Содержание go

Андрей Богатырев. Руководство полного идиота по программированию (на языке Си)

Email: abs@opentech.olvit.ru

ПЕРЕМЕННЫЕ

Переменная - это такой "ящичек" с именем,
в котором может храниться некое ЗНАЧЕНИЕ.
Имя у переменной постоянно и неизменно,
значение же может меняться.

Например, пусть у нас есть переменная с именем "x".

```
   -----
  / x /
 ---------------
 | Текущее     |
 | значение,   |
 | например 12 |
 ---------------
```

Переменную можно изменять при помощи операции ПРИСВАИВАНИЯ.
В языке Си она обозначается знаком равенства.

```
x = 12  ;
```

Это читается не как "икс равно 12",
а как "присвоить переменной икс значение 12",
то есть "Положить в ящик с надписью ИКС число 12".

Такая строка является простейшим ОПЕРАТОРОМ,
то есть ДЕЙСТВИЕМ. В конце операторов ставится точка с запятой.

Рассмотрим оператор

```
x = x + 3;
```

Это не уравнение. Если рассматривать эту строку как математическое
уравнение, оно не имеет решений. На самом деле тут написано:

```
1) "взять значение переменной ИКС"
2) "прибавить к нему 3"
3) "положить новое значение в переменную ИКС",
    стерев   в ней прежнее значение.
```

У опреатора присваивания есть две части: ЛЕВАЯ и ПРАВАЯ.

```
ЛЕВАЯ_ЧАСТЬ = ПРАВАЯ_ЧАСТЬ ;
```

В левой части обычно стоит просто имя переменной В КОТОРУЮ
записывается вычисленный справа результат.

Если имя переменной встречается в ПРАВОЙ части, то это
означает "подставить сюда текущее значение этой переменной".

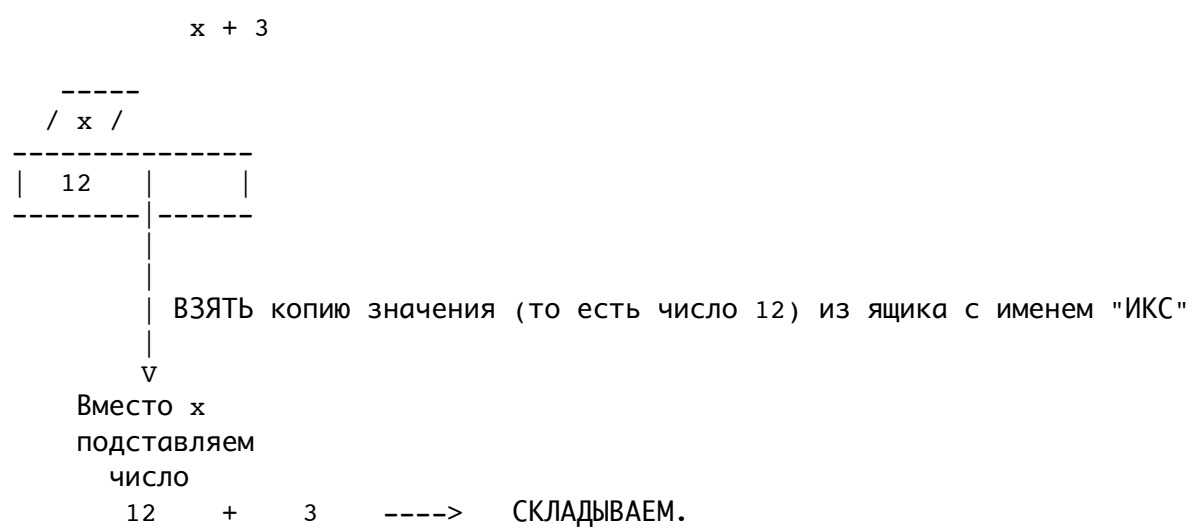
При этом текущее значение самой переменной НЕ изменяется,
берется его копия. То есть, "вынутое из ящика значение"
не оставляет ящик пустым! Вынимается копия, другая копия остается в ящике.

Итак:

```
x = x + 3 ;
```

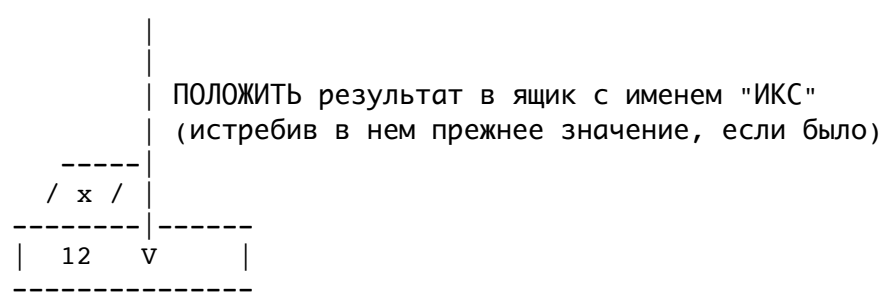
Пусть сейчас значение x есть 12

Сначала вычисляется ПРАВАЯ часть оператора присваивания.

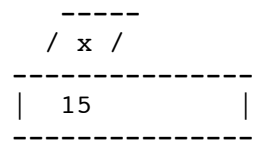


Теперь вычисляется само присваивание:

```
x = 15 ;
```



Стало:



В переменной с тем же именем теперь находится новое значение,
она изменилась. Потому и "переменная".

В некоторых языках программирования, например в Pascal или Modula,
операция присваивания обозначается символом := а не =
Это уменьшает путаницу, но к смыслу = можно привыкнуть довольно быстро.
Не огорчайтесь.

В правой части значение переменной может использоваться несколько раз:

```
z = x * x + 2 * x;
```

Тут есть две переменные:

```
z - для результата.
x - уже имеющая какое-то значение.

x * x означает "умножить икс на икс" (при этом само значение,
      лежащее в ящике икс не изменяется!)

x * 2 означает "взять два значения икс"

+     означает сложение.
```

Переменные надо ОБЪЯВЛЯТЬ.
Это необходимо потому, что иначе, если бы переменные вводились просто
использованием имени переменной, и мы вдруг допустили бы ОПЕЧАТКУ, например:

```
иднекс = 1;
```

вместо

```
индекс = 1;
```

то у нас появилась бы "лишняя" переменная "иднекс", а ожидаемое действие не
произошло бы. Такую ошибку найти черезвычайно тяжело. Если же переменные
надо объявлять, то необъявленные переменные будут выявлены еще на стадии
компиляции программы.

Переменные, которые будут хранить целые числа (..., -2, -1, 0, 1, 2, 3, ...),
объявляют так:

```
int переменная1;
int переменная2;
```

Или сразу несколько в одной строке:

```
int переменная1, переменная2;
```

int означает сокращение от слова integer - "целый".

ПРОГРАММА

Программа состоит из ОПЕРАТОРОВ, то есть действий.
Операторы выполняются последовательно в том порядке,
в котором они записаны.

```
/* ОБЪЯВЛЯЕМ ДВЕ ПЕРЕМЕННЫЕ */
int x, y;      /* 0 */

/* Это еще не операторы, хотя при этом создаются 2 ящика для
```

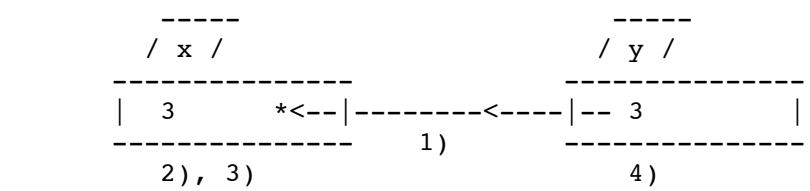
```
          целых чисел
        */

        /* А ТЕПЕРЬ - ОПЕРАТОРЫ. */
        /* Мы начнем с простых операторов присваивания и арифметики */

        x = 3;          /* 1 */
        y = 4;          /* 2 */
        x = x + y;      /* 3 */
        y = y - 1;      /* 4 */
        x = y;          /* 5 */
```

Значения переменных (то, что лежит в ящиках) меняются таким образом:

```
                          x       y
/* 0 */                 мусор   мусор

/* после 1 */           3       мусор
/* после 2 */           3       4
/* после 3 */           7       4
/* после 4 */           7       3
/* после 5 */           3       3
```

Как вы видите, переменные, которые не участвуют в левой части оператора присваивания, этим оператором НЕ МЕНЯЮТСЯ.

