А.А. Вылиток

Некоторые примеры из семинарских занятий по Си

Задача. Дана матрица целых чисел размера n × m. В каждой строке выбирается наименьшее из значений элементов строки, затем среди выбранных значений находится наибольшее. Написать программу, определяющую это значение.

Паскаль

```
 (1) pRoGraM Maxmin (input,output);
 (2) const m=5;
 (3)       N=6;
 (4) type vector=array[0..M-1] of integer;
 (5) type matrix=array[0..N-1] of vector;
 (6) var i,k,res: integer;
 (7)     matr:Matrix;
 (8) function min(var vec:vector):integer;
 (9)  var i,l:integer;
     begin
(10)   l:=vec[0];
(11)   for i:=1 to M-1 do
(12)     if vec[i]<l then l:=vec[i];
(13)   min:=l
(14) end;

(15) begin
(16)   for i:=0 to N-1 do
(17)     for k:= 0 to M-1 do
(18)       read(matr[i][k]);
(19)   res:=min(matr[0]);
(20)   for i:=1 to N-1 do
(21)     begin  k:=min(matr[i]);
(22)        if k > res then res:=k
(23)     end;
(24)   writeln('Maxmin =', res)
(25) end.
```

язык Си

```
 (1)  #include <stdio.h>
 (2)  #define  M  5
 (3)  #define  N  6
 (4)  typedef int vector[M];
 (5)  typedef vector matrix[N];
 (6)  int i,k,res;
 (7)  matrix matr;
 (8)  int min(vector vec)
 (9)  {
        int i, l;
(10)    l = vec[0];
(11)    for (i=1; i<=M-1; ++i)
(12)      if (vec[i]<l) l=vec[i];
(13)    return l;
(14)  }

(15)  main(){
(16)    for (i=0; i<= N-1; ++i)
(17)      for (k=0;k<=M-1; ++k)
(18)        scanf("%d", &matr[i][k]);
(19)    res = min(matr[0]);
(20)    for (i=1; i < N; ++i)
(21)      { k = min(matr[i]);
(22)        if (k >res) res = k;
(23)      }
(24)    printf("Maxmin=%d\n", res);
(25)  }
```

Замечания и комментарии

1. Лексические различия:
   - В Паскале строчные и прописные буквы при написании имен и служебных слов не различаются: имя M в 11-й строке и m во 2-й строке обозначают одну и ту же константу. В языке Си регистр букв важен: m и M считаются различными именами; *for* является ключевым словом, означающим начало оператора цикла, а *FOR* и *For* таковыми не являются.
   - Присваивание в языке Си обозначается одиночным знаком равенства = (в Паскале :=) ; для обозначения сравнения в Си используется двухсимвольный знак == Ср. строки 11 в обеих программах.
   - Операторные скобки (ограничивающие составной оператор) в Паскале обозначаются словами begin и end, в Си им соответствуют символы { и }.
   - Строковые константы ограничиваются двойными кавычками, а не апострофами, как в Паскале. (Символьные константы в Си берутся в апострофы (одинарные кавычки), как в Паскале)
2. При описании переменных в Си, в отличие от Паскаля, сначала указывается имя типа, а затем перечисляются имена переменных (строка 6).
3. Элементы массивов в языке Си нумеруются целыми числами, начиная с нуля. Поэтому при описании массива указывается не диапазон изменения индекса (как в Паскале), а размер массива (строки 4,5).
4. Программа на Паскале может содержать вложенные процедуры и функции. Программа на Си состоит из последовательности переменных и функций. Вложенность функций не допускается. Функции могут не возвращать значений (такие функции являются аналогами паскалевских процедур). Выполнение программы на языке Си начинается с первого оператора функции с именем *main*.
5. В Си с помощью директивы *define* (строки 2,3) константам можно давать имена.

6. В языке Паскаль процедуры ввода/вывода (*read, write*) являются предопределенными, т.е. не требуют явного описания в программе. В языке Си описания стандартных функций ввода/вывода (*scanf, printf*) добавляются в программу с помощью директивы *include* (строка 1).
7. Присваивание в языке Си является не оператором, а операцией (результат которой - присвоенное значение). Эта операция может быть использована в сложных выражениях. Так, например, строки 21,22 в данном примере программы на Си можно заменить строкой:

   if ((k = min(matr[i])) > res) res = k;
8. Точка с запятой «;» в языке Паскаль является «разделителем». Например, она разделяет два соседних оператора, или два соседних описания. В языке Си смысл точки с запятой иной : она является «завершителем» операторов (кроме составного оператора) и описаний, т.е. является признаком конца оператора или описания. Сравните строки 13 и 14 программ на Паскале и Си.

