[i] - '0');
   if (s[i] == '.')
      i++;
   for (power = 1.0; isdigit(s[i]; i++)
{
      val = 10.0 * val + (s.[i] - '0');
      power *= 10.0;
   }
   return sign * val / power;
}
```

Кроме того, важно, чтобы вызывающая программа знала, что atof возвращает нецелое значение. Один из способов обеспечить это - явно описать atof в вызывающей программе. Подобное описание демонстрируется ниже в программе простенького калькулятора (достаточного для проверки баланса чековой книжки), который каждую вводимую строку воспринимает как число, прибавляет его к текущей сумме и печатает ее новое значение.

```
#include <stdio.h>
#define MAXLINE 100
/* примитивный калькулятор */
main()
```

```
{
   double sum, atof (char[]);
   char line[MAXLINE];
   int getline (char line[], int max);

   sum = 0;
   while (getline(line, MAXLINE) > 0)
      printf ("\t%g\n", sum += atof(line));
   return 0;
}
```

В объявлении

```
double sum, atof (char[]);
```

говорится, что sum - переменная типа double, а atof - функция, которая принимает один аргумент типа char[] и возвращает результат типа double.

Объявление и определение функции atof должны соответствовать друг другу. Если в одном исходном файле сама функция atof и обращение к ней в main имеют разные типы, то это несоответствие будет зафиксировано компилятором как ошибка. Но если функция atof была скомпилирована отдельно (что более вероятно), то несоответствие типов не будет обнаружено, и atof возвратит значение типа double, которое функция main воспримет как int, что приведет к бессмысленному результату.

Это последнее утверждение, вероятно, вызовет у вас удивление, поскольку ранее говорилось о необходимости соответствия объявлений и определений. Причина несоответствия, возможно, будет следствием того, что вообще отсутствует прототип функции, и функция неявно объявляется при первом своем появлении в выражении, как, например, в

```
sum += atof(line);
```

Если в выражении встретилось имя, нигде ранее не объявленное, за которым следует открывающая скобка, то такое имя по контексту считается именем функции, возвращающей ре-

зультат типа int; при этом относительно ее аргументов ничего не предполагается. Если в объявлении функции аргументы не указаны, как в

```
double atof();
```

то и в этом случае считается, что ничего об аргументах atof не известно, и все проверки на соответствие ее параметров будут выключены. Предполагается, что такая специальная интерпретация пустого списка позволит новым компиляторам транслировать старые Си-программы. Но в новых программах пользоваться этим - не очень хорошая идея. Если у функции есть аргументы, опишите их, если их нет, используйте слово void.

Располагая соответствующим образом описанной функцией atof, мы можем написать функцию atoi, преобразующую строку символов в целое значение, следующим образом:

```
/* atoi: преобразование строки s в int с помощью atof */
int atoi (char s[])
{
   double atof (char s[]);
   return (int) atof (s);
}
```

Обратите внимание на вид объявления и инструкции return. Значение выражения в

```
return выражение;
```

перед тем, как оно будет возвращено в качестве результата, приводится к типу функции. Следовательно, поскольку функция atoi возвращает значение int, результат вычисления atof типа double в инструкции return автоматически преобразуется в тип int. При преобразовании возможна потеря информации, и некоторые компиляторы предупреждают об этом. Оператор приведения явно указывает на необходимость преобразования типа и подавляет любое предупреждающее сообщение.

Упражнения по программированию

1. Для 10 пар чисел, вводимых в цикле по парам, найти квадрат разности этих чисел. Использовать функцию для нахождения квадрата разности. Печать получившихся значений осуществлять в основной программе.

2. Для целых чисел s и t вычислить:

z(s,t)+(min(z^2(s*t,t),z(s+t,10)))+z(s+s,t*t), где z(a,b)=((a+b)*(a-b))*a.

1.3.3. Функции, возвращающие несколько значений

/* Программа, использующая функцию, возвращающую два значения*/

```
#include <stdio.h>
#include <conio.h>
void fff(int, int, int*summa,int*raznost);
int main(void)
{
  int x,y,sum,razn,i;
   clrscr();
   for (i=0;i<4;i++)
   {
   printf("Введите x=");
   scanf("%d",&x);
     printf("Введите y=");
   scanf("%d",&y);
   fff(x,y,&sum,&razn);
   printf("sum=%d ",sum);
   printf("razn=%d ",razn);
   printf("\n");
   };
   getch();
```

```
   return 0;}
 void fff(int a,int b, int *summa,int *raznost)
 {
*summa=a+b;
*raznost=a-b;
 }
```

В данной программе разработана функция fff() для нахождения суммы и разности двух чисел. С помощью этой функции в основной программе вычисляется сумма и разность четырех пар чисел, которые вводятся с клавиатуры последовательно.

Как мы уже знаем, с помощью оператора return в основную функцию main() можно передать только один параметр. Для передачи в основную программу нескольких параметров используются указатели, с помощью которых можно имитировать работу подпрограммы-процедуры существующей в других языках программирования, но отсутствующей в языке программирования Си.

Указатели

Указатель представляет собой адрес, по которому хранится соответствующая переменная [1,2].

Унарная операция & выдает адрес, по которому хранится соответствующая переменная.

Если pp является именем переменной, то &pp является адресом, по которому хранится переменная. Адрес можно представить как некоторую ячейку в памяти.

Предположим, что имеется оператор: pp=24.

Предположим, что адрес, по которому хранится переменная pp, выглядит следующим образом: OB76(адреса задаются в шестнадцатеричной форме).

Тогда оператор printf(“%d %p\n”,pp,&pp) выводит следующие значения:

24 OB76.

Для описания переменной, в которой хранится указатель, используется специальный тип.

Переменная типа char в качестве значения имеет символ. Переменная int в качестве значения имеет число. Переменная типа указатель принимает значение адреса переменной.

Тип указатель задается следующим образом:

