любом случае *в функцию может быть передан массив произвольной длины. Никакой неявной информации о том, сколько элементов содержит массив, не передаётся.* Если вам нужна такая информация, вы должны ввести отдельный параметр.

```
1 double average(int n, double a[])
2 {
3         int    i;
4         double s = 0;
5
6         for (i = 0; i < n; i++)
7                 s += a[i];
8         return s / n;
9 }
```

1.8 Упражнения

1. Написать рекурсивную функцию вычисления чисел Фибоначчи.
2. Написать функцию, которая подсчитывает частоты распределения символов во входном потоке.

1 Занятие №2

1.1 Функции

На прошлом занятии нами был рассмотрен пример простой программы на языке **C**. Было сказано, что программа является совокупностью функций и глобальных переменны Кроме того, в программе должна присутствовать функция `main`, с которой начинается вы полнение программы.

Рассмотрим более подробно способы определения новых функций. Замечу, что в язы **Си** нет разделения на процедуры и функции, всякий объект, который может исполнятьс называется функцией.

Простейшее определение функции в общем виде выглядит следующим образом:

```
<result type> name(<params>) <compound-stmt>
```

Для обозначения того, что функция не возвращает значения, используется ключевое сл во **void**. Например,

```
void f(void)
{
        /* ... */
}
```

В телах void-функций может отсутствовать оператор **return**, тогда возврат из функц произойдёт после того, как будут выполнены все операторы. Либо оператор **return** мож присутствовать, но тогда не должен содержать возвращаемого выражения. Например,

```
1 void doprnt(int x)
2 {
3         if (!x) return;
4         printf("ERROR: %d\n", x);
5 }
```

Не void-функции должны завершаться оператором **return** содержащим выражени *Если вы на какой-то ветке забыли **return**, вам может быть выдано предупрежд ние, а может и нет, тогда в этом случае будет возвращено неопределённое знач ние.*

Функция может возвращать значения произвольного типа, *кроме массива*. Допустимы типы результата - это целые и вещественные типы, структуры (будем рассматривать), ук затели, перечислимые типы — кроме массива. Как мы позже увидим, массивы вообще явл ются «необычным» типом в **Си**.

Для обозначения того, что функция не имеет никаких аргументов, используется то ж самое ключевое слово **void**, которое записывается вместо параметров функции. Наприме

```
int f(void)
{
        /* ... */
}
```

Это функция, которая не имеет параметров и возвращает целое значение.

Обратите внимание, что пустой список параметров (т. е. `int f()`) обознач ет функцию, число и тип параметров которой неизвестны. Это не имеет смысла, ес

мы рассматриваем только определения функций, но очень заметно, когда функции, так объявленные, вызываются. В этом случае, компилятор не будет проверять соответствие количества и типов фактических параметров количеству и типам формальных параметров. Применяются специальные правила приведения типов. В одном месте программы такая функция может быть вызвана, скажем, с двумя целыми параметрами, а в другом месте — с тремя вещественными. Такое истолкование пустого списка параметров отличается от языка **Си++**, в котором это обозначает функцию без параметров.

Определения функций в «старом стиле» мы вообще не будем рассматривать.

Типы формальных параметров функции задаются следующим образом:

```
(<type1> <par1>, <type2> <par2>, ..., <typen> <parn>)
```

причём для каждого параметра каждый раз указывается его тип. `f(int a,b)` — неправильная запись.

Все параметры передаются по значению. Передачи параметров по ссылке не существует. Для получения нужного эффекта используются указатели. *Единственное исключение — массивы*, которые и здесь ведут себя по-другому.

Функции не могут быть определены внутри других функций. Поэтому программа на **Си** состоит из «плоских» функций и глобальных переменных.

Вызов функции в теле другой функции записывается естественным образом:

```
<имя>(<факт. параметры>)
```

Если у вызываемой функции нет параметров, пара скобок все равно должна быть указана. Если функция не **void**, полученное значение может дальше использоваться в выражении, *но может и игнорироваться*. а если **void**, то попытка использовать значение функции будет считаться ошибкой.

Когда в теле некоторой функции компилятор встречает конструкцию `name(<pars>)`, он считает это вызовом функции (не всегда!). Возможны три случая, в зависимости от кого, как было ранее определено имя `name`.

1. Если `name` была объявлена как функция с явным списком параметров, то проверяется соответствие количества и типов фактических параметров количеству и типу формальных параметров, при необходимости производится приведение типов. Тип возвращаемого значения определён самой функцией.

2. Если `name` была объявлена как функция с неявным (пустым) списков параметров, никакие проверки типов фактических параметров не производятся, делаются стандартные приведения типов. Тип возвращаемого значения определён самой функцией.