Примеры перечислимых типов

```
enum season {winter=1,spring,summer,autumn}; /* season – это тег
                                                перечисления   */
enum season x, y;  /* описание переменных перечислимого типа  */
x=spring; y=x;
```

Пример описания перечислимого типа и оператора switch:

```
typedef  enum {red, blue, green=5,yellow} colortype;
/* константы перечисления имеют значения соответственно 0, 1, 5, 6 */

colortype next_color (colortype color)
{
  switch(color) {
     case  red    : return blue; /* break не нужен, т.к. сразу возврат из
                                    функции */
     case  blue   : return green;
     case  green  : return yellow;
     case  yellow : return red;
  }
}
```

Допускается давать одинаковые значения константам одного перечисления, например

```
enum  unit { bike, car, helicopter=5, truck=1 };
```

В этом перечислении константы `car` и `truck` имеют одно и то же значение 1.

Структуры

Аналогом паскалевских записей в языке Си являются структуры:

```
struct  point { int x;     /* point – это тег структуры  */
                int y;     /* x и y – это поля структуры */
              }
struct  point p;  /*описание переменной типа  struct  point */

p.x=1;     /*  присваивание значений полям структуры  p   */
p.y=2;
```

Можно задать тип, который будет означать структуру:

```
typedef  struct point  Point      /* Point - имя типа структуры   */
```

Можно задать тип структуры и таким образом

```
typedef struct  { int x;     /* тег структуры пропущен */
                  int y;     /* x и y – это поля структуры */
                } Point      /*  имя типа  */
Point p1; /*описание переменной типа struct { int x;int y;} */
p1.x=4;
p1.y=5;
```

Можно совместить описание структуры с тегом и задание нового имени типа:

```
typedef struct point  { int x;     /* тег структуры — point */
                        int y;     /* x и y – поля структуры */
                      } Point      /* имя типа  */
```

При таком описании `Point` является синонимом для `struct point`.

Указатели
Аналогом паскалевского описания указателя
pr: ^ integer;
является следующее описание на языке Си:
int * pr;

Указатели на структуру:

```
struct point p = {0.0,0.0}; /* инициализация структуры p    */
struct point * pp=&p;       /* pp – указатель на структуру p */
(*pp).x=1.0;
  pp->y=2.0;   /*  эквивалентно (*pp).y=2.0  */

/*   выражение *pp.x=3; ошибочно, т.к. воспринимается как
    *(pp.x)=3 в силу большего приоритета операции точка.    */
```

Примеры использования операторов

Найти среднее арифметическое `s` положительных чисел массива `mas`.

```
double s=0;
for (i=0,k=0; i < N; i++){
     if (mas[i]<=0) continue; /* пропускаем неположительные  */
     s+=mas[i]; /*  то же, что s=s+mas[i];  */
     k++;
}
if (k!=0)   s/=k;  /*  то же, что s=s/k;  */
```

На каждой итерации проверяем положительность очередного элемента массива. Если элемент не положительный, то оператор `continue` передает управление на следующую итерацию цикла, которая для цикла `for` начинается с вычисления выражения в третьей

секции заголовка оператора цикла, затем вычисление и сравнение с нулем результата выражения из второй секции. Цикл продолжается, пока не ноль (т.е. ложь).

Поиск элемента в неупорядоченном массиве

Пусть заданы массив из N целых чисел `mas` и переменная `x`. Все числа в массиве попарно различны. Присвоить переменной `i` номер элемента массива, такого что `mas[i]==x`. Если такого элемента нет присвоить `i` значение `-1`.

```
for (i=0; i < N;i++)
    if (mas[i]==x) break; /* досрочный выход из цикла */
if (i==N) i=-1; /* нет такого элемента */
```

Оператор `break` осуществляет выход из цикла (а также из переключателя switch) — как только нужное значение найдено, продолжать цикл нет необходимости.

Другое решение:

```
i=N; /* просмотр от конца к началу */
while(i--)
    if (mas[i]==x) break;
 /* после выхода из цикла i имеет нужное значение */
```

Выражение `i--` состоит из операции постфиксного уменьшения, она обладает побочным эффектом: значение переменной `i` уменьшается на 1. Результат выражения — старое значение `i` до уменьшения.