```
int *ptr; /*  *ptr указатель на целочисленную переменную, т.е. адрес целочисленной переменной*/
char *pc; /*  *pc указатель на символьную переменную*/
float *pf,*pg; /*  *pf  и *pg указатели на вещественные переменные*/
```

Рассмотрим следующий пример:

```
nurse=22;
ptr=&nurse;
val=*ptr;
```

В данном примере переменной nurse присваивается значение 22. Затем переменной ptr, которая описана как указатель, присваивается значение адреса nurse. Третья производимая операция называется операцией разыменования, или операцией снятия косвенности, т.е. переменной val присваивается значение, хранящееся по адресу ptr. В конечном итоге переменная val получает значение 22.

Теперь посмотрим как работает программа, использующая функцию, возвращающую два значения. Вызов функции имеет вид: fff(x,y,&sum,&razn).

Переменные x и y являются входными фактическими параметрами. Они передаются из основной функции main() в функцию fff() и замещают своими значениями формальные параметры a и b. Для передачи параметров sum и razn, которые являются выходными, используются не их значения, а их адреса. Это означает, что формальные аргументы summa и raznost, появляющиеся в прототипе и определении функции fff(), используют адреса в качестве своих значений. В силу этого они должны быть объявлены как указатели. Поскольку переменные sum и razn целого типа, summa и raznost указатели на целые зна-

чения, то описание функции должно выглядеть следующим образом:

void fff(int a,int b, int *summa,int *raznost).
Тело функции fff() содержит два следующих оператора:
*summa=a+b;
*raznost=a-b;

Эти операторы являются операторами снятия косвенности, т.е. по адресу переменной summa записывается значение a+b, а по адресу переменной raznost записывается значение a-b.

Таким образом происходит имитация передачи данных в основную программу. На самом деле никакой передачи не происходит. В основной программе при обращении к функции указываются адреса переменных, куда необходимо записать результаты работы этой функции, а в ее теле по этим адресам записываются получившиеся значения переменных.

В следующем примере в качестве аргумента используется массив чисел. В основной программе вводится количество элементов массива mas. Затем с помощью функции vvod() осуществляется заполнение массива элементами. Вывод элементов массива происходит в основной программе. Для передачи массива используется указатель, в котором хранится адрес первого элемента массива.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
void vvod(int*,int);
int main(void)
{
   int n,i;
   int mas[20];
   clrscr(); randomize();
   printf("Введите количество элементов в массиве n=");
   scanf("%d",&n);
   vvod(&mas[0],n);
    printf("\n");
```

```
   for (i=0;i<n;i++)
   printf("%d  ",mas[i]);
   getch();
   return 0;
};
 void vvod(int *mass,int nn)
 {
 int ii;
  printf("Количество элементов в массиве равно \n",nn);
   for (ii=0;ii<nn;ii++)
   {
  mass[ii]=random(12);
   }
 }
```

В теле функции ввода осуществляется заполнение массива с помощью генератора случайных чисел. Если необходимо заполнение осуществлять с помощью ввода с клавиатуры, то используется следующий оператор: scanf("%d",&mass[ii]).

Упражнения по программированию

1. Для 10 пар чисел, вводимых в цикле по парам, найти квадрат разности и квадрат суммы. Использовать функцию для нахождения квадрата разности и квадрата суммы. Печать получившихся значений осуществлять в основной программе.

2. В 9 прямоугольниках, задаваемых его сторонами а и в, найти сумму всех сторон и разность между суммой больших и суммой меньших сторон. Использовать функцию для нахождения заданных условием сумм.

3. В массивах D(12), C(14) вычислить произведение и среднее значение всех элементов. Для формирования массивов и для нахождения произведения и среднего использовать функ-

ции. Вывод массивов, вывод произведения и среднего осуществлять в основной программе.

4. Составить функцию, которая вычисляет сумму и произведение ненулевых элементов массива X(n). С помощью функции подсчитать сумму и произведение ненулевых элементов массивов A(10), B(12), C(8).

5. Составить функцию, которая находит максимальный и минимальный элементы в массиве Y(m). С помощью функции найти максимальный и минимальный элементы в массивах D(12), B(16).

6. Составить функцию, которая вычисляет сумму и количество элементов массива X(n), принадлежащих интервалу A?B. С помощью функции вычислить сумму и количество элементов массивов A(10), D(14), B(17), принадлежащих интервалу Y?Z. Значения Y и Z ввести с клавиатуры.

7. Отсортировать массивы C(14) и D(20) по возрастанию. Для формирования массивов и для сортировки использовать функции. Вывести массивы в основной программе до и после сортировки.

8. В массивах Z(14), Y(10),X(12) найти количество одинаковых элементов и сумму максимального и минимального элементов. Для формирования массивов и для нахождения количества одинаковых элементов и суммы максимального и минимального элементов использовать функции.

2. ФУНКЦИИ И ПРОЦЕДУРЫ В ЯЗЫКЕ ПРОГРАММИРОВАНИЯ ПАСКАЛЬ

2.1. Определение подпрограмм–процедур и подпрограмм–функций

Подпрограмма – именованная, логически законченная группа операторов языка, которую можно вызвать для выполнения по имени любое количество раз из различных мест программы.

В языке Паскаль существуют два вида подпрограмм: процедура и функция.

Главное отличие процедур от функций: результатом функции является одно единственное число, результатом процедуры может быть как одно число, так и множество чисел.

Таким образом, при передаче в основную программу нескольких чисел, нам не нужно имитировать работу функции, используя указатели, как мы это делали в языке программирования Си. В Паскале в этом случае мы будем использовать подпрограмму процедуру.

Все процедуры и функции языка Паскаль можно разделить на два класса:

- стандартные процедуры и функции языка Паскаль;
- процедуры и функции определенные пользователем.

Впрочем, то же самое можно сказать и о функциях языка Си.