Последняя операция `x = y;` НЕ делает имена `x` и `y` синонимами.
Такой вещи, как "перевешивание табличек с именами с ящика на ящик" не происходит. Вместо этого, два ящика с именами `x` и `y` содержат одинаковые значения, то есть две копии одного и того же числа.



1) Из ящика `y` берется КОПИЯ числа `3` (безымянное значение).
2) Старое содержимое ящика `x` уничтожается.
3) Число `3` кладется в ящик `x`.
4) В исходном ящике `y` попрежнему осталось `3`.

Значение целой переменной можно вывести на экран оператором печати:

```
        printf("%d\n", x);
```

Пока будем рассматривать его как "магический".

Над целыми числами можно производить такие арифметические операции:

```
        x + y   сложение
        x - y   вычитание
        x * y   умножение
        x / y   деление нацело (то есть с остатком; результат - целое)
        x % y   вычислить остаток от деления нацело

        5 / 2 даст 2
        5 % 2 даст 1
```

В операторах присваивания используются такие сокращения:

ДЛИННАЯ ЗАПИСЬ СМЫСЛ СОКРАЩАЕТСЯ ДО

```
x = x + 1;              "увеличить на 1"        x++;    (или ++x; )
x = x - 1;              "уменьшить на 1"        x--;    (или --x; )
x = x + y;              "прибавить y"           x += y;
x = x * y;              "умножить на y"         x *= y;
x = x / y;              "поделить на y"         x /= y;
```

В том числе x++; можно записать как x += 1;

* СТРУКТУРЫ УПРАВЛЕНИЯ *

Обычно операторы выполняются последовательно,
в том порядке, в котором они записаны в программе.

```
оператор1;      |
оператор2;      |
оператор3;      |
оператор4;      V
```

УСЛОВНЫЙ ОПЕРАТОР

```
if(условие) оператор;

...продолжение...
```

Работает так:

Вычисляется условие.

Если оно истинно, то выполняется оператор,
затем выполняется продолжение.

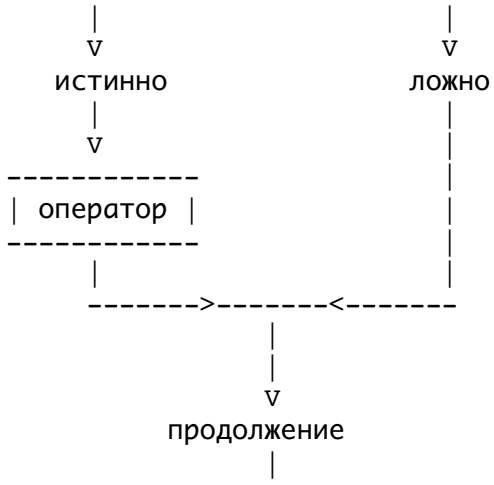
Если оно ложно, то сразу выполняется продолжение,
а оператор не выполняется.

Если нам надо выполнить при истинности условия несколько операторов,
мы должны заключить их в скобки { ... } - это так называемый
"составной оператор".

```
if(условие) {
        оператор1;
        оператор2;
        ...
}
продолжение
```

После } точка с запятой НЕ СТАВИТСЯ (можно и поставить, но не нужно).

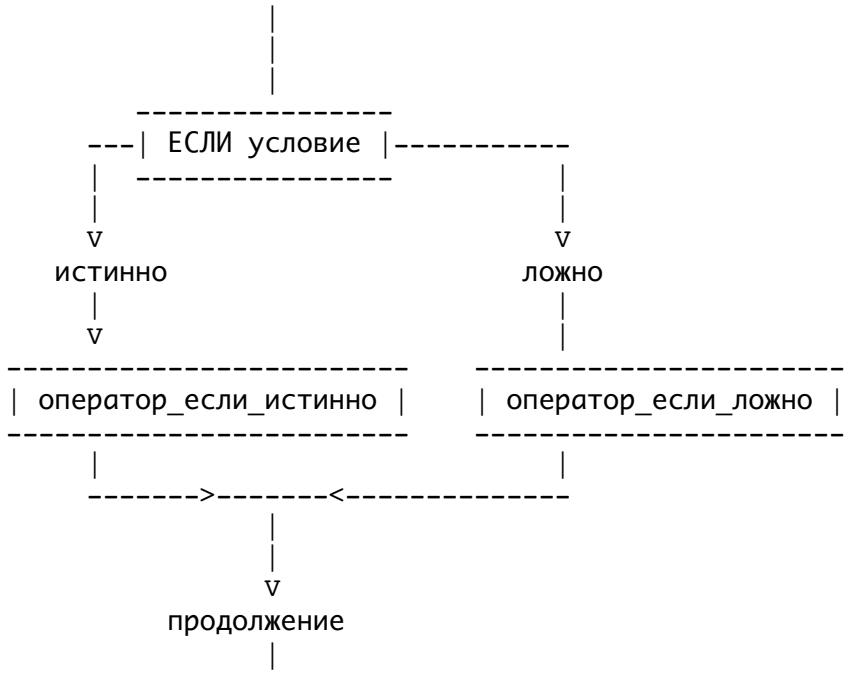
Условный оператор изображают на схемах так:



Имеется вторая форма, с частью "иначе":

```
if(условие) оператор_если_истинно;
else        оператор_если_ложно;
```

"или то, или другое" (но не оба сразу)



Пример1:

```
if(x > 10)
        printf("Икс больше десяти\n");
```

Пример2:

```
int x, y, z;

if(x < y)       z = 1;
else            z = 2;
```

Условия:

В качестве условий могут использоваться операторы СРАВНЕНИЯ
(сравнивать можно переменные, выражения, константы)

```
x <  y          меньше
```

```
x >  y          больше
x <= y          меньше или равно
x >= y          больше или равно
x == y          равно
x != y          не равно
```

Все эти операторы в качестве результата операции сравнения выдают
1, если сравнение истинно
0, если оно ложно.

Таким образом, на самом деле условный оператор работает так:

```
if(условие) ....
```

Если условие есть НОЛЬ - то условие считается ложным.
Если условие есть НЕ НОЛЬ а ... -2, -1, 1, 2, 3, ... - то условие истинно.

Это определение.

Из него в частности вытекает, что сравнение с целым нулем можно опускать:

```
if(x != 0) ... ;        сокращается до          if(x)  ... ;
if(x == 0) ... ;        сокращается до          if(!x) ... ;
```

Пример:

```
int x, y, z;

if(x == 1){ y = 2; z = x + y; }
else      { y = 1; z = x - y; }
```

Пример со вложенными условными операторами:

```
if(x == 1){
        printf("Икс равен 1\n");
        if(y == 2){
                printf("Игрек равен 2\n");
        }
} else {
        printf("Икс не равен 1\n");
}
```

Часто применяется последовательность условных операторов,
перебирающая различные варианты:

```
if(x == 1)
        printf("Икс равен 1\n");
else if(x == 2)
        printf("Икс равен 2\n");
else if(x == 3){
        printf("Икс равен 3\n");
        y = 1;
} else
        printf("Непредусмотренное значение икс\n");
```

Самое сложное - привыкнуть к тому, что сравнение обозначается знаком ==,
а не =
Знак = означает "присвоить значение", а не "сравнить на равенство".

ЦИКЛ while ("до тех пор, пока истинно")

```
        while(условие)
                оператор;
        ...продолжение...
```

или

```
        while(условие){
                операторы;
                ...
        }
        ...продолжение...
```

Пример:

```
        int x;

        x = 10;
        while(x > 0){
                printf("x=%d\n", x);
                x = x - 1;
        }
        printf("Конец.\n");
        printf("x стало равно %d.\n", x);       /* печатает 0 */
```

"Цикл" он потому, что его тело повторяется несколько раз.

Чтобы цикл окончился, оператор-тело цикла должен менять
какую-то переменную, от которой зависит истинность условия повторений.

ОПЕРАТОРЫ "И, ИЛИ, НЕ"

Условия могут быть сложными.

```
        ЕСЛИ красный И вес < 10 ТО ...;
        ЕСЛИ красный ИЛИ синий  ТО ...;
```

```
ЕСЛИ НЕ красный ТО ...;
```

На языке Си такие условия записываются так:

```
if(условие1 && условие2) ...;          /* "И"   */
if(условие1 || условие2) ...;          /* "ИЛИ" */
if(! условие1)           ...;          /* "НЕ"  */
```

Например:

```
if(4 < x && x <= 12) ...;
```

Было бы неправильно записать

```
if(4 < x <= 12) ...;
```

ибо язык программирования Си НЕ ПОНИМАЕТ двойное сравнение!