В языке Си нет логического типа и констант *true* и *false* как в Паскале. Вместо этого используется арифметический целый тип по следующему соглашению: 0 означает ложь, все остальные значения (ненулевые) означают истину. При первой проверке условия цикла значение выражения будет равно `N`, оно отлично от нуля, следовательно, будет выполнено тело цикла. В результате побочного эффекта `i` получит значение `N-1` и на первой итерации цикла переменная `x` будет сравниваться с последним элементом массива. Если их значения совпадут, цикл прекращается и в `i` находится искомый номер, в противном случае осуществится переход на проверку условия цикла. Если `i` не равно `0`, то произойдет продвижение по массиву на один элемент влево и сравнение очередного элемента с x. Если `i` равно `0`, то весь массив уже просмотрен, выражение в условии цикла даст значение ноль, что означает ложь, и цикл завершится. При этом в результате побочного эффекта `i` получит значение -1, означающее, что искомый элемент не встречается в массиве.

Упорядочение массива (сортировка)

Дан массив целых чисел размера $N \geq 1$. Упорядочить его элементы по неубыванию.

Сортировка методом пузырька

```
void bubble_sort( int mas[], int N) {
  int i,j, temp;
  for (i=N-1; i>=0; i--)
     for (j=0; j<i; j++ )
       if (mas[j]>mas[j+1]) {
```

```
            temp=mas[j+1];
            mas[j+1]=mas[j];
            mas[j]=temp;
        }
}
```

В языке Си размер массива передается не вместе с массивом, а как отдельный параметр, (в параметре, означающем массив, размер вообще можно опускать). Причины этого будут ясны после более детального знакомства с массивами и их связью с указателями. Другой вариант — задавать размер глобальной константой, как это было в примере с поиском седловой точки в матрице.

Сортировка простым выбором (минимального элемента)

```
void insert_sort(int mas[], int N) {
  register int i,j, min;
  for (i=0; i<N; i++){
     min=i;
     for (j=i+1; j<N ; j++)
         if (mas[j]<mas[min]) min=j;

     j=mas[i];
     mas[i]=mas[min]; /*  минимальный среди  mas[i]… mas[N-1]  */
     mas[min]=j;      /*  переставили в позицию i  */
  }
 }
```

Ключевое слово `register` (спецификатор класса памяти) указывает компилятору, что соответствующие переменные будут часто использоваться и их желательно разместить с учетом минимизации времени доступа (например, на регистрах).
Противоположным по смыслу является ключевое слово `volatile` – не размещать на регистре или в кэше, так как переменная может изменяться независимо от выполняемой программы другими (внешними) процессами (параллельно выполняющимися программами).

Бинарный поиск (поиск в упорядоченном массиве)

```
/* binsearch: найти x в v[0]<=v[1]<=…<=v[n-1]  */
int  low, high, mid;
low=0;
high=n-1;
while ( low <= high){
  mid = (low+high)/2;
  if (x<v[mid])  high=mid-1;
  else if ( x>v[mid]) low=mid+1;
          else return mid; /* x найден */
}
return -1; /* x не найден */
```

Примеры разумного использования оператора goto

```
for (…)
   for (…) {
     if (disaster)   /* если «бедствие», например, обнаружено неверное данное, */
         goto error / * уйти на «ошибку»  */
}
…
/* обработка ошибки */
error: … /* здесь должны быть действия  по реакции на ошибку, например, сообщение
            пользователю о том, что произошло, и подготовка к корректному
            завершению программы  */
```

В качестве еще одного примера рассмотрим такую задачу: определить, есть ли в массивах `a` и `b` совпадающие элементы. Один из возможных вариантов ее реализации имеет следующий вид:

```
for (i = 0; i < n; i++)
    for (j = 0;j < m; j++)
        if (a[i] == b[j])
            goto found ;   /* break здесь не подойдет, так как он
                              завершит только внутренний цикл for */
/* нет одинаковых элементов */
…
found:
/* обнаружено совпадение: a[i] == b[j] */
```

Программу нахождения совпадающих элементов можно написать и без `goto`, введя переменную `found`. Она имеет значение «ложь», пока совпадения не найдено, и «истина», как только обнаружены совпадающие элементы. Проверка значения `found` входит в условия продолжения каждого из двух вложенных циклов `for`:

```
found =0;
for (i=0; i < n && !found; i++)
   for (j = 0; j <m && !found j++)
      if (a[i] == b[j])
         found = 1;
if (found)
/* обнаружено совпадение: a[i-1] == b[j-1] */
else
/* нет одинаковых элементов */
…
```

Еще один разумный вариант, где можно использовать goto, это тело функции, эмулирующей поведение некоторого конечного автомата (про конечные автоматы материал будет в дальнейших лекциях, посвященных теории трансляции).