К стандартным функциям относятся функции: abs(x), sin(x), cos(x), exp(x), round(x) и многие другие. К стандартным процедурам: dec(x,n) – уменьшает значение целочисленной переменной x на n. Если n отсутствует, то уменьшает значение x на 1. Inc(x,n) - увеличивает значение целочисленной переменной x на n. Если n отсутствует, то увеличивает значение x на 1. С полным перечнем стандартных процедур и функций можно ознакомиться в справочниках по программированию на языке Паскаль.

Пользователь может создавать свои процедуры и функции. Для этого он должен описать процедуру или функцию в разделе

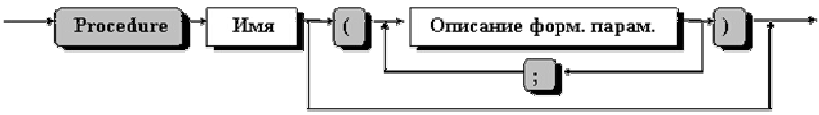
описания процедур и функций. Затем там, где это необходимо, вызвать процедуру (функцию). Прототипирование процедур и функций в языке Паскаль не требуется.

2.2. Описание подпрограммы–процедуры

Описание процедуры включает заголовок и тело процедуры.

Заголовок состоит из зарезервированного слова procedure, имени процедуры и, заключенного в скобки, списка формальных параметров с указанием типа.

Заголовок имеет вид:



Примеры заголовка процедуры:

```
procedure Picture;
procedure Power(X: real; n: Integer; var u, v: Real);
procedure Integral( a, b, epsilon: Real; var S: Real);
```

Название «формальные» параметры получили в связи с тем, что в этом списке заданы только имена для обозначения исходных данных и результатов работы процедуры, а при вызове подпрограммы на их место будут поставлены конкретные значения, которые называются фактическими параметрами. Механизм формальных – фактических параметров обеспечивает механизм замены, который позволяет выполнять процедуру с различными данными. Между фактическими параметрами в операторе вызова процедуры и формальными параметрами в заголовке процедуры устанавливается взаимно однозначное соответствие. Количество, типы и порядок следования формальных и фактических параметров должны совпадать.

Тело процедуры – блок, по структуре аналогичный программе.

При создании программ, использующих процедуры, следует учитывать, что все объекты, которые описываются после заголовка в теле процедуры, называются локальными объектами и доступны только в пределах этой процедуры.

Все объекты, описанные в вызывающей программе, называются глобальными и являются доступными внутри процедур, вызываемых этой программой.

Формальные параметры нельзя описывать в разделе описания процедуры.

Можно выделить два класса формальных параметров:

1. параметры-значения;
2. параметры-переменные.

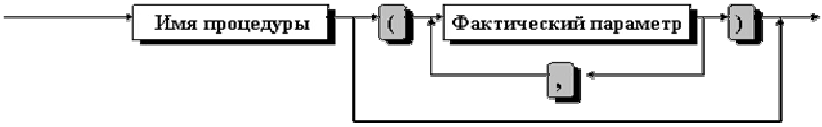
Группа параметров, перед которыми отсутствует служебное слово Var, называется параметрами-значениями. Группа параметров, перед которыми есть служебное слово Var, называется параметрами-переменными. Параметры-значения используются в качестве входных параметров подпрограммы. При их использовании фактические параметры никак не изменяются. При обращении к подпрограмме, значения фактических параметров передаются в подпрограмму и там обрабатываются уже формальные параметры. Параметры-переменные могут использоваться как в качестве входных параметров, так и в качестве выходных параметров. При их использовании в результате выполнения подпрограммы изменяются фактические параметры. При обращении к подпрограмме фактические параметры замещают формальные и сами непосредственно участвуют в выполнении процедуры. В результате выполнения подпрограммы изменяются значения фактических параметров.

В качестве входных параметров нужно использовать параметры значения, в качестве выходных – параметры-переменные.

Оператор процедуры имеет вид:

< имя > или < имя > (< список фактических параметров >)

Синтаксическая диаграмма оператора процедуры следующая:



Примеры операторов процедуры:

```
Picture
Power(( a + b )/2, 3, degree, root )
Integral ( 0, P/2, 1E-6, SUMMA)
```

Обратите внимание на соответствие между заголовком процедуры и оператором процедуры. Между списками формальных и фактических параметров установлено взаимнооднозначное соответствие, определенное их местами в списках. Это соответствие иллюстрируется следующим примером:

Рассмотрим заголовок процедуры и оператор этой процедуры:

```
Procedure Integral ( a, b, eps: real; var s: real );
Integral ( -Pi/2, Pi/2, 1E-6, summa );
```

Соответствие:

Формальный параметр	***Фактический параметр***
Значение a	Выражение - Pi/2
Значение b	Выражение Pi/2
Значение eps	Данное 1E-6
Переменная s	Переменная Summa

Тело процедуры оформляется следующим образом:

```
Procedure <имя> (список формальных параметров, блок описания);
Const …;   ⎫
   …       ⎬ блок описания
Var ….;    ⎭
begin
<операторы>
end;
```

Следует отметить, что в отличие от Си, подпрограммы в языке программирования Паскаль, должны располагаться до основной программы.

Рассмотрим примеры программ с использованием процедур.