3. Если `name` *вообще не была раньше объявлена*, предполагается, что эта функция имеет определение `int name()`, то есть возвращает тип **int**, с неопределённым списком параметров. Если впоследствии функция будет объявлена с типом **int** и явным списком параметров, компилятор может выдать предупреждение. Если она будет объявлена не как **int** — ошибка.

Случай 3, рассмотренный выше, считается очень плохим стилем программирования. Мы будем полагать, что функция должна быть всегда описана перед её использованием. Для случая рекурсивных функций, или если желательно разместить функции в определённом порядке в файле, функцию перед использованием не обязательно полностью определять, т. е. описывать её тело, а достаточно описать. Для этого нужно задать имя функции, формальные параметры, возвращаемое значение, а тело функции опустить. Например,


```
a++, ++a, a--, --a
```

увеличивают (уменьшают) значение переменной на 1. Они применимы только к интегральным переменным.

Исправляя все ошибки нашего примера, получим такую программу:

```
 1 #include <stdio.h>
 2 enum { MAX_N = 1000000 };
 3 int main(void)
 4 {
 5   double s = 0;
 6   int    n;
 7
 8   if (scanf("%d", &n) != 1) {
 9     fprintf(stderr, "cannot␣read␣n\n");
10     return 1;
11   }
12   if (n <= 0 || n >= MAX_N) {
13     fprintf(stderr, "invalid␣value␣of␣n:␣%d\n", n);
14     return 1;
15   }
16
17   double x[n];
18   for (int i = 0; i < n; i++) {
19     if (scanf("%lf", &x[i]) != 1) {
20       fprintf(stderr, "cannot␣read␣x[%d]\n", i);
21       return 1;
22     }
23   }
24
25   for (int i = 0; i < n; i++) {
26     s += x[i];
27   }
28   s /= n;
29   printf("%.10g\n", s);
30   return 0;
31 }
```

1.7 Передача массивов в функции

При передаче параметров в функции и возврате значений из функций массивы ведут се особенным образом. Во-первых, как уже было сказано ранее, *функция не может вернут массив*. Во-вторых, в отличие от всех других типов языка, *массивы в функции передаютс по ссылке*. Это записывается следующим образом, например

```
double sum(double a[])
```

Обратите внимание, что *выражение в квадратных скобках опущено*. На самом деле, та может стоять произвольное выражение (константное), но оно ничего не будет значить.

1.4 Цикл for

```
for (<init expr>; <test expr>; <incr expr>) <stmt>
```

в точности совпадает с

```
<init expr>;
while (<test expr>) {
  <stmt>
  <incr expr>;
}
```

кроме того, в цикле for каждый из трёх элементов может быть опущен. Если опущено `<test expr>`, то считается, что вместо него стоит число 1, то есть цикл будет выполнятся всегда. В языке Си цикл **for** не нагружен дополнительными семантическими особенностями, в отличие от языка Паскаль, например, запретом изменять переменную цикла внутри цикла, или однократным вычислением предельного значения, или неопределённостью счётчика цикла после завершения выполнения цикла.

В новом стандарте C99 допускается объявление переменной, локальной для цикла, непосредственно в его заголовке. Например:

```
for (int i = 0; i < 10; i = i + 1) {
}
```

Переменная `i` будет видна только в теле цикла `for`.

1.5 Описание целых констант

Константы имеющие целое значение описываются следующим образом.

```
enum { <name> = <value> };
```

Например,

```
enum { N = 10 };
```

На самом деле это — определение перечислимого типа, содержащего единственную константу `N`, имеющую значение 10.

Константы, объявленные таким образом, подчиняются правилам видимости идентификаторов языка Си.

1.6 Сокращённые операции

```
a = a <op> b
```

можно заменить на

```
a <op>= b
```

Такие операции присваивания определены для всех бинарных операций, которые мы рассмотрели: `+, -, *, /, %`.

Инкрементные (декрементные) операции:


```
int gcd(int a, int b);
```

Описание функции также называется *прототипом*. После такого описания функцию мож
но использовать. Имена формальных параметров могут быть здесь опущены, или могут
совпадать с именами формальных параметров в месте действительного определения фун
ции — это не имеет значения. Например,

```
int gcd(int, int);
```

1.2 Определение глобальных переменных

Если переменная описана вне тела функции, эта переменная является глобальной,
есть доступна из всех точек программы, которые расположены ниже точки определения эт
переменной. Глобальная переменная может перекрываться локальной переменной с тем ж
именем.