Приведение типа

В языке Си есть префиксная операция явного приведения типа (ее уровень в таблице приоритетов – 14):
`(тип) выражение`
выражение должно быть первичным (см. материал про выражения на сайте) или состоящим из операции не ниже четырнадцатого уровня. Результатом будет значение указанного типа. Способ преобразования к новому типу – см. таблицы в приложении к задачнику Руденко. Результат не является l-обозначением, то есть не может стоять в левой части присваивания.

Квалификатор const

```
const int ci=5; /* ci – неизменяемое целое, проинициализированное значением 5 */
ci=6; /* ошибка: попытка изменить неизменяемое целое*/
int * pi; /* указатель на целое */
pi = (int *) &ci; /* явное приведение типа, к указателю на изменяемое целое */
*pi=6; /* может быть ошибка во время выполнения из-за попытки записи в память,
          доступной только для чтения; в некоторых реализациях ошибки не будет */

const int * pci = & ci; /* указатель на неизменяемое целое */
*pci = 6; /* ошибка: попытка изменить неизменяемое целое*/
const int b =7;
pci = &b; /* правильно: сам указатель изменять можно */

int a,b;
int * const cpi = & a; /* cpi -- неизменяемый указатель на целое */
cpi = & b;/* ошибка: попытка изменить неизменяемы указатель */
*cpi =10; /* правильно: содержимое, на которое указывает cpi изменять можно */

const int * const cpci = & ci; /* неизменяемый указатель на неизменяемое
целое */
```

Примеры выражений

Выражение

```
(unsigned) (unsigned short)  0xFFFFFF
```

дает значение 0xFFFF при условии, что в реализации под значения типа short отводится 16 бит, под значения типа int – 32 бита.

Выражение

```
(double) (float) 3.14159265358979323 84
```

может дать значение меньшей точности, чем исходная константа. Операции явного приведения типа относятся к префиксным и группируются справа налево:

```
(double)( (float)3.14159265358979323 84 )
```

Еще примеры:

```
unsigned i= -1; /* переменная i получит значение 65535, если под тип int
                   отводится 16 бит */

(1.0 + −3)+(unsigned)1; /* результат равен –1.0 ; */

1.0 +(−3 +(unsigned)1); /* результат – большое число; если в данной реализации
```

под тип unsigned отводится 16 бит, то результат равен $1.0 + (2^{16} - 3) + 1 = 65535.0$; тип результата double */

Применение операции «запятая»

Операция «запятая» обычно применяется в тех местах программы, где по синтаксису языка Си требуется одно выражение, но по смыслу нужно выполнить несколько выражений с побочными эффектами, например – начальная инициализация в for-цикле.

```
for (x=0,y=N; x< N && y>0; x++,y--) { … }
```

Запятая в вызове функции всегда трактуется как разделитель в списке аргументов, а не операция-запятая:

```
f(a,b=2,5*b,c); /* вызов функции с четырьмя аргументами */

f(a,(b=2,5*b),c); /* это вызов функции с тремя аргументами, второй аргумент –
                     это выражение с операцией запятая */
```

В процедуре сортировки методом пузырька `bubble_sort( )` вместо трех операторов для обмена значениями элементов массива

```
temp=mas[j+1];
mas[j+1]=mas[j];
mas[j]=temp;
```

можно использовать один оператор-выражение с операцией «запятая»:

```
temp=mas[j+1],mas[j+1]=mas[j],mas[j]=temp;
```

Условные выражения

Следующий оператор-выражение, содержащий условное выражение,

```
r=a?b:c;
```

эквивалентен условному оператору

```
if (a!= 0) r=b;
else r=c;
```

Выражение с тремя последовательными условными операциями

```
a?b:c?d:e?f:g
```

в силу правой ассоциативности операции ?: интерпретируется как

```
a?b:(c?d:(e?f:g))
```

Тип результата для выражения

```
a==b ? 1.0 : 2
```

всегда вещественный, так как выполняется балансировка 2-го и 3-го операндов.

Логические операторные выражения

В языке Си логические операции И (&&) и ИЛИ (||) возвращают значения 0 («ложь») или 1 («истина») и вычисляются по «ленивой» (сокращенной) схеме. Сначала вычисляется левый операнд, и, если по его значению можно определить результат операции (0, т.е. «ложь» в случае И при нулевом значении левого операнда; 1, т.е. «истина» в случае ИЛИ при ненулевом значении левого операнда), то правый операнд не вычисляется. В противном случае вычисления продолжаются: вычисляется значение правого операнда – если он ненулевой, то значением операции будет 1 («истина»), иначе 0 («ложь»).