Еще примеры:

```
if(x < 3 || y > 4) ...;

if( ! (x < 3 || y > 4)) ...;
```

ЦИКЛ for ("для каждого")

Этот цикл является просто иной записью одного из вариантов цикла while.
Он служит обычно для выполнения опередеенного действия несколько раз,
не "пока истинно условие", а "выполнить N-раз".

У такого цикла есть "переменная цикла" или "счетчик повторений".

```
int i;

i = a;  /* начальная инициализация */

while(i < b){

        тело_цикла;

        i += c; /* увеличение счетчика */
}
...продолжение...
```

переписывается в виде

```
int i;

for(i=a; i < b; i += c)
        тело_цикла;
```

тело_цикла будет выполнено для значений i

```
a
a+c
a+c+c
...

пока i < b
```

В простейшем случае

```
for(i=1; i <= N; i++)
        printf("i=%d\n", i);
```

i означает "номер повторения".

Такой цикл служит для повторения СХОЖИХ действий НЕСКОЛЬКО раз
с разным значением параметра.

ОПЕРАТОР break ("вывалиться из цикла")

Оператор break заставляет прервать выполнение тела цикла
и сразу перейти к продолжению программы.

```
while(условие1){
        операторы1;

        if(условие2)
                break;  ------->----+
                                    |
        операторы2;                 |
}                                   |
...продолжение...<--------<---------+
```

и

```
for(i=0; условие1; i++){
        операторы1;

        if(условие2)
                break;  ------->----+
                                    |
        операторы2;                 |
}                                   |
...продолжение...<--------<---------+
```

Этот оператор позволяет организовывать дополнительные
точки выхода из цикла (при дополнительных условиях).

Пример:

```
for(i=0; i < 20; i++){
        printf("i=%d\n", i);
        if(i == 7){
                printf("break loop!\n");
                break;          /* вывалиться из цикла */
        }
        printf("more\n");
}
printf("finished, i=%d\n", i);  /* печатает 7 */
```

В частности, с его помощью можно организовывать бесконечный цикл:

```
for(;;){        /* заголовок бесконечного цикла */
        операторы1;

        if(условие2)
                break;  ------->----+
```

```
                                        |
                операторы2;             |
        }                               |
        ...продолжение...<--------<---------+
```

Здесь в самом заголовке цикла НЕ ПРОВЕРЯЕТСЯ НИКАКИХ УСЛОВИЙ,
такой цикл продолжается бесконечно.
Условие продолжения считается всегда истинным.

Единственный способ выйти из него -
это сделать `break` (при каком-то условии) в теле цикла, что и написано.

Бесконечный цикл можно также организовать при помощи

```
        while(1){
                ...
        }
```

ОПЕРАТОР ВЫВОДА (ПЕЧАТИ)

```
        printf("текст");
```

Печатает на экран текст.

```
        printf("текст\n");
```

Печатает на экран текст и переходит к новой строке.

```
        printf("слово1 слово2 ");
        printf("слово3\n");
```

печатает

слово1 слово2 слово3
и переходит на новую строку.

Если переход на новую строку не задан явно, символом `\n`,
то текст продолжает печататься в текущей строке.

```
        printf("%d", x);
```

Печатает в текстовом виде ЗНАЧЕНИЕ переменной `x`.
Специальная конструкция `%d` означает
"взять переменную из списка после запятой
 и напечатать ее значение в ивде целого числа".

```
        printf("икс равен %d - ого-го\n", x);
```

Печатает сначала текст

```
                икс равен
```

затем значение переменной `x` как целое число,
затем текст

```
                - ого-го
```

и переходит на новую строку (поскольку указан символ `\n`).

Этот оператор может печатать и несколько значений переменных:

```
        int x, y;

        x = 12; y = 15;
        printf("икс есть %d, игрек есть %d, все.\n", x, y);
                                                    ~~~~~~
```

Данный оператор работает так.
Строка "икс есть %d, игрек есть %d\n" называется ФОРМАТОМ.
Компьютер читает формат слева направо и печатает текст
до тех пор, пока не встретит символ %d.
Курсор изображен символом _

```
        икс есть _
```

Далее он берет ПЕРВУЮ переменную из списка ~~~~ и
печатает ее как целое число.

```
        икс есть 12_
```

далее он снова печатает текст пока не встретит %d

```
        икс есть 12, игрек есть _
```

Теперь он берет ВТОРУЮ переменную из списка и печатает ее:

```
        икс есть 12, игрек есть 15_
```

Снова печатает текст, включая перевод строки \n.
Как только строка формата кончилась, оператор printf завершен.

```
        икс есть 12, игрек есть 15, все.
        _
```

Печатать можно не только значения переменных, но и значения арифметических
выражений:

```
        printf("равно: %d\n", 12 + 3 * 5);
```

Контрольный вопрос, что печатается:

```
        int x, y, z;

        x = 13;
        y = 23;
        z = 34;

        printf("x=%d xx=%d\nzzz=%d\n", x, y - 1, z * 2 + 1);
```

Тут в формате есть ДВА перевода строки,
поэтому будет напечатано:

```
        x=13 xx=22
        zzz=69
        _
```

Заметьте, что перед тем как быть напечатанными,
выражения в списке после формата ВЫЧИСЛЯЮТСЯ.

Что напечатает

```
        printf("x=%d\n y=%d\n", x, y);
```

x=13

```
 y=23
_
```

Пробел перед y возник потому, что он СОДЕРЖИТСЯ
в строке формата после символа \n !!!
Будьте внимательны.

ФУНКЦИИ

Функцией называется фрагмент программы,
в который передаются ПАРАМЕТРЫ,
и который ВОЗВРАЩАЕТ значение (или ничего).

Прелесть функции в том, что ее можно выполнить много раз
из разных точек программы.

Функция состоит из

ОБЪЯВЛЕНИЯ - описания того, как она что-то вычисляет
Объявление бывает ровно одно.

ВЫЗОВОВ - с конкретными значениями параметров,
что именно она должна на этот раз вычислить.
Вызовов может быть сколько угодно.

Объявление простейшей функции выглядит так:

```
int func(int x){

        /* Один или несколько операторов,
           завершающихся оператором return(нечто);
         */

        return x+1;
}
```

```
int func(...
```

задает функцию с именем func
(имя выдумывает программист, как и имена переменных).

```
int
```
означает, что функция возвращает целое значение.

```
...(int x)...
```

задает список аргументов (или параметров) функции.

```
      ...){
      ...
}
```

задает тело функции - некую последовательность объявлений
переменных и операторов.

```
        return выражение;
```

задает оператор выхода из функции в точку ее вызова с возвратом значения

выражения.

Покажем простой пример ВЫЗОВА этой функции:

```
int y;
...
y = func(5);            /* a */
...продолжение...       /* b */
```

Этот фрагмент работает следующим образом:

```
y = func(5);
```

В этой точке мы

1) "записываем на бумажке",
   что вызов произошел в такой-то строке, таком-то месте
   нашей программы.

2) Смотрим на ОПРЕДЕЛЕНИЕ функции `func`.

```
 int func(int x){...
```

Мы вызвали функцию как `func(5)`.
Это значит, что в теле функции `x` получает начальное значение `5`.

То есть ДЛЯ ДАННОГО ВЫЗОВА наша функция (ее тело) превращается в

```
int x;

x = 5;
return x+1;
```

3) `x+1` есть `6`.

   Далее должен выполниться оператор `return`.

   Он выполняется так:

   Мы "читаем с бумажки" - откуда была вызвана функция `func`,
   и смотрим на это место. Это было

```
y = func(5);
```

   Вычеркиваем `func(5)` и заменяем его ЗНАЧЕНИЕМ выражения,
   вычисленного в операторе `return`;

```
y = 6;
```

4) Выполняем этот оператор и переходим к продолжению.

```
int y, z, w;

y = func(5);
z = func(6);
w = func(7) + func(8) + 1;
```

Превратится в

```
y = 6;
z = 7;
w = 8 + 9 + 1;
```

При этом мы четыре раза "прыгнем" на определение функции func(),
пройдем все ее операторы с разными значениями параметра x
и вернемся обратно в точку вызова.

ПРОГРАММА В ЦЕЛОМ

Программа в целом состоит из функций.
Одна из функций должна иметь имя main(),

С ФУНКЦИИ main НАЧИНАЕТСЯ ВЫПОЛНЕНИЕ ПРОГРАММЫ.

(на самом деле этому предшествует отведение и инициализация
глобальных переменных; смотри последующие лекции).