Переменная в момент определения может быть инициализирована начальным значение
Например,

```
int a = 10;
double b = 1.0, c = 1.34e-4;
```

Переменная, описанная вне тела функции может быть инициализирована только ко
стантным выражением, то есть выражением, значение которого может быть вычислено
этапе компиляции. Такие переменные по умолчанию получают значение 0 соответствующе
типа. Блочные переменные могут инициализироваться произвольным выражением, но е
ли переменная не была инициализирована явно, *её начальное значение не определен*
Блочная переменная существует все время, пока выполняется блок. Как только управлен
покидает блок, переменная уничтожается и память под неё освобождается.

```
1  int x(int a)
2  {
3          int y = a + 2;
4          return g(y);
5  }
```

Для объектов языка Си действуют правила перекрытия имён, когда имя, объявленное
самом вложенном блоке относительно текущей точки определения, перекрывает имена, н
ходящиеся в объемлющих блоках или глобальные имена. Пример:

```
1  int x = 10;
2  void f(int y)
3  {
4    printf("%d\n", x);
5  }
6  int main(void)
7  {
8    int x = 15;
9    printf("%d\n", x);
10   {
11     int x = 20;
12     printf("%d\n", x);
13     f(x);
```

```
14    }
15    printf("%d\n", x);
16    return 0;
17  }
```

Для переменных разделяют два понятия: *время существования (жизни)* и *область видимости*. Область видимости — это все точки в исходном коде программы, в которых данная переменная может быть использована. Время жизни переменной — это отрезки времени при выполнении программы, когда под данную переменную выделена память. Например, переменная x, определённая в строке 8, существует, когда выполняются строки 8—16 (с учётом вызова вложенных функций), но видима только в строках 8, 9, 15, 16 исходной программы.

На одном и том же уровне вложенности не может быть определено двух объектов с одинаковым именем. Вопрос. Допустимы ли следующие определения:

```
int a(int a) { ... }
int a() { int a; ... }
int a(int a) { int a; ... }
```

1.3 Массивы

Рассмотрим первый составной тип — массив. Определение простейшего одномерного массива записывается следующим образом:

```
<тип> <имя> [ <число элементов> ]
```

В квадратных скобках указывается число элементов в массиве. Такое определение отводит память для массива из заданного числа элементов заданного типа, например

```
double val[40];
```

отводит память под массив из 40 элементов типа **double**.

Если определяется глобальный массив, то число элементов в массиве должно быть константным, то есть вычислимым при компиляции программы. Другими словами, оно не должно использовать значений переменных и результатов вызовов функций. Стандарт **C90** и для локальных массивов допускал только массивы константного размера, но в новом стандарте **C99** локальные массивы могут быть переменного размера, определяемого значением выражения в момент выделения памяти под переменную. Выражение должно принимать положительное целое значение.

```
1  int main(void)
2  {
3    int n;
4    scanf("%d", &n);
5    // ОШИБКА: обязательно нужно проверить n на допустимость
6    double v[n];
7    n = n + 1;
8    v[n - 1] = 0; // ОШИБКА: нет столько элементов
9  }
```

Обратите внимание, что если уже после создания массива значение выражения, определяющего его размер, изменилось, размер массива не изменится!

Для доступа к массиву используется запись вида <имя массива>[<индекс>], инде
может быть произвольным выражением. *Элементы массива всегда нумеруются от 0
до значения, равного количеству элементов в массиве минус 1.* Таким образом, эл
менты описанного выше массива перечисляются так:

```
val[0], val[1], val[2], ..., val[38], val[39].
```

Обратите внимание, что *выражение val[40] индексирует элемент, не принадлеж
щий массиву*. Это очень распространённая ошибка. *При обращении к элементам масс
ва контроль индексов не производится*, то есть можно написать, например, val[-5
val[60]. Компилятор не заметит никакой ошибки, ваша программа скомпилируется и п
выполнении будет давать неправильный результат.

Язык **Си** не имеет типа диапазона, поэтому и *переполнения при выполнении операци
с целыми числами тоже не контролируются*.

Рассмотрим пример — вычисление среднего арифметического вводимых чисел.

```
1  #include <stdio.h>
2  int main(void)
3  {
4    double x[10];
5    double s = 0;
6    int    i;
7    int    n;
8
9    scanf("%d", &n);
10   i = 0;
11   while (i < n) {
12     scanf("%lf", &x[i]);
13     i = i + 1;
14   }
15
16   i = 0;
17   while (i < n) {
18     s = s + x[i];
19     i = i + 1;
20   }
21   s = s / n;
22   printf("%.10g\n", s);
23   return 0;
24 }
```

Этот фрагмент содержит несколько ошибок и избыточных конструкций языка:

1. неудобная форма цикла **while**;
2. число 10, записанное в явном виде;
3. неудобная запись вида i = i + 1;
4. отсутствие проверки вводимых данных;
5. неявное ограничение на количество вводимых чисел.