Оператор-выражение

```
r=a&&b;
```

эквивалентен условному оператору

```
if (a==0) r=0;
else{
    if (b==0) r=0;
    else  r=1;
}
```

Оператор

```
r=a||b;
```

эквивалентен оператору

```
if (a!=0) r=1;
else  {
     if (b!=0) r=1;
     else  r=0;
}
```

«Ленивая» семантика логических операций позволяет, например, в задачах обработки массивов совмещать проверку выхода за границу массива и значения текущего элемента

массива в одном логическом выражении. В Паскале такое было невозможно. В самом деле, оператор

if i<=N **then if** mas[i]<>0 **then** s:=s+1

нельзя заменить на

if (i<=N)**and** (mas[i]<>0) **then** s:=s+1, так как правый операнд операции **and** может быть вычислен независимо от левого, и при i>N произойдет ошибка — выход за границу массива. Поскольку в Си вычисление логических операций идет слева направо с досрочным прекращением вычислений, аналогичные операторы на Си

`if (i<=N) if (mas[i]<>0) s=s+1;` и

`if (i<=N && mas[i]<>0) s=s+1;` эквивалентны: второй можно считать сокращенной записью первого.

В языке Си приоритет логических операций ниже, чем у отношений, поэтому скобки вокруг отношений в выражении `i<=N && mas[i]<>0` не нужны.

Покажем на примере сортировки массива простыми вставками возможность совмещения в заголовке цикла проверки границы массива и текущего элемента.

Сортировка простыми вставками

```
void insert_sort(int mas[], int N) {
  register int i,j, temp;
  for (i=1; i<N; i++){
     temp=v[i];
     for (j=i-1; j>=0 && v[j]>temp; j--)
            mas[j+1]=mas[j];

     mas[j+1]=temp;
  }
 }
```

В условии цикла `j>=0 && v[j]>temp` вычисление выражения `v[j]>temp` производится, только если истинно `j>=0`, поэтому выхода за границу массива не будет.

Переменная `i` во внешнем цикле for пробегает значения от `1` до `N-1` включительно. На шаге `i` все элементы от `mas[i]` до `mas[i-1]` уже отсортированы, и остается сортировать только элементы от `mas[i]` до `mas[N]`. Переменная `j` во внутреннем цикле уменьшается от `i-1`, при этом элементы массива перемещаются по одному вверх, пока не будет найдено место для помещения `mas[i]` – почему данный метод и называется методом вставок. Максимальное число выполняемых операций пропорционально `N*N`.

Об особенности операции отрицания

Выражение `x==!!x` (двойное логическое отрицание) может быть и истинным и ложным, т.е. не является тождеством в языке Си. (При `x`, равном 0 или 1 оно истинно, при остальных `x` ложно.)

Однако в условных конструкциях, например, `if`(⟨выражение⟩) или `while`(⟨выражение⟩) выражения `x` и `!!x` взаимозаменяемы, т.е. в этих конструкциях важно равно или не равно значение выражения нулю. Оба этих выражения принимают значение ноль только при `x` равном нулю. При других значениях `x` их значения могут отличаться друг от друга, но оба не равны нулю.

Операции отношения и равенства

Выражение

```
x==y==2
```

ложно в языке Си при любых x, y. Ему эквивалентно выражение (x==y)==2 в силу левой ассоциативности операции ==. Подвыражение (x==y) может принимать значения 0 или 1, ни одно из которых не равно 2. Чтобы сохранить на Си математический смысл двойного равенства, нужно привлечь логическую операцию И.

```
x==y && y==7
```

Смысл выражения

```
3<x<7
```

в языке Си также отличается от математического. Это выражение всегда истинно, так как оно эквивалентно в силу левой ассоциативности операции ”<” выражению (3<x)<7, а значениями подвыражения (3<x) могут быть только 0 или 1. Чтобы соблюсти математическую интерпретацию двойного неравенства, следует написать так:

```
3<x && x<7
```

Упражнение:

Эквивалентны ли выражения a<x == x< b и

```
a<x && x< b || b<=x && x<=a
```

?

Побитовые операции

Выражение

```
b << 4 >> 8
```

хотя и выглядит необычно, допустимо в языке Си. Операции сдвига левоассоциативны, поэтому выражение эквивалентно выражению

```
(b << 4) >> 8
```

В операциях сдвига происходит повышение целочисленности каждого операнда в отдельности, но нет балансировки. Если правый операнд отрицательный, результат неопределенный. Результат также не определен, если значение правого операнда превышает размер (в битах) значения левого операнда. При сдиге влево освобождающиеся битовые позиции заполняются нулями. При сдвиге вправо заполнение нулями гарантируется только для беззнаковых целых или неотрицательных знаковых, для отрицательных знаковых результат определяется реализацией.