Пример 1.

```
Program Prog;
Uses Crt;
Type vector=array[1..9] of real;
Var  Y,Z:vector;
     j: integer;
     M,D,S:real;
{ Процедура заполнения массива }
   Procedure Zap_mas(Var X:vector; n:integer);
   begin
   for j:=1 to 9 do
   X[j]:=random;
    end;
{ Процедура вывода массива }
   Procedure Vivod(X:vector; n:integer);
    begin
   for j:=1 to 9 do
    write( X[j]:4:2);
    writeln;
    end;
```

{ Процедура вычисления математического ожидания, дисперсии и среднеквадратического отклонения }

```
Procedure Mat_Disp(X:vector;Var Mat,Disp,Sro:real);
begin
{Мат. ожидание}
Mat:=0.0;
for j:=1 to 9 do Mat:=Mat+X[j];
Mat:=Mat/9;
{Дисперсия}
Disp:=0.0;
for j:=1 to 9 do Disp:=Disp+SQR(X[j]-Mat);
Disp:=Disp/9;
{Среднеквадратичное отклонение}
Sro:=SQRT(Disp)
end;
      {Основная программа}
Begin
ClrScr;
Zap_mas (Y,9);
Zap_mas (Z,9);
Mat_Disp(Y,M,D,S);
Vivod(Y,9);
writeln(' M= ',M:6:4, ' D=',D:6:4,' S=',S:6:4);
Mat_Disp(Z,M,D,S);
Vivod(Z,9);
writeln(' M= ',M:6:4, ' D=',D:6:4,' S=',S:6:4);
writeln('Для выхода из программы нажмите любую клавишу');
readkey;
end.
```

Данная программа с помощью процедуры Zap_mas() формирует массивы Y и Z**,** состоящие из 9 случайных чисел. Формирование происходит с помощью генератора случайных чисел.

Заголовок процедуры имеет вид:

```
Procedure Zap_mas(Var X:vector; n:integer).
```

Формальными параметрами для данной процедуры являются массив X и его размерность n. Формальный массив X имеет такой же тип, как и фактические массивы Y и Z, которые передаются в данную процедуру для заполнения. В заголовке процедуры перед формальным параметром X, стоит служебное слово Var. В данном случае это означает, что массив X является и входным и выходным параметром одновременно. Формальный параметр n является входным параметром. Его значение в теле процедуры не меняется.

Для заполнения двух массивов программа два раза обращается к процедуре Zap_mas(), посылая туда соответствующие фактические параметры:

```
Zap_mas (Y,9);
Zap_mas (Z,9);
```

Далее программа использует процедуру Mat_Disp(), в которой вычисляется математическое ожидание (Mx), дисперсия (Dx), и среднеквадратическое отклонение σx случайной величины X:

$$M_x = \frac{\sum_{i=1}^{N} X_i}{N} \qquad D_x = \frac{\sum_{i=1}^{N} (X_i - M_x)^2}{N} \qquad \sigma_x = \sqrt{D_x}\ .$$

Заголовок процедуры имеет вид:

```
Procedure Mat_Disp(X:vector; Var Mat,Disp,Sro:real).
```

Массив X является формальным входным параметром. Параметры Mat, Disp, Sro являются формальными выходными параметрами, поэтому перед ними стоит служебное слово Var.

После формирования массивов программа обращается к процедуре вычисления математического ожидания, дисперсии и среднеквадратического отклонения, посылая туда фактический входной параметр Y и принимая назад в основную программу

фактические выходные параметры M,D,S. Обращение к процедуре имеет вид: Mat_Disp(Y,M,D,S). Затем программа обращается к процедуре Vivod(), в которой осуществляется вывод, сформированного массива Y: Vivod(Y,9). Параметры процедуры являются входными, так как вывод элементов массива осуществляется в теле процедуры. Назад в основную программу информация не передается. Значения M, D, S для массива Y печатаются в основной программе.

Затем те же действия осуществляются для массива Z:

```
Mat_Disp(Z,M,D,S);
Vivod(Z,9);
writeln(' M= ',M:6:4, ' D=',D:6:4,' S=',S:6:4);
```

Пример 2. Попробуйте разобраться в тексте программы самостоятельно.

Программа вычисляет координаты точки (x0, y0) при последовательных поворотах и параллельных переносах системы координат.

```
Program Coordinates;
  Const Pi = 3.141592;
    Var Alfa, Beta : Real;
        x0, y0, x1, y1, x2, y2 : Real;
        x, y : Real;
  Procedure Rotate(x, y, Fi: Real; var u, v: Real );
       var cosFi, sinFi : Real; { локальные переменные }
      begin
        Fi := Fi*Pi/180 ;
        cosFi := Cos(Fi);
        sinFi := Sin(Fi);
        { параметры x, y защищены от глобальных переменных x, y }
         u := x * cosFi - y * sinFi ;
         v := x * sinFi + y * cosFi
      end ;
  Procedure Move(x, y, a, b : Real; var u, v: Real);
     begin
```

```
    u := x + a ; v := y + b
   end;
begin
  Read (x0, y0);
  Read (Alfa);
  Rotate(x0, f0, alfa, x, y);
  Read (x1, y1);
  Move(x, y, x1, y1, x, y);
  Read (Beta);
  Rotate(x, y, Beta, x, y);
  Read ( x2, y2 );
  Move(x, y, x2, y2, x, y);
  Writeln ('==============================');
  Writeln ('абсцисса точки : ', x);
  Writeln ('ордината точки : ', y);
end.
```

Упражнения по программированию

1. В массивах A(10), B(12), C(15) заменить все элементы следующие за элементом с максимальным значением на значение минимального элемента. Для формирования массивов, для замены и для вывода массивов использовать подпрограммы. Массивы необходимо вывести до и после замены.

2. В массивах A(100), B(120), C(150) подсчитать количество ненулевых элементов, лежащих до максимального элемента и количество нулевых элементов, лежащих после минимального элемента.

3. В массивах A(10,15), B(12,14), C(15,10) найти значение максимума в первой строке и значение минимума в последней строке элементов. Для ввода, вывода элементов массивов, а также для нахождения максимума и минимума разработать соответствующие подпрограммы. Максимумы и минимумы печатать в основной программе.

4. В матрицах A(10,10), B(12,12), C(15,15) найти сумму элементов, лежащих выше главной диагонали, и произведение элементов, лежащих на главной диагонали.

5. Составить подпрограмму, которая преобразует матрицу X(n,m) таким образом, чтобы нечетные строки матрицы были упорядочены по возрастанию, а четные по убыванию. С помощью подпрограммы преобразовать матрицы A(6,7), B(6,8), C(5,7), D(5,8). Составить процедуру для ввода матриц, значения элементов которых должны лежать в диапазоне от 0 до 10. Составить процедуру для вывода матриц.