Часто main() - единственная функция в программе.

Структура программы такова:

```
#include <stdio.h>      /* магическая строка */

/* ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ (о них позже) */
int a = 7;
int b;          /* по умолчанию 0 */

/* ФУНКЦИИ */
f1(){....}
f2(){....}

/* НАЧАЛЬНАЯ (ГЛАВНАЯ) ФУНКЦИЯ */
void main(){
        ...
}
```

Пример программы:

```
#include <stdio.h>

int f1(int x, int y){
        return (x + y*2);
}

int f2(int x){
        int z;

        z = x+7;
        return 2*z;
}

void main(){
        /* Объявления переменных */
        int a, b, c;

        /* Операторы */
        a = 5; b = 6;

        c = f1(a, b+3);
        b = f1(1, 2);
        a = f2(c);
```

```
        printf("A есть %d B есть %d C есть %d\n", a, b, c);
}
```

Она печатает:

```
A есть 60 B есть 5 C есть 23
```

КАК НЕ НАДО ПРОГРАММИРОВАТЬ ЦИКЛЫ

```
        int i;

        for(i=0; i < 4; i++){
                if(i == 0)      func0();
                else if(i == 1) func1();
                else if(i == 2) func2();
                else if(i == 3) func3();
        }
```

В данном примере цикл АБСОЛЮТНО НЕ НУЖЕН.
То, что тут делается, есть просто ПОСЛЕДОВАТЕЛЬНОСТЬ операторов:

```
        func0();
        func1();
        func2();
        func3();
```

Цикл имеет смысл лишь тогда, когда много раз вызывается
ОДНО И ТО ЖЕ действие, но может быть зависящее от параметра, вроде `func(i)`.
Но не разные функции для разных `i`.

Аналогично, рассмотрим такой пример:

```
        int i;

        for(i=0; i < 10; i++){
                if(i==0)        func0();
                else if(i == 1) func1();
                else if(i == 2) func2();
                else            funcN(i);
        }
```

Тут `funcN(i)` берет на себя роль "а в остальных случаях".
Однако, этот пример более естественно может быть записан так:

```
        int i;

        func0();
        func1();
        func2();
        for(i = 3; i < 10; i++)
                funcN(i);
```

Заметьте, что цикл теперь начинается с индекса `3`.

А теперь - случай, где смесь цикла и условного оператора оправдана:

```
        int i;

        for(i=0; i < 100; i++){
                if((i % 2) == 0) even();        /* четный */
                else             odd();         /* нечетный */
```

```
        }
```

Тут в цикле проверяется четность индекса i.

03.c

```
/* Треугольник из звездочек */

#include <stdio.h>

/* putchar('c') - печатает одинокий символ c */
/* символ \n - переводит строку              */
/* nstars - сколько звездочек напечатать     */

/* Функция рисования одной строки треугольника */
void drawOneLine(int nstars){
        int i;          /* номер печатаемой звездочки, счетчик */

        for(i=0; i < nstars; i++)  /* Рисуем nstars звездочек подряд  */
                putchar('*');
        putchar('\n');             /* И переходим на следующую строку */
}

void main(){
        /* ОБЪЯВЛЕНИЕ ПЕРЕМЕННЫХ */
        int nline;      /* номер строки */

        /* ВЫПОЛНЯЕМЫЕ ОПЕРАТОРЫ (ДЕЙСТВИЯ) */
        for(nline=1; nline <= 25; nline++)
                drawOneLine(nline);
        /* сколько звездочек? столько же, каков номер строки */
}
```

04.c

```
/* Треугольник из звездочек */
/* Тот же пример со вложенным циклом, а не с функцией */

#include <stdio.h>

void main(){
        /* ОБЪЯВЛЕНИЕ ПЕРЕМЕННЫХ */
        int nline;      /* номер строки */
        int i;          /* номер печатаемой звездочки, счетчик */

        /* ВЫПОЛНЯЕМЫЕ ОПЕРАТОРЫ (ДЕЙСТВИЯ) */
        for(nline=1; nline <= 25; nline++){
                /* сколько звездочек? столько же, каков номер строки */
                for(i=0; i < nline; i++)
                        putchar('*');
                putchar('\n');
        }
}
```

05.c

```
/* Треугольник из звездочек */
/* Теперь треугольник должен быть равнобедренным */
```

```
#include <stdio.h>

/* nstars  - сколько звездочек напечатать     */
/* nspaces - сколько пробелов напечатать перед звездочками */

void drawOneLine(int nspaces, int nstars){
        int i;          /* номер печатаемой звездочки, счетчик */
                        /* он же - номер печатаемого пробела   */

        for(i=0; i < nspaces; i++)
                putchar(' ');
        for(i=0; i < nstars; i++)
                putchar('*');
        putchar('\n');
}

/*
                        n (номер строки)
     ...*               1
     ..***              2
     .*****             3
     *******            4

        Всего строк:                                    LINES
        Число звездочек в n-ой строке:                  n*2 - 1
        Число пробелов спереди (обозначены точкой):     LINES - n

        Все эти числа подсчитываются с картинки...

        Их мы будем передавать в функцию drawOneLine в точке _вызова_,
        а не вычислять в самой функции. Функция для того и заведена,
        чтобы не вычислять ничего КОНКРЕТНОГО -
        все параметры ее переменные, и должны ПЕРЕДАВАТЬСЯ в нее
        из точки вызова.

        В качестве параметра в точке вызова можно передавать не
        только значение переменной, но и значение выражения,
        то есть формулы.

*/
void main(){
        /* ОБЪЯВЛЕНИЕ ПЕРЕМЕННЫХ */
        int LINES = 25; /* всего строк.
                           Это описание переменной
                           сразу с ее инициализацией
                         */
        int nline;      /* номер строки */

        /* ВЫПОЛНЯЕМЫЕ ОПЕРАТОРЫ (ДЕЙСТВИЯ) */
        for(nline=1; nline <= LINES; nline++)
                drawOneLine(LINES - nline,   /* число пробелов  --> nspaces */
                            nline*2 - 1      /* число звездочек --> nstars */
                           );
}
```

06.c

```
/* Треугольник из звездочек */
/* Теперь треугольник должен быть равнобедренным */

#include <stdio.h>
```

```
void drawOneLine(int nspaces, int nstars){
        int i;          /* номер печатаемой звездочки, счетчик */
                        /* он же - номер печатаемого пробела   */

        for(i=0; i < nspaces; i++)
                putchar(' ');
        for(i=0; i < nstars; i++)
                putchar('*');
        putchar('\n');
}

void main(){
        /* ОБЪЯВЛЕНИЕ ПЕРЕМЕННЫХ */
        int LINES = 25; /* всего строк. */
        int nline;      /* номер строки */

        /* Для человека естественно считать с 1.
           Для машины же первое число - это НУЛЬ.
           Поэтому цикл
                for(nline=1; nline <= LINES; nline++)
           Следует записать в виде
                for(nline=0; nline <  LINES; nline++)

           Он тоже выполнится 25 раз, но значение переменной-счетчика
                nline будет на каждой итерации на 1 меньше. Поэтому надо
                поменять расчет параметров для функции рисования.

                        n (номер строки)
     ...*               0
     ..***              1
     .*****             2
     *******            3

        Всего строк:                                    LINES
        Число звездочек в n-ой строке:                  n*2 + 1
        Число пробелов спереди (обозначены точкой):     LINES - n - 1

        */

        /* ВЫПОЛНЯЕМЫЕ ОПЕРАТОРЫ (ДЕЙСТВИЯ) */
        for(nline=0; nline < LINES; nline++)
                drawOneLine(LINES - nline - 1, nline*2 + 1);
}
```