Унарные операции + и –

Эти операции являются сокращенной записью выражений (0+x) и (0–x).

Операция sizeof

Результатом первичного выражения `sizeof(тип)` над заключенным в скобки именем типа будет размер объекта этого типа (в основной единице измерения, т.е. количество объектов типа char, которые в сумме занимают тот же объем памяти, что и данный объект).
По определению, `sizeof(char)` равно `1`.

Есть еще одноименная префиксная операция `sizeof` ⟨выражение⟩. Ее уровень в таблице приоритетов – 15.

При выполнении операции `sizeof` ⟨выражение⟩ само выражение не вычисляется. Только определяется его тип с учетом целочисленного повышения и балансировки.
`e= sizeof j++;` /* `j` – не изменяется */

Выражение
```
sizeof (long)-2
```

эквивалентно выражению

```
(sizeof (long))-2
```

а не

```
sizeof ((long)(-2)).
```

Тип, возвращаемый операцией `sizeof`, называется `size_t` . Он определяется в библиотечном файле `stddef.h` и во многих реализациях совпадает с типом `long` .

Операнд операции `sizeof` не должен иметь незавершенный тип или тип функции, за исключением ситуации, в которой `sizeof` выполняется над именем формального параметра, объявленного как массив, или функция. В этом случае, результат равен размеру указательного типа, полученного в результате обычного преобразования формальных параметров указанных типов.(Параметр `T a[]` преобразуется к `T* a`).
Операнд операции `sizeof` не может быть l-выражением, обозначающим битовое поле в структуре или объединении.

Вопросы по теме «выражения»

Может ли при каком-нибудь описании `p` считаться правильным выражение:
`++p++` ?
Может ли при каком-нибудь описании `a` считаться правильным выражение:
`&*a[i]++` ?
Может ли при каком-нибудь описании `c` и `foo` считаться правильным выражение:
`c[--foo]=c[foo++]` ?

.

Механизм передачи параметров функции в языке Си

Передача параметров функции в языке Си осуществляется только по значению. Каждый формальный параметр функции считается ее локальной переменной. При вызове функции вычисляются значения выражений, указанных в качестве аргументов (фактических параметров). Выполнение функции начинается с того, что создаются локальные переменные для формальных параметров. В эти переменные записываются значения соответствующих фактических параметров, приведенных к типу формальных параметров как при присваивании. Далее выполняется тело функции (составной оператор). Если в начале тела функции есть описания переменных, то эти переменные создаются. Созданные переменные существуют, пока функция не вернет значение в точку вызова с помощью оператора `return` или пока не завершится тело функции. В описании переменной может быть выражение-инициализатор. Тогда при создании переменной это выражение вычисляется и его значение присваивается переменной в качестве начального значения. Возвращаемое функцией значение приводится к типу, указанному в заголовке функции как тип возвращаемого значения. Функция может возвращать значение любого типа, кроме массива или функции (но может возвращать указатель на массив или указатель на функцию).

Пример. Пусть `a` и `b` – переменные типа `int`. Нужно описать функцию `swap( )`, которая должна поменять местами значения переменных `a` и `b`. Эту функцию можно было бы использовать, например, в сортировке обменами `bubble_sort( )` (метод пузырька) для обмена значений соседних элементов: `mas[j]` и `mas[j+1]`.

Если функцию `swap` описать таким образом

```
void swap(int x, int y)      /* НЕВЕРНОЕ  РЕШЕНИЕ */
{
 int temp; /* вспомогательная переменная */
 temp=x;
 x=y;
 y=temp;
}
```

то после вызова `swap(a,b)` значения переменных `a` и `b` останутся прежними, так как в функцию передаются не сами переменные, а их значения. Эти значения при вызове функции будут присвоены временным локальным переменным `x` и `y`, далее в теле функции происходит обмен значениями переменных `x` и `y`, после чего функция завершает свою работу, а все ее локальные переменные (`x`, `y`, `temp`) исчезают. Как мы видим, вызов `swap(a,b)` не оказал никакого влияния на переменные `a` и `b`.

Чтобы получить желаемый эффект, надо вызывающей программе передать *указатели* на те значения, которые должны быть изменены:

```
swap(&a, &b);
```

Операция `&` получает адрес переменной, и тип выражения `&a` есть указатель на `a`. В самой функции `swap` параметры должны быть описаны как указатели, при этом доступ к значениям параметров будет осуществляться через них косвенно.