5. Опишите процедуры сложения и перемножения n*n матриц, состоящих из действительных чисел. С помощью этих процедур составьте программу вычисления матрицы B = AX + XA + E по данным матрицам A и X.

6. Опишите процедуру поиска наибольшего и наименьшего элементов в массиве. С помощью этой процедуры составьте программу, определяющую, совпадают ли наибольший и наименьший элементы двух массивов A[1..n] и B[1..n].

7. Опишите процедуры поиска среднего по величине элемента в массиве A[1..2n+1] и среднего арифметического элементов этого массива. С помощью этих процедур составьте программу, сравнивающую по величине среднее по величине и среднее арифметическое в массиве A[1..99].

8. Опишите процедуру поиска наибольшего и наименьшего элементов в массиве A[1..n]. С помощью этой процедуры составьте программу, определяющую координаты вершин наименьшего прямоугольника со сторонами, параллельными осям координат и содержащего в себе множество точек T1(X1,Y1), ... , Tn(Xn,Yn).

9. Опишите процедуру проверки разложения натурального числа в сумму двух квадратов. Составьте программу, кото-

рая выбирает из данного массива те и только те числа, которые разложимы в сумму двух квадратов.

10. Опишите процедуру перевода целого числа из десятичной системы в р-ичную систему счисления. Опишите процедуру перевода правильной десятичной дроби в р-ичную с заданным количеством разрядов. Составьте программу перевода произвольного действительного числа из 10-чной в р-ичную систему счисления.

11. Опишите процедуру поиска наименьшего элемента в строке матрицы. Опишите функцию проверки на максимальность данного элемента в столбце матрицы. Составьте программу поиска седловой точки матрицы.

2.3. Описание подпрограммы–функции

Подпрограмма–функция обрабатывает данные, переданные ей из главной программы, и затем возвращает полученный результат. Функция, определенная пользователем, состоит из заголовка и тела функции.

Описание функции имеет, по существу, такой же вид, как и описание процедуры. Отличие заключается в заголовке, который имеет вид:

Function < имя > : < тип результата > ;

или Function < имя > (<список описаний формальных параметров>): < тип результата >;

Синтаксическая диаграмма заголовка функции следующая:

Заголовок содержит зарезервированное слово Function, имя, список формальных параметров (заключенный в скобки) и тип возвращаемого функцией значения.

Тело функции представляет собой локальный блок, по структуре сходный с программой.

Общий вид описания функции:

```
Function <имя> (<параметры>): <тип результата>;
Const …;   ⎫
   …       ⎬ блок описания
Var ….;    ⎭
begin
<операторы>
end;
```

В разделе операторов должен находиться, хотя бы один оператор, присваивающий имени функции значение. Обращение к функции осуществляется по имени с указанием списка аргументов. Каждый аргумент должен соответствовать формальным параметрам и иметь тот же тип.

Рассмотрим примеры, использующие подпрограмму–функцию.

Пример 1.

```
Program func:
Uses Crt;
Var rez:real; m,n,mn1:integer;
      Function Fact(l:integer):integer;
      Var i,p:integer;
        begin
```

```
        i:=1;
        p:=1;
        while i<=l do
          begin
          p:=p*i;
          i:=i+1
          end;
        Fact:=p;
        writeln ('Fact=', p)
        end;
Begin {Основная программа}
ClrScr;
Write('Введите значение n=');
Readln(n);
  Write('Введите значение m=');
  Readln (m);
  mn1:=n+m;
  rez:=(Fact(n)+Fact(m))/Fact(mn1);
writeln ('Результат равен rez=',rez:5:2);
writeln ('Для выхода из программы нажмите Enter');
readln
end.
```

В данной программе вычисляется значение функции

$$r = \frac{m!+n!}{(m+n)!}.$$

Для вычисления m!, n! и (m+n)! используется функция Fact().

Заголовок функции имеет следующий вид:

```
Function Fact(l:integer):integer.
```

Переменная l целого типа является формальным входным параметром. Имя функции Fact является выходным параметром. Выходной параметр также целого типа. В теле функции имени функции присваивается результат вычисления факториала.

В основной программе три раза происходит обращение к подпрограмме Fact():

```
rez:=(Fact(n)+Fact(m))/Fact(mn1).
```

В первый раз в качестве фактического входного параметра используется значение переменной n, второй раз – значение переменной m, в третий раз – значение переменной mn1= m+n.

В следующих примерах попробуйте разобраться самостоятельно.

Пример 2.

Функция GCD (алгоритм Евклида) вычисляет наибольший общий делитель двух натуральных чисел x и y.

```
Function GCD (x, y : Integer) : Integer ;
   Begin
     While x <> y do
       If x < y
        then y := y - x
        else x := x - y ;
        GCD := x
      End;
```

Пример 3.

Функция IntPow возводит действительное число x в степень N. (Y = x N)

```
Function IntPow(x: Real; N: Integer) : Real;
     Var i: Integer;
   Begin
     IntPow := 1;
     For i:=1 to Abs(N) do IntPow := IntPow * x;
     If N < 0 then IntPow := 1/IntPow
    End;
```

Пример 4.

Программа вычисляет наибольший общий делитель последовательности натуральных чисел, представленненных в виде массива.

```
Program GCD_of_Array;
   Const n = 100 ;
     Var i, D : Integer;
         A : Array[1..n] of Integer;
Function GCD (x, y : Integer) : Integer ;
  Begin
   While x <> y do
    If x < y
     then y := y - x
     else x := x - y;
     GCD := x
   End;
Begin { основная программа }
{Блок чтения массива натуральных чисел}
  D := GCD (A[1], A[2]);
  For i := 3 to n do D := GCD(D, A[i]);
  writeln ( ‘ НОД последовательности = ‘ , D )
 End.
```

Упражнения по программированию

1. Задать массивы X(10), Y(20), Z(12). Получить:

F=

$$\begin{cases} max(X_1, X_2, ..., X_{10}) + max(Y_1, Y_2, ..., Y_{20}), & если \quad max(Z_1, Z_2, ..., Z_{12}) \geq 2 \\ 1 + (max(X_1, X_2, ..., X_{10}))^2, & если \quad max(Z_1, Z_2, ..., Z_{12}) < 2 \end{cases}$$

3. Используя подпрограмму–функцию найти суммы элементов массивов D(50), V(40), C(30), A(20). Поместить их в массив DS, который затем отсортировать по возрастанию.