07.c

```
/*
        Тип переменных для хранения БУКВ называется

                char

        (от слова character).

        Буквы изображаются в одиночных кавычках 'a'  'b'   '+'.

        Пример:
                char letter;

                letter = 'a';
                putchar(letter);
                letter = 'b';
                putchar(letter);
```

```
                letter = '\n';
                putchar(letter);

        Символ '\n' обозначает "невидимую букву" -
        переход на новую строку, new line.
        Есть несколько таких специальных букв, о них - позже.

        Зато сразу сделаем оговорку.
        Чтобы изобразить саму букву \
        следует использовать '\\'

                putchar('\');   или
                printf ("\");   ошибочны.

        Надо:   putchar('\\');  printf("\\");

        Дело в том, что символ \ начинает последовательность из ДВУХ букв,
        изображающих ОДНУ букву, иногда вызывающую специальные
        действия на экране или на принтере.
*/

/*
        Число делится на n, если ОСТАТОК от деления его на n равен 0,
        то есть если

                        (x % n) == 0

        В частности, так можно проверять числа на четность/нечетность,
        беря x%2.

        Остатки от деления числа x на n
        это 0 1 2 ... n-1.
        В случае деления на 2 остаток
                0 соответствует четному   x
                1 соответствует нечетному x

*/

/* Задача:
        Нарисовать треугольник
        из звездочек в нечетных строках
        из плюсиков  в четных строках
 *-------------------------------------------------------*

 Решение: используем прежнюю программу,
 добавив в функцию drawOneLine еще один аргумент - symbol -
 каким символом рисовать строку.

 Далее в основном цикле используем условный оператор и
 проверку номера строки на четность.

*/

#include <stdio.h>

void drawOneLine(int nspaces, int nsymbols, char symbol){
        int i;          /* счетчик */

        for(i=0; i < nspaces; i++)
                putchar(' ');
        for(i=0; i < nsymbols; i++)
                putchar(symbol);
        putchar('\n');
```

```
}

/* Мы вынесем объявление этой переменной из функции,
   сделав ее "глобальной", то есть видимой во ВСЕХ функциях.
 */
int LINES = 25; /* всего строк. */

void main(){
        /* ОБЪЯВЛЕНИЕ ПЕРЕМЕННЫХ */
        int nline;      /* номер строки */

        /* ВЫПОЛНЯЕМЫЕ ОПЕРАТОРЫ (ДЕЙСТВИЯ) */
        for(nline=0; nline <  LINES; nline++){

                if((nline % 2) == 0)    /* четное ? */
                        drawOneLine(LINES - nline - 1, nline*2 + 1, '+');
                else    drawOneLine(LINES - nline - 1, nline*2 + 1, '*');
        }
}
```

08.c

```
/* То же самое, но теперь нужно еще и печатать номер строки.
 */

#include <stdio.h>

/* Вообще-то глобальные переменные
   принято объявлять в самом начале файла с программой.
 */

int LINES = 25; /* всего строк. */

/* Добавим к функции еще один аргумент, указатель - печатать ли
   номер строки. Назовем его drawLineNumber.
   Не впадите в заблуждение по аналогии с именем ФУНКЦИИ drawOneLine() !
   В данном случае - это имя ПЕРЕМЕННОЙ - АРГУМЕНТА ФУНКЦИИ.

   Оператор if(x) .....;
   РАБОТАЕТ ТАКИМ ОБРАЗОМ (так он устроен):
        в качестве условия он принимает целое число (типа int).
        Условие истинно, если x != 0,
        и ложно, если         x == 0.

   Второй добавленный аргумент - собственно номер строки.
*/
void drawOneLine(int nspaces,
                 int nsymbols,
                 char symbol,
                        /* а это мы добавили */
                 int drawLineNumber,
                 int linenum
){
        int i;          /* счетчик */

        if(drawLineNumber)
                printf("%d\t", linenum);  /* без перевода строки */

        /* На самом деле это условие более полно надо записывать как

              if(drawLineNumber != 0)
```

```
           но в языке Си это то же самое.
        */

/* Тут мы снова видим новый специальный символ \t - ТАБУЛЯЦИЯ.
   Весь экран (или лист бумаги) условно поделен
   на колонки шириной по 8 позиций.
   Примерно так:
|       |       |       |       |       |       |       |       ...
   Символ табуляции вызывает переход из текущей позиции в начало следующей
   колонки. Например
|       |       |       |       |       |       |       |       ...
                    ^ отсюда

|       |       |       |       |       |       |       |       ...
                        ^ в это место

*/
        for(i=0; i < nspaces; i++)
                putchar(' ');
        for(i=0; i < nsymbols; i++)
                putchar(symbol);
        putchar('\n');
}

void main(){
        /* ОБЪЯВЛЕНИЕ ПЕРЕМЕННЫХ */
        int nline;      /* номер строки */

        /* ВЫПОЛНЯЕМЫЕ ОПЕРАТОРЫ (ДЕЙСТВИЯ) */
        for(nline=0; nline <  LINES; nline++){

            if((nline % 2) == 0)    /* четное ? */
                    drawOneLine(LINES - nline - 1, nline*2 + 1, '+', 1, nline);
            else    drawOneLine(LINES - nline - 1, nline*2 + 1, '*', 9, nline);
        }

        /* А почему именно 1 или именно 9 ?
         * А все что попало, лишь бы не 0.
         * Можно 3, 333, 666, -13445, итп
         *
         * Вопрос: что будет, если тут написать 0 ?
         */
}
```

09.c

```
/* Следующая задача будет касаться того,
   чтобы каждая строка треугольника печаталась
   в виде:
        *+*+*+*.....*+*

   Тут нам уже придется модифицировать функцию рисования строки.
*/

#include <stdio.h>

int LINES = 25; /* всего строк. */

void drawOneLine(int nspaces, int nsymbols){
        int i;

        for(i=0; i < nspaces; i++)
```

```
                putchar(' ');

        /* в цикле мы будем проверять на четность НОМЕР
           печатаемого символа.
         */
        for(i=0; i < nsymbols; i++){
                if((i % 2) == 0)
                        putchar('*');
                else    putchar('+');
        }
        putchar('\n');
}

void main(){
        int nline;      /* номер строки */

        for(nline=0; nline <  LINES; nline++) {
                drawOneLine(LINES - nline - 1, nline*2 + 1);
        }
}
```

10.c

```
/* Задача нарисовать РОМБ:
        *
       ***
      *****
       ***
        *
*/

#include <stdio.h>

int LINES = 10; /* всего строк в половине ромба. */

void drawOneLine(int nspaces, int nsymbols){
        int i;

        for(i=0; i < nspaces; i++)
                putchar(' ');

        for(i=0; i < nsymbols; i++)
                putchar('+');
        putchar('\n');
}

void main(){
        int nline;      /* номер строки */

        for(nline=0; nline <  LINES; nline++)
                drawOneLine(LINES - nline - 1, nline*2 + 1);

        /* Мы нарисовали треугольник.
           Теперь нам нужен перевернутый треугольник.
           Пишем цикл по убыванию индекса.
           С данного места номера строк отсчитываются в обратном порядке:
           от LINES-2 до 0
         */

        for(nline=LINES-2; nline >= 0; nline--)
                drawOneLine(LINES - nline - 1, nline*2 + 1);
}
```

11.c

```
/* А теперь рисуем ромб, используя математические формулы. */

#include <stdio.h>

void draw(int nspaces, int nstars, char symbol){
        int i;

        for(i=0; i < nspaces; i++)
                putchar(' ');
        for(i=0; i < nstars; i++)
                putchar(symbol);
        putchar('\n');
}

void main(){
        int LINES = 21;
        int MIDDLELINE = LINES/2 + 1;   /* середина ромба */
        int nline;

        for(nline=0; nline < MIDDLELINE; nline++)
                draw(MIDDLELINE - nline -1, nline*2+1, 'A');

        /* У следующего цикла for() нет инициализации
           начального значения индекса.
           Начальное nline наследуется из предыдущего цикла,
           таким, каким оно осталось после его окончания, то есть
           равным MIDDLELINE.
         */

        for( ; nline < LINES; nline++)
                draw(nline - MIDDLELINE + 1, (LINES - 1 - nline) * 2 + 1, 'V');
}
```

* 12_ARRAYS.txt *

МАССИВЫ

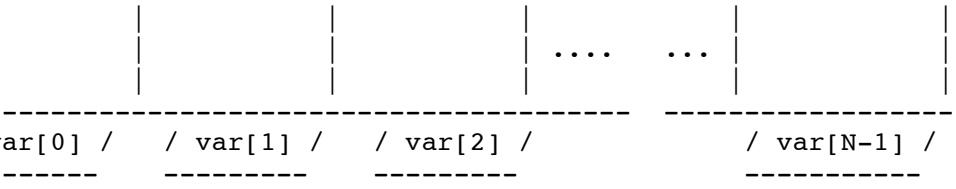
Массив - это несколько пронумерованных переменных,
объединенных общим именем.
Все переменные имеют ОДИН И ТОТ ЖЕ ТИП.

Рассмотрим ПОЛКУ с N ящиками,
пусть имя полки - var.
Тогда кажждый ящик-ячейка имеет имя

```
var[0]
var[1]
...
var[N-1]
```

Нумерация идет с НУЛЯ.

```
  --------
 /  var  /
/       /
----------------------------------------  -----------------
```



Массив объявляется так:

```
int var[N];
```

здесь N - его размер, число ячеек.