В Паскале указатель на целое описывался как `px: ↑ integer`. В Си аналогичным описанием будет `int * px`. С учетом этого описание `swap` будет таким:

```
void swap(int *px, int *py)     /* перестановка *px и *py  */
{
  int temp;
  temp=*px;
  *px=*py;
  *py=temp;
}
```

Использование побитовых операций

Для целых чисел Стандарт Си требует от реализаций использование двоичного кодирования. Существуют операции, позволяющие оперировать с числами как с битовыми наборами. Наряду со знаковыми целыми, в Си можно работать и с беззнаковыми целыми. Беззнаковым аналогом типа `int` является тип `unsigned`.
Для опреаций сдвига, а также для побитовых операций (`^`, `|`, `&`, `~`) оба операнда должны быть целыми, просходит повышение операндов до `int` или `unsigned int`. Для побитовых (в отличие от сдвигов) типы операндов балансируются.

В качестве иллюстрации использования побитовых операций рассмотрим модификацию алгоритма Евклида.
Алгоритм Евклида вычисления НОД, использующий операцию взятия остатка %, записывается на Си следующим образом:

```
unsigned gcd(unsigned x, unsigned y)
{
 while (y!=0){
     unsigned temp=y;
     y=x%y;
     x=temp;
 }
 return x;
}
```

Так как деление с остатком реализуется с помощью вычитания, можно записать алгоритм Евклида с помощью только операции вычитания. Правда число итераций цикла в этой версии алгоритма будет большим, чем в предыдущей версии.

```
unsigned gcd(unsigned x, unsigned y)
{
 while (x!=y)
     if (x>y) x-=y; else y-=x;

 return x;
}
```

Рассмотрим еще одну модификацию — алгоритм Евклида вычисления НОД, использующий сдвиги и вычитание, известный как бинарный алгоритм Евклида.
Он может оказаться быстрее, чем алгоритм, использующий операцию взятия остатка %, если операция % в данной реализации работает существенно «медленнее», чем сдвиги и вычитания. Данный алгоритм отличается от обычного алгоритма Евклида с

использованием вычитания тем, что позволяет сократить число шагов, основываясь на двух свойствах:
1) НОД($2^k \cdot x$, $2^k \cdot y$) = $2^k \cdot$ НОД(x, y), где x **или** y — нечетное;
2) НОД($2^k \cdot x$, y) = НОД(x, y), где x **и** y — нечетные;

```
unsigned binary_gcd (unsigned x, unsigned y)
{
  unsigned temp;
  unsigned common_power_of_two = 0;
  if (x == 0) return y;   /* Особые случаи */
  if (y==0) return x;
   /* Определение наибольшей степени двух, делящей x и y */
  while  ((( x|y)&1)==0){
      x>>=1;          /* или по-другому:  x=x>>1;  */
      y>>=1;
      ++ common_power_of_two;
  }
  while ((x&1)==0) x>>=1;   /* можно   while (!(x&1)) x>>=1; */

      while(y) {
           /*  x – нечетно, y – ненулевое  */
          while((y&1) ==0) y>>=1;
           /*  x и y нечетные */
              temp=y;
              if  (x>y) y=x-y;
              else y=y-x;
              x=temp;
   /*        Теперь x равно прежнему значению y, которое нечетно.       Значение y
             четно, поскольку равно разности двух нечетных  значений; значит, в
             следующем проходе цикла оно будет сдвинуто вправо хотя бы на один бит
    */
       }
  return (x<< common_power_of_two);
}
```

Про массивы

Если массив не является (1) аргументом операции `sizeof`, (2) операндом адресной операции &, (3) строковой константой, инициализирующей символьный массив, то значение массива преобразуется (в процессе обычных унарных преобразований) в указатель на его первый элемент

Сложные описания

Сложные описания составляются с помощью комбинации имен типов с описателями и группировки их с помощью круглых скобок. К описателям относятся: * -- описатель указателя , [ ] --массива , ( ) – функции.

Например, чтобы задать «шестиэлементный массив указателей на функции, возвращающие значения целого типа», используется описание:

```
int (*m[6])( );
```

Важно, чтобы сложное описание давало корректный тип.

Типы, не являющиеся корректными :

1) Любой тип, включающий `void`, кроме «… функция, возвращающая значение типа `void`» или «указатель на `void`».
2) «Массив функций типа…». Массивы могут содержать указатели на функции, но не сами функции.
3) «Функция, возвращающая массив…». Функции могут возвращать указатели на массивы, но не сами массивы.
4) «Функция, возвращающая функцию…». Функции могут возвращать указатели на другие функции, но не сами функции.