4. Используя подпрограмму–функцию для нахождения факториала вычислить сумму факториалов всех четных чисел от 2 до 100.

5. Найти разность между средним арифметическим и минимальным элементом массивов C(4,5), A(5,6), D(5,6).

6. Используя подпрограмму–функцию вычислить в матрицах X(4,6), D(4,5), L(4,3) количество нечетных элементов. Вывести матрицы и количество нечетных элементов.

7. Используя подпрограмму–функцию вычислить в матрицах X(4,6), D(4,5), L(4,3) количество элементов кратных 7. Вывести матрицы и количество элементов.

8. Опишите функцию проверки простоты числа. Опишите функцию проверки числа на разложимость в сумму вида 1+2a. Составьте программу, которая выбирает из данного массива чисел те и только те его элементы, которые удовлетворяют обоим свойствам.

2.4. Вложенные процедуры и функции

Каждая процедура или функция может, в свою очередь, содержать раздел процедур и функций, в котором определены одна или несколько процедур и функций. В этом случае говорят о вложении процедур. Количество уровней вложенности может быть произвольным.

Понятия локальных и глобальных объектов распространяются и на вложенные процедуры. Например, переменная, описанная в процедуре A локальна по отношению к основной программе и глобальна для процедур B и C, вложенных в A.

В некоторых случаях необходимо из процедуры осуществить вызов другой процедуры, описанной в том же разделе процедур и функций.

Например, процедура C может содержать оператор вызова процедуры B. В этом случае компилятор правильно обработает текст программы, поскольку процедура B описана до процедуры C. Если же из процедуры B необходимо обратиться к C, для

правильной обработки вызова C необходимо использовать механизм опережающего (предварительного) описания C. Опережающее описание процедуры (функции) - это ее заголовок, вслед за которым через “;” поставлено служебное слово Forward. В тексте программы опережающее описание должно предшествовать процедуре, в которой предварительно описанная процедура вызывается.

Если процедура или функция описана предварительно, описанию ее тела предшествует сокращенный заголовок, состоящий только из соответствующего служебного слова и имени - без списка описаний параметров.

```
Procedure A (x : TypeX; Var y : TypeY); Forward;
Procedure B (z : TypeZ) ;
Begin
... A( p, q); ...
End;
Procedure A;
Begin
 ...
End;
```

2.5. Рекурсивно-определенные процедуры и функции

Описание процедуры A, в разделе операторов которой используется оператор этой процедуры, называется рекурсивным. Таким образом, рекурсивное описание имеет вид:

```
Procedure A(u, v : ParType);
 ...
Begin
...;
A(x, y);
 ...
End;
```

Аналогично, описание функции F, в разделе операторов которой используется вызов функции F, называется рекурсивным. Рекурсивное описание функции имеет вид:

```
Function F(u, v : ArgType) : FunType;
 ...
Begin
...;
z := g(F(x, y));
...
End;
```

Использование рекурсивного описания процедуры (функции) приводит к рекурсивному выполнению этой процедуры (вычислению этой функции). Задачи, естественным образом формулируемые как рекурсивные, часто приводят к рекурсивным решениям.

Рассмотрим рекурсивное определение функции n!=1*2*...*n (n-факториал).

Пусть F(n) = n! Тогда

1. F(1) = 1
2. F(n) = n*F(n - 1) при n > 0

Средствами языка это определение можно сформулировать как вычисление:

```
If  n = 1
then
F:= 1
else
F:= F(n - 1) * n
```

Оформив это вычисление как функцию, и изменив имя, получим:

```
Function Fact(n: Integer): Integer;
  Begin
```

```
If  n = 1
 then
 Fact := 1
 else
  Fact := Fact(n - 1) * n
 End;
```

Вычисление функции Fact можно представить как цепочку вызовов новых копий этой функции с передачей новых значений аргумента и возвратов значений функции в предыдущую копию.

Цепочка вызовов обрывается при передаче единицы в новую копию функции. Движение в прямом направлении (разворачивание рекурсии) сопровождается только вычислением условия и вызовом. Значение функции вычисляется при сворачивании цепочки вызовов. Сложность вычисления Tfact(n) функции Fact можно оценить, выписав рекуррентное соотношение:

$$Tfact(n) = Tfact(n-1) + Tm + Tl + Tc$$

Для того, чтобы вычислить Tfact(n), нужно осуществить одну проверку, одно умножение и один вызов Tfact(n-1). Вызов Tfact требует затрат времени Tc на "административные" вычисления: передачу параметра, запоминание адреса возврата и т.п. Положив C = Tm+Tl+Tc, получим Tfact(n) = Tfact(n-1) + C. Нетрудно теперь показать, что Tfact(n) = Cn.

Рассмотрим пример, позволяющий сгенерировать все перестановки элементов конечной последовательности, состоящей из букв.

Попытаемся свести задачу к нескольким подзадачам, более простым, чем исходная задача.

Пусть S = [s1, s2, ..., sn] – конечный набор символов.

Через Permut(S) обозначим множество всех перестановок S, а через Permut(S,i) – множество всех перестановок, в которых на последнем месте стоит элемент si. Тогда

$$Permut(S) = Permut(S,n) \cup Permut(S, n-1) \cup ... \cup Permut(S,1)$$

Элемент множества Permut(S, i) имеет вид [sj2,..., sjn,si], где j2,..., jn – всевозможные перестановки индексов, не равных I. Поэтому Permut(S, i) = (Permut(S \ si), si) и Permut(S)=(Permut(S \ s1),s1)+...+ (Permut(S \ sn), sn).