Это описание как бы объявляет N переменных типа int с именами
`var[0] ... var[N-1];`

В операторах для обращения к n-ому ящичку (где 0 <= n < N)
используется имя ящика

```
var[n]
```

где n - целое значение (или значение целой переменной,
или целочисленного выражения), "индекс в массиве".
Эта операция [] называется "индексация массива".
Индексация - есть ВЫБОР одного из N ящиков при помощи указания целого номера.

```
var    - массив (N ячеек)
n      - выражение (формула), выдающая целое значение в интервале 0..N-1
var[n] - взят один из элементов массива. Один из всех.
n      - номер ящика - называется еще и "индексом" этой переменной в массиве.
```

Пример:

```
int var[5];                                          /* 1 */

var[0] = 2;                                          /* 2 */
var[1] = 3 + var[0];                                 /* 3 */
var[2] = var[0] * var[1];                            /* 4 */
var[3] = (var[0] + 4) * var[1];                      /* 5 */

printf("var третье есть %d\n", var[3]);
```

В ходе этой программы элементы массива меняются таким образом:

```
          var[0]    var[1]     var[2]     var[3]    var[4]
          ------------------------------------------------
/* 1 */   мусор     мусор      мусор      мусор     мусор
/* 2 */   2         мусор      мусор      мусор     мусор
/* 3 */   2         5          мусор      мусор     мусор
/* 4 */   2         5          10         мусор     мусор
/* 5 */   2         5          10         30        мусор
```

Как видим, каждый оператор изменяет лишь ОДНУ ячейку массива за раз.

Массив - набор переменных, которые не ИМЕНОВАНЫ разными именами,
вроде var0, var1, var2, ...
а ПРОНУМЕРОВАНЫ под одним именем:
var[0], var[1], var[2], ...

Индекс - часть ИМЕНИ ПЕРЕМЕННОЙ.

На самом деле индексация - это

1) выбор элемента в массиве
2) справа от присваиваний и в выражениях - еще и разыменование,

то есть взятие вместо имени переменной - значения, в ней хранящегося.

Если в переменную не было занесено значение,
а мы используем эту переменную,
то в ней лежит МУСОР (любое, непредсказуемое значение).

```
printf("var4 есть %d\n", var[4]);
```

напечатает все что угодно.

Поэтому переменные надо всегда инициализировать
(давать им начальное значение).

Глобальные переменные автоматически инициализируются нулем,
если мы не задали иначе.

Локальные переменные не инициализируются автоматически, и содержат МУСОР.

Массивы НЕЛЬЗЯ присваивать целиком, язык Си этого не умеет.

```
int a[5];
int b[5];

a = b;  /* ошибка */
```

Также нельзя присвоить значение сразу всем элементам (ячейкам) массива:

```
a = 0;  /* ошибка */
```

не делает того, что нами ожидалось, а является ошибкой.
Для обнуления всех ячеек следует использовать цикл:

```
int i;

for(i=0; i < 5; i++)    /* для каждого i присвоить a[i] = 0; */
        a[i] = 0;
```

СВЯЗЬ МАССИВОВ И ЦИКЛОВ

Вследствие этого массивы приходится копировать (и инициализировать)
поэлементно, в цикле перебирая все (или часть) ячейки массива.

```
int i;

for(i=0; i < 5; i++)
        a[i] = b[i];
```

В данном случае индекс цикла служит также и индексом в массиве.

Индексы в массиве идут с НУЛЯ.

Пример инициализации:

```
int index, array[5];

for(index=0; index < 5; index++)
        array[index] = index * 2 + 1;
```

или

```
int index, array[5];
```

```
        index = 0;
        while(index < 5){
                array[index] = index * 2 + 1;
                index++;
        }

        /* В массиве будет: { 1, 3, 5, 7, 9 } */
```

ИНДЕКС

```
        для массивов -
                номер "ящика/ячейки" в массиве.

        для циклов -
                номер повторения цикла, счетчик.
                Мы должны изменять его САМИ.
```

Обычно массивы и циклы совмещаются так:
индекс цикла есть индекс в массиве;
то есть индекс цикла используется для перебора всех
элементов массива:

```
        int a[N], i;

        for(i=0; i < N; i++)
                ...a[i]...
```

Примеры:

```
        int a[5];

        a[0] = 17;
        a[0] += 4;
        a[0]++;
```

Пример: числа Фибоначчи.
Задаются математическими формулами:

```
        f[1]   = 1
        f[2]   = 1
        f[n+2] = f[n+1] + f[n]
```

Вот программа:

```
-----------------
#include <stdio.h>      /* магическая строка */
#define N 20            /* сколько первых чисел посчитать */

void main(){
        int fibs[N], index;

        fibs[0] = 1;    /* индексы отсчитываются с нуля!!! */
        fibs[1] = 1;

        /* Тут показано, что индекс элемента массива может вычисляться */

        for(index=2; index < N; index++)
                fibs[index] = fibs[index-1] + fibs[index-2];

        /* Распечатка в обратном порядке */
        for(index = N-1; index >= 0; index--)
                printf("%d-ое число Фибоначчи есть %d\n",
                        index+1,                  fibs[index]);
}
```

Здесь мы видим новый для нас оператор #define

Он задает текстуальную ЗАМЕНУ слова N на слово 20,
в данном случае просто являясь эквивалентом

```
const int N = 20;
```

К несчастью размер массива не может быть задан при помощи переменной,
а вот при помощи имени, определенного в #define - может.

СТРОКИ

Строки есть массивы БУКВ - типа char,
оканчивающиеся спецсимволом \0

```
char string[20];

string[0] = 'П';
string[1] = 'р';
string[2] = 'и';
string[3] = 'в';
string[4] = 'е';
string[5] = 'т';
string[6] = '\0';

printf("%s\n", string);
```

%s - формат для печати СТРОК.
Никакие другие массивы не могут быть напечатаны
целиком одним оператором.

```
char string[20];

string[0] = 'П';
string[1] = 'р';
string[2] = 'и';
string[3] = 'в';
string[4] = 'е';
string[5] = 'т';
string[6] = '\n';       /* Перевод строки - тоже буква */
string[7] = '\0';

printf("%s", string);
```

или даже просто

```
printf(string);
```

Такие массивы можно записать в виде строки букв в ""

```
char string[20] = "Привет\n";
```

Оставшиеся неиспользованными символы массива от string[8] до string[19]
содержат МУСОР.

ПОЧЕМУ ДЛЯ СТРОК ИЗОБРЕЛИ СИМВОЛ "ПРИЗНАК КОНЦА"?
==========
Строка - это ЧАСТЬ массива букв.
В разное время число букв в строке может быть различным,
лишь бы не превышало размер массива (тогда случится сбой программы).
Значит, следует где-то хранить текущую длину строки (число использованных

символов). Есть три решения:

(1) В отдельной переменной. Ее следует передавать во все функции обработки данной строки (причем она может изменяться).

```
char str[32];     /* массив для строки */
int  slen;        /* брать первые slen букв в этом массиве */
...
func(str, &slen); /* ДВА аргумента для передачи ОДНОЙ строки */
...
```

Этот подход работоспособен, но строка разбивается на два объекта: сам массив и переменную для его длины. Неудобно.

(2) Хранить текущую длину в элементе `str[0]`, а буквы - в `str[1]` ... итд.
Плохо тем, что в `str[0]` можно хранить лишь числа от `0` до `255`, и если строка длиннее - то такой подход неприменим.

(3) Не хранить длину НИГДЕ, а ввести символ-признак конца строки. Теперь в

```
func(str);      /* ОДИН аргумент - сам массив */
```

передается только сам массив, а его текущая длина может быть при нужде вычислена при помощи некоей функции, вроде такой:

```
int strlen(char s[]){                   /* функция от массива букв            */
        int counter = 0;                /* счетчик и одновременно индекс      */

        while(s[counter] != '\0')       /* пока не встретился признак конца текста */
                counter++;              /* посчитать символ                   */
        return counter;                 /* сколько символов, отличных от '\0' */
}
```

Тут никаких ограничений нет. Именно этот подход и был избран в языке Си, хотя в принципе можно самому пользоваться и другими. На самом деле в языке есть такая СТАНДАРТНАЯ функция `strlen(s)` (вам не надо писать ее самому, ее уже написали за вас).

ИНИЦИАЛИЗАЦИЯ ГЛОБАЛЬНОГО МАССИВА

Массив, заданный вне каких-либо функций, можно проинициализировать константными начальными значениями:

```
int array[5]   = { 12, 23, 34, 45, 56 };

char string[7] = { 'П', 'р', 'и', 'в', 'е', 'т', '\0' };
```

Если размер массива указан БОЛЬШЕ, чем мы перечислим элементов, то остальные элементы заполнятся нулями (для `int`) или `'\0'` для `char`.

```
int array[5] = { 12, 23, 34 };
```

Если мы перечислим больше элементов, чем позволяет размер массива - это будет ошибкой.

```
int a[5] = { 177, 255, 133 };
```

Операция индексации массива `a[]` дает:

при `n` значение выражения `a[n]` есть

```
-------------------------------------------
-1              не определено (ошибка: "индекс за границей массива")
0               177
1               255
2               133
3               0
4               0
5               не определено (ошибка)
```

* 13_FUNCS.txt *

КАК ПРОИСХОДИТ ВЫЗОВ ФУНКЦИИ

Пусть у нас описана функция, возвращающая целое значение.

```
        /* ОПРЕДЕЛЕНИЕ ФУНКЦИИ func(). */
        /* Где func - ее имя. Назвать мы ее можем как нам угодно. */

        int func(int a, int b, int c){
                int x, y;

                ...
                x = a + 7;
                ...
                b = b + 4;
                ...

                return(некое_значение);
        }
```

Здесь

```
        a, b, c     - аргументы функции (параметры)
        x, y        - локальные переменные
```

Точка вызова - находится внутри какой-то другой
функции, например функции main()

```
main(){

        int zz, var;
        ...
        var = 17;
        zz = func(33, 77, var + 3) + 44;
        ...
}
```

Когда выполнение программы доходит до строки

```
        zz = func(33, 77, var + 3) + 44;
```

1) Происходит ВЫЗОВ ФУНКЦИИ func()

(a) Этот пункт мы увидим ниже.

(b) Создаются переменные с именами a, b, c, x, y;

(c) Переменным-аргументам присваиваются начальные значения,
которые берутся из точки вызова.
В точке вызова перечислен список (через запятую) выражений (формул):

```
                func(выражение1, выражение2, выражение3)
```

Вычисленные значения этих выражений соответственно будут присвоены
1-ому, 2-ому и 3-ему аргументам (параметрам) из определения функции:

```
int func(a, b, c){      /* a = номер 1, b = 2, c = 3 */
```

Первый параметр:

```
a = 33;
```

Второй параметр:

```
b = 77;
```

Третий параметр:

```
c = var + 3;
```

то есть, вычисляя,

```
c = 20;
```

Локальные переменные x и y содержат неопределенные значения,
то есть мусор (мы не можем предсказать их значения,
пока не присвоим им явным образом какое-либо значение сами).

2) Выполняется ТЕЛО функции, то есть вычисления, записанные внутри { ... }
в определении функции. Например:

```
x = a + 7;
```

И параметры, и локальные переменные - это ПЕРЕМЕННЫЕ,
то есть их можно изменять.

```
b = b + 4;
```

При этом никакие переменные ВНЕ этой функции не изменяются.
(Об этом еще раз позже).

3) Производится ВОЗВРАТ из функции.

```
...
return(некое_значение);
}
```

Например, это может быть

```
...
return(a + 2 * x);
}
```

Рассмотрим, что при этом происходит в точке вызова:

```
zz = func(33, 77, var + 3) + 44;
```

(1) Вычеркиваем func(.....)

```
zz = XXXXXXX + 44;
```

(2) Вычисляем значение "некое_значение" в операторе return,
и берем КОПИЮ этого значения.
Пусть при вычислении там получилось 128.

(3) Подставляем это значение на место вычеркнутого func(.....)
У нас получается

```
zz = 128 + 44;
```

(4) АВТОМАТИЧЕСКИ УНИЧТОЖАЮТСЯ локальные переменные и аргументы функции:

```
a       - убито
b       - убито
c       - убито
x       - убито
y       - убито
```

Таких переменных (и их значений) больше нет в природе.

(5) Пункт, который мы обсудим позже.

(6) Продолжаем вычисление:

```
zz = 128 + 44;
```

Вычисляется в

```
zz = 172;       /* оператор присваивания */
```

```
int func1(int x){
        printf("func1: x=%d\n", x);     /* 1 */
        x = 77;
        printf("func1: x=%d\n", x);     /* 2 */
        return x;
}

void main(){
        int var, y;

        var = 111;
        y = func1(var);                 /* @ */

        printf("main: var=%d\n", var);  /* 3 */
}
```

В данном случае в точке @ мы передаем в функцию func1()
ЗНАЧЕНИЕ переменной var, равное 111.
Это значит, что при вызове функции будет создана переменная x
и ей будет присвоено начальное значение 111

```
x = 111;
```

Поэтому первый оператор printf() напечатает 111.

Затем мы изменяем значение переменной x на 77.
Мы меняем переменную x, но не переменную var !!!
Использовав ЗНАЧЕНИЕ (его копию) из переменной var для x,
мы о переменной var забыли - она нас не касается (а мы - ее).

Поэтому второй оператор printf() напечатает 77.
В переменной же var осталось значение 111,
что и подтвердит нам третий оператор printf,
который напечатает 111.

ВРЕМЕННОЕ СОКРЫТИЕ ПЕРЕМЕННЫХ
==========

```
int func1(int x){                       /* f.1 */
        printf("func1: x=%d\n", x);     /* f.2 */
        x = 77;                         /* f.3 */
        printf("func1: x=%d\n", x);     /* f.4 */
        return x;                       /* f.5 */
}

void main(){
        int x, y;                       /* 1 */

        x = 111;                        /* 2 */
        y = func1(x);                   /* 3 */

        printf("main: x=%d y=%d\n", x, y);  /* 4 */
}
```

А теперь мы и переменную внутри main(), и аргумент функции func1() назвали одним и тем же именем. Что будет?

Будет то же самое, что в предыдущем примере.

В момент вызова функции func1() будет создана НОВАЯ переменная с именем x, а старая (прежняя) переменная и ее значение будут ВРЕМЕННО СПРЯТАНЫ (скрыты).

Можно было бы уточнить эти переменные именами функций, в которых они определены:

```
        main::x

           и

        func1::x
```

(но это уже конструкции из языка Си++, а не Си).

Выполним программу по операторам:

```
|/* 1    */      Отводятся переменные main::x и main::y для целых чисел;
|/* 2    */      main::x = 111;
|/* 3    */      Вызывается func1(111);
|
+-------+
.       |/* f.1  */      Отводится переменная func1::x со значением 111;
.       |/* f.2  */      Печатается 111 из переменной func1::x;
.       |
.       |/* f.3  */      func1::x = 77; (это не main::x, а другая переменная,
.       |                                ЛОКАЛЬНАЯ для функции func1.
.       |                                Переменную main::x мы сейчас не видим -
.       |                                она "заслонена" именем нашей локальной
.       |                                переменной.
.       |                                Поэтому мы не можем ее изменить).
.       |
.       |/* f.4  */      Печатает 77 из func1::x;
.       |/* f.5  */      Возвращает значение func1::x , то есть 77.
.       |                Переменная func1::x уничтожается.
.       |
.       |                Теперь мы снова возвращаемся в функцию main(),
.       |                где имя x обозначает переменную main::x
.       |                а не func1::x
```

```
+-------+
|
|/* 3    */      y = 77;
|/* 4    */      Печатает значения main::x и main::y, то есть
|                111 и 77.
```

Этот механизм сокрытия имен позволяет писать функции `main()` и `func1()` разным программистам, позволяя им НЕ ЗАБОТИТЬСЯ о том, чтобы имена локальных переменных в функциях НЕ СОВПАДАЛИ. Пусть совпадают - хуже не будет, механизм упрятывания имен разрешит конфликт.
Зато программист может использовать любое понравившееся ему имя в любой функции - хотя бы и `x`, или `i`.

То же самое происходит с локальными переменными, а не с аргументами функции.

```
int func1(int arg){     /* локальная переменная-параметр func1::arg */
        int x;          /* локальная переменная          func1::x   */

        x = arg;
        printf("func1: x=%d\n", x);
        x = 77;
        printf("func1: x=%d\n", x);
        return x;
}

void main(){
        int x, y;       /* переменные main::x и main::y */

        x = 111;
        y = func1(x);

        printf("main: x=%d y=%d\n", x, y);
}
```

Действует тот же самый механизм временного сокрытия имени `x`.
Вообще же, аргументы функции и ее локальные переменные отличаются только одним:

> аргументам автоматически присваиваются начальные значения, равные значениям соответствующих выражений в списке
>
> ```
> имя_функции(..., ..., ....)
>             арг1 арг2 арг3
> ```
>
> в месте вызова функции.