Вместо того, чтобы писать

```
int *(*(*(*x)())[10])();
```

что означает, что переменная `x` имеет тип указателя на функцию, возвращающую указатель на 10-элементный массив указателей функций, возвращающих указатели на целые, можно написать так:

```
typedef   int * T4;

typedef  T4 (*T3) ();

typedef  T3 (*T2)[10];

typedef  T2 (*T1)();

T1 x;
```

Про оператор `switch`

Оператор `switch` – это конструкция, обеспечивающая множество вариантов перехода в зависимости от значения управляющего выражения.

`switch` (*выражение*) *оператор*

Управляющее *выражение* должно возвращать значение целого типа и может подвергаться повышению целочисленности (promoting). *Оператор*, входящий в состав конструкции `switch`, именуется телом оператора `switch`. Как правило (хотя и не всегда) это составной оператор. Любой оператор, располагающийся в теле оператора `switch`, в том числе и само тело, может быть помечен меткой `case` или `default`. Более того, один и тот же оператор может быть помечен несколькими метками `case` и, вдобавок, меткой `default`. Выражение, следующее за ключевым словом `case`, должно быть целым константным выражением. (Константное выражение вычисляется во время компиляции).

`case-метка`:

`case` *константное выражение*

Метка `case` или `default` считается принадлежащей самому внутреннему из вложенных операторов `switch`, в котором она расположена. Метки `case` и `default`

не могут располагаться вне оператора switch. Не может быть двух case-меток с одинаковым значением или двух меток default.

Порядок выполнения оператора switch:

1. Вычисляется управляющее выражение.

2. Если значение управляющего выражения совпадает со значением константного выражения одной из меток case, принадлежащих оператору switch, управление передается оператору, помеченному этой меткой, как это делается при выполнении оператора goto.

3. Если значение управляющего выражения не совпадает со значением константного выражение ни одной из меток case, но в операторе switch есть оператор, помеченный меткой default, управление передается этому оператору.

4. Если значение управляющего выражения не совпадает со значением константного выражение ни одной из меток case и в операторе switch нет метки default, ни один из операторов тела switch не выполняется, а управление передается оператору, следующему непосредственно за оператором switch.

Если при сравнении значений управляющего выражения и константного выражения метки case их типы не совпадают, происходит преобразование типа значения метки в тип управляющего выражения. Для прекращения выполнения оператора switch в любой точке его тела следует воспользоваться оператором break. Сравните следующие два примера.

```
switch(x) {

    case 1: printf(”*”);

    case 2: printf(”**”);

    case 3: printf(”***”);

}
```

При x равном 2 будет напечатано **пять** звездочек.

```
switch(x) {

    case 1: printf(”*”); break;

    case 2: printf(”**”); break

    case 3: printf(”***”); break;

}
```

При x равном 2 будет напечатано **две** звездочки.

В последнем операторе break из последнего примера нет необходимости, однако он может предотвратить ошибку при добавлении в эту конструкцию новых операторов.

В заключение приведем замысловатый, но эффективный и лаконичный фрагмент для решения следующей задачи. Пусть имеется функция prime(n), которая определяет, простое ли число n. Если число простое, его надо передать попдпрограмме prog_1, иначе – попдпрограмме prog_2. Известно, что для данного фрагмента малые числа – из первого десятка – будут встречаться значительно чаще, чем остальные.

```
switch(n)
  default:
  if (prime(n))
      case 2: case 3: case 5 : case 7:  prog_1(n);
  else
      case 4: case 6: case 8: case 9: case 10: prog_2(n);
```

Как видим, тело оператора switch не обязано быть составным оператором: {…}. Здесь мы имеем один условный оператор, помеченный меткой default. Особенностью языка Си, в отличие от Паскаля, является возможность перехода по метке внутрь сложных операторов (условных, циклов, составных). Для чисел от 2 до 10 в данном фрагменте происходит прямая передача управления на оператор, помеченный соответствующей case-меткой, для остальных попадаем на метку default и вычиселние начинается с проверки условия в операторе if.

Отметим, однако, что переход внутрь сложных операторов на case-метки в операторе switch, или аналогичный переход на обычные метки с помощью goto, существенно затрудняют понимание программы, нарушая концепцию структурного программирования. Кроме того, не выполняется начальная инициализация для объектов, находящихся в начале блока, внутрь которого произошел переход, хотя память под эти объекты все же отводится, например:

```
goto m;
…
{ int i=23; /* инициализация */
   …
  m:   /*  i не инициализирована, если попали сюда по goto  */

   …
}
```