Полученное соотношение выражает множество Permut(S) через множества перестановок наборов из (n-1) символа. Дополнив это соотношение определением Permut(S) на одноэлементном множестве, получим:

1. Permut({s}) = {s}
2. Permut(S) = (Permut(S\s1), s1) + ... + (Permut(S\sn), sn)

Уточним алгоритм, опираясь на представление набора S в виде массива S[1..n] of char.

Во первых, определим параметры процедуры Permut:

k – количество элементов в наборе символов;

S – набор переставляемых символов.

Алгоритм начинает работу на исходном наборе и генерирует все его перестановки, оставляющие на месте элемент s[k]. Если множество перестановок, в которых на последнем месте стоит s[j], уже порождено, меняем местами s[j-1] и s[k], выводим на печать полученный набор и применяем алгоритм к этому набору. Параметр k управляет рекурсивными вычислениями: цепочка вызовов процедуры Permut обрывается при k = 1.

```
Procedure Permut(k : Integer; S : Sequence);
    Var j : integer;
  Begin
    if k <> 1 then Permut(k - 1, S);
    For j := k - 1 downto 1 do
    begin
    Buf := S[j]; S[j] := S[k]; S[k] := Buf;
    WriteSequence(S); Permut(k - 1, S)
    end
```

```
End;
Begin { Раздел операторов программы}
{Генерация исходного набора S}
 WriteSequence(S); {Вывод первого набора на печать}
  Permut(n, S)
End.
```

Оценим сложность алгоритма по времени в терминах C(n): Каждый вызов процедуры Permut(k) содержит k вызовов процедуры Permut(k-1) и 3(k-1) пересылки. Каждый вызов Permut(k-1) сопровождается передачей массива S как параметра-значения, что по времени эквивалентно n пересылкам. Поэтому имеют место соотношения:

$$C(k) = kC(k-1) + nk + 3(k-1),$$
$$C(1) = 0, \text{ откуда } C(n) = (n+3)n!$$

Оценим теперь размер памяти, требуемой алгоритму. Поскольку S – параметр-значение, при каждом вызове Permut резервируется n ячеек (байтов) для S, а при выходе из этой процедуры память освобождается. Рекурсивное применение Permut приводит к тому, что цепочка вложенных вызовов имеет максимальную длину (n - 1):

Permut(n) --> Permut(n-1) --> ... --> Permut(2)

Поэтому данные этой цепочки требуют (n2 – n) ячеек памяти, т.е алгоритм требует память размера O(n2). Количество перестановок – элементов множества Permut(S) равно n!. Поэтому наш алгоритм “тратит” C(n)/n! = O(n) действий для порождения каждой перестановки. Разумеется, такой алгоритм нельзя назвать эффективным. (Интуиция нам подсказывает, что эффективный алгоритм должен генерировать каждую перестановку за фиксированное, не зависящее от n количество действий.) Источник неэффективности очевиден: использование S как параметра–значения. Это позволило нам сохранять неизменным массив S при возврате из рекурсивного вызова для того, чтобы правильно переставить элементы, готовя следующий вызов.

Рассмотрим другой вариант этого алгоритма. Используем S как глобальную переменную. Тогда при вызове Permut S будет изменяться. Следовательно при выходе из рекурсии массив S нужно восстанавливать, используя обратную перестановку!

```
Program All_permutations;
  Const n = 4;
   Type Sequence = array[1..n] of char;
    Var S : Sequence;
        Buf : char;
          i : Integer;
Procedure WriteSequence;
     begin
      For i := 1 to n do Write(S[i]); Writeln
      end;
Procedure Permut(k : Integer);
      Var j : integer;
Procedure Swap(i, j : Integer);
     begin  Buf := S[j]; S[j] := S[i]; S[i] := Buf  end;
    Begin
      if k > 1 then Permut(k - 1);
      For j := k - 1 downto 1 do begin
       Swap(j, k); {Прямая перестановка}
       WriteSequence; Permut(k - 1);
       Swap(k, j) {Обратная перестановка}
      end
    End;
 Begin
   {Генерация исходного набора S}
   WriteSequence;  Permut(n)
  End.
```

Теперь оценка C(n) получается из соотношений

$C(k) = kC(k-1) + 3(k-1)$, $C(1) = 0$, т.е. $C(n) = O(n!)$

Интересно, что этот вариант не требует и памяти размера O(n2) для хранения массивов. Необходима только память размера O(n) для хранения значения параметра k.

Далее рассмотрим несколько примеров рекурсивных процедур, которые помогут лучшему усвоению техники применения рекурсии. Мы также увидим преимущества и недостатки рекурсивных описаний.

Пример 1. Возвратные последовательности.

Рассмотрим последовательность V(n), заданную рекуррентно:

1.V(0) = V0, V(1) = V1, ..., V(k) = Vk;
2.V(n+k) = F(V(n+k-1), ... V(n+1), V(n), n)

Такое определение задает последовательность V(n) фиксированием первых ее нескольких членов и функции F, вычисляющей каждый следующий член, исходя из предыдущих. Рекуррентное определение “дословно переводится” в функцию, вычисляемую рекурсивно:

```
Function V(n: Integer) : Integer;
  Begin
   Case n of
     0:  V := V0;
      . . . . .
     k:  V := Vk
    else V := F(V(n+k-1), ... V(n+1), V(n), n)
   end    End;
```

Один из примеров возвратных последовательностей – последовательность Фибоначчи. Вот ее рекурсивная версия:

```
Function RF(n: Integer) : Integer;
  Begin
   Case n of
     0,1: RF := 1
     else RF := RF(n-1) + RF(n-2)
    end
```

End;

К сожалению, эта красивая и прозрачная версия крайне неэффективна: ее сложность определяется из соотношения Trf (n) = Trf (n-1) + Trf (n-2) + 1.