То есть

ОПИСАНИЕ ФУНКЦИИ:

```
int f(int арг1, int арг2, int арг3){
        int перем1, перем2;
        ...
        /* продолжение */
}
```

ВЫЗОВ:

```
        .... f(выражение1, выражение2, выражение3) ...
```

ТО В ТЕЛЕ ФУНКЦИИ ВЫПОЛНИТСЯ (в момент ее вызова):

```
        арг1 = выражение1;
        арг2 = выражение2;
        арг3 = выражение3;

        перем1 = МУСОР;
        перем2 = МУСОР;

        ...
        /* продолжение */
```

ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ

Наконец, существуют переменные, которые объявляются ВНЕ ВСЕХ ФУНКЦИЙ,
и существующие все время выполнения программы
(а не только то время, когда активна функция, в которой они созданы).

Локальные переменные и аргументы УНИЧТОЖАЮТСЯ при выходе
из функции. Глобальные переменные - нет.

```
int x = 12;             /* ::x - ей можно заранее присвоить константу */
int globvar;            /* ::globvar                                  */

int f1(){
        int x;          /* f1::x */

        x = 77;
        printf("x=%d\n", x);            /* 4 */
        return x;
}

int f2(){
        printf("x=%d\n", x);            /* 5 */
        return 0;
}

void main(){
        int x, y;       /* main::x */

        x = 111;                         /* 1 */
        printf("x=%d\n", x);             /* 2 */
        printf("glob=%d\n", globvar);    /* 3 */

        y = f1();
        y = f2();
}
```

В данном примере мы видим:
- во-первых мы видим ФУНКЦИИ БЕЗ ПАРАМЕТРОВ. Это нормальная ситуация.
- во-вторых тут используются ТРИ переменные с именем "x".

Как выполняется программа?

```
/* 1 */   main::x = 111;
          Это локальный x, а не глобальный.
          Глобальный x попрежнему содержит 12.

/* 2 */   Напечатает значение переменной main::x, то есть 111.
          Внутри функции main глобальная переменная ::x
          заслонена своей собственной переменной x.
          В данном случае НЕТ СПОСОБА добраться из main к глобальной
```

переменной x, это возможно только в языке Си++ по имени ::x

К переменной же globvar у нас доступ есть.

/* 3 */ Печатает ::globvar. Мы обнаруживаем, что ее значение 0.
В отличие от локальных переменных,
которые изначально содержат МУСОР,
глобальные переменные изначально содержат значение 0.

В рамочку, подчеркнуть.

/* 4 */ При вызове f1()
переменная f1::x
заслоняет собой как
main::x
так и
::x

В данном случае напечатается 77,
но ни ::x ни main::x не будут изменены оператором x = 77.
Это изменялась f1::x

/* 5 */ При вызове f2() история интереснее.
Тут нет своей собственной переменной x.
Но какая переменная печатается тут -
::x или
main::x ?

Ответ: ::x
то есть 12.

Переменные названы локальными еще и потому,
что они НЕВИДИМЫ В ВЫЗЫВАЕМЫХ ФУНКЦИЯХ.

Это ОПРЕДЕЛЕНИЕ локальных переменных.
(Поэтому не спрашивайте "почему?" По определению)

То есть, если мы имеем

```
funca(){
        int vara;
        ...
        ...funcb();...  /* вызов */
        ...
}
```

то из функции funcb() мы НЕ ИМЕЕМ ДОСТУПА К ПЕРЕМЕННОЙ vara.

```
funcb(){
        int z;

        z = vara + 1;   /* ошибка,
                           vara неизвестна внутри funcb() */
}
```

Если, в свою очередь, funcb() вызывает funcc(),
то и из funcc() переменная vara невидима.

Остановитесь и осознайте.
Это правило служит все той же цели - разные функции
могут быть написаны разными программистами, которые могут
использовать одни и те же имена для РАЗНЫХ переменных,

не боясь их взаимопересечения.
Множества имен, использованных в разных функциях, независимы
друг от друга. Имена из одной функции НИКАК не относятся
к переменным с теми же именами ИЗ ДРУГОЙ функции.

Вернемся к параграфу КАК ПРОИСХОДИТ ВЫЗОВ ФУНКЦИИ
и рассмотрим пункт (a). Теперь он может быть описан как

(a) Локальные переменные и аргументы вызывающей функции делаются невидимыми.
~~~~~~~~~~

А при возврате из функции:

(5) Локальные переменные и аргументы вызывающей функции снова делаются видимыми.

ОДНАКО глобальные переменные видимы из ЛЮБОЙ функции,
исключая случай, когда глобальная переменная заслонена
одноименной локальной переменной данной функции.

ПРОЦЕДУРЫ
=========
Бывают функции, которые не возвращают никакого значения.
Такие функции обозначаются void ("пустышка").
Такие функции называют еще ПРОЦЕДУРАМИ.

```
void func(){
        printf("Приветик!\n");
        return;  /* вернуться в вызывающую функцию */
}
```

Такие функции вызываются ради "побочных эффектов",
например печати строчки на экран или изменения глобальных (и только)
переменных.

```
int glob;

void func(int a){
        glob += a;
}
```

Оператор return тут необязателен, он автоматически выполняется
перед последней скобкой }

Вызов таких функций не может быть использован
в операторе присваивания:

```
main(){
        int z;

        z = func(7);     /* ошибка, а что мы присваиваем ??? */
}
```

Корректный вызов таков:

```
main(){
        func(7);
}
```

Просто вызов и все.

ЗАЧЕМ ФУНКЦИИ?
~~~~~~~~~~

Чтобы вызывать их с разными аргументами!

```
int res1, res2;
        ...

res1 = func(12 * x * x + 177, 865,     'x');
res2 = func(432 * y + x, 123 * y - 12, 'z');
```

Кстати, вы заметили, что список фактических параметров
следует через запятую;
и выражений ровно столько, сколько параметров у функции?

Функция описывает ПОСЛЕДОВАТЕЬНОСТЬ ДЕЙСТВИЙ,
которую можно выполнить много раз,
но с разными исходными данными (аргументами).
В зависимости от данных она будет выдавать разные результаты,
но выполняя одни и те же действия.

В том то и состоит ее прелесть:
мы не дублируем один кусок программы много раз,
а просто "вызываем" его.

Функция - абстракция АЛГОРИТМА, то есть последовательности действий.
Ее конкретизация - вызов функции с уже определенными параметрами.

Оператор `return` может находиться не только в конце функции,
но и в ее середине.
Он вызывает немедленное прекращение тела функции и возврат значения
в точку вызова.

```
int f(int x){
        int y;

        y = x + 4;
        if(y > 10) return (x - 1);
        y *= 2;
        return (x + y);
}
```

РЕКУРСИВНЫЕ ФУНКЦИИ. СТЕК

Рекурсивной называется функция, вызывающая сама себя.

```
int factorial(int arg){
        if(arg == 1)
                return 1;                               /* a */
        else
                return arg * factorial(arg - 1);        /* b */
}
```

Эта функция при вызове `factorial(n)` вычислит произведение

```
n * (n-1) * ... * 3 * 2 * 1
```

называемое "факториал числа n".
В данной функции переменная `arg` будет отведена (а после и уничтожена) МНОГО РАЗ.

Так что переменная `factorial::arg` должна получить еще и НОМЕР вызова функции:

```
factorial::arg[уровень_вызова]
```

И на каждом новом уровне новая переменная скрывает все предыдущие.
Так для `factorial(4)` будет

```
            +---------------------------------------+
    |       | factorial::arg[ 0_ой_раз ]  есть 4    |
    |       | factorial::arg[ 1_ый_раз ]  есть 3    |
    |       | factorial::arg[ 2_ой_раз ]  есть 2    |
    |       | factorial::arg[ 3_ий_раз ]  есть 1    |
    V  ---------+                                    +---------
```

Затем пойдут возвраты из функций:

```
            +---------------------------------------+
    A       | /* b */ return 4 * 6 = 24             |
    |       | /* b */ return 3 * 2 = 6              |
    |       | /* b */ return 2 * 1 = 2              |
    |       | /* a */ return 1;                     |
    |  --------+                                    +---------
```

Такая конструкция называется СТЕК `(