Функция Trf(n) равна функции RF(n). Нетрудно показать, что Tr f (n) = RF(n) > 2 n-2 при n > 5.

Таким образом, сложность вычисления RF(n) эспоненциальна. Сложность же итеративного алгоритма вычисления Fib линейна. В общем случае ситуация точно такая же: можно построить итеративный алгоритм вычисления возвратной последовательности линейной сложности, рекурсивная же версия при k >= 1 имеет экспоненциальную сложность.

Пример 2. Линейные диафантовы уравнения.

Перечислить все неотрицательные целые решения линейного уравнения a1x1 + a2x2 + ... + anxn = b с целыми положительными коэффициентами.

Опишем алгоритм рекурсивно, осуществив сведение исходной задачи к задаче меньшего размера.

Перепишем исходное уравнение в виде:

a1 x1 + a2 x 2 + ... + a n-1 x n-1 = b - a n x n

Организуем перебор всевозможных значений xn, при которых правая часть b - a n x n > 0. xn = 0, 1, ... y, где y = b div an. Тогда первые n-1 компонент решения (x1, ... , x n-1, x n) исходного уравнения - решение уравнения a1 x1 + a2 x2 + ... + a n-1 x n-1 = b - a n x n .

Таким образом, мы свели решение исходного уравнения к решению y +1 уравнения с n-1 неизвестным, и, следовательно, можем реализовать алгоритм рекурсивно. Определим условия выхода из рекурсии:

при b = 0 существует единственное решение - (0, 0, ..., 0)

при n = 1 если b mod a1 = 0 то x1 = b div a1 иначе решений нет.

Таким образом, выходить из рекурсивных вычислений нужно в двух (крайних) случаях. Мы установили и параметры процедуры: n и b.

```
Procedure Solution(n, b : integer);
     Var i, j, y, z : Integer;
   Begin
    If   b = 0
     then begin
      For j := 1 to n do X[j] := 0; WriteSolution end
      else If (n = 1) and (b mod a[1] = 0)
        then begin X[1]:= b div a[1]; WriteSolution end
       else If n > 1
        then begin
          z := a[n]; y := b div z;
          For i := 0 to y do begin
          X[n] := i; Solution(n - 1, b - z*i)
        end
     end
  End;
  Program AllSolutions;
     Const n = 4;
      Type Vector = array[1..n] of Integer;
       Var a, X : Vector; b : Integer;  i : Integer;
     {Procedure WriteSolution распечатывает решение X[1..n]
}
     {Procedure Solution}
     Begin
      {Ввод массива коэффициентов a[1..n] и св. члена b}
      Solution(n, b)
     End;
```

Пример 3. Возведение числа в натуральную степень.
Возвести вещественное число a в натуральную степень n.

Применим метод половинного деления степени n: an = (a n/2))2. Поскольку число n не обязательно четное, формулу нужно уточнить:

a^n = (a n div 2) 2 * a n mod 2. Дополнив определение a n определением
a1 = a, и заменив домножение на a n mod 2 разбором случаев чет-нечет, получим:

```
Function CardPower(a : Real; n : Integer) : Real; var b : Real;
  Begin
   If n = 1
    then CardPower := a
     else begin
       b := Sqr(CardPower(a, n div 2));
       If n mod 2 = 0
        then CardPower := b
        else CardPower := a*b
     end
   End;
```

Упражнения по программированию

Известная в теории алгоритмов функция Аккермана A(x, y) определена на множестве натуральных чисел рекурсивно:

A(0, y) = y + 1,
A(x, 0) = A(x - 1, 1),
A(x, y) = A(x - 1, A(x, y - 1));

а) Опишите функции A(1, y), A(2, y), A(3, y), A(4, y) явным образом;

б) Реализуйте рекурсивную программу вычисления A(x, y);

в) Реализуйте программу вычисления A(x, y) без рекурсии.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Стивен Прата. Язык программирования С. Лекции и упражнения. 5-изд. Пер. с англ.− М.:Издательский дом “Вильямс”, 2006.− 960с.
2. Подбельский В.В., Фомин С.С. Программирование на языке Си: Учеб. пособие. 2-е доп. изд. – М.: Финансы и статистика, 2005.− 600с.
3. Культин Н.Б. Программирование в Turbo Pascal 7.0 и Delphi. СПб.:БХВ – Санкт-Петербург, 1999.
4. Прайс Д. Программирование на языке Паскаль: Практическое руководство/ Пер. с англ. – М.:Мир, 1987.
5. Рюттен Т., Франкен Г. Турбо Паскаль 7.0. − К.: Торгово-издательское бюро BNV, 1997.
6. Фаронов В.В. Турбо-Паскаль 7.0. Начальный курс. − М.: Нолидж, 1999.
7. Абрамов С. А., Гнездилова Г.Г., Капустина Е.Н., Селюн М.И. Задачи по программированию. − М.: Наука, 1988.

КЛЕВЦОВА АЛЛА БОРИСОВНА

Работа с подпрограммами в Си и Паскале на ПК и в микроконтроллере

Учебно-методическое пособие
по курсу

ИНФОРМАТИКА

Для студентов специальностей
210106 Промышленная электроника и
230201 Информационные системы и технологии

Ответственный за выпуск Клевцова А.Б.
Редактор Кочергина Т.Ф.
Корректор Надточий З.И.

ЛР №020565 от 23 июня 1997г. Подписано к печати
Формат 60x84 1/16. Бумага офсетная.
Офсетная печать. Уc.п.л.– 1,6. Уч.-изд.л. – 1,4.
Заказ № Тираж 80 экз.
“С”

Издательство Технологического института
Южного федерального университета
ГСП 17А, Таганрог, 28, Некрасовский, 44
Южного федерального университета

Типография Технологического института
Южного федерального университета
ГСП 17А, Таганрог, 28, Энгельса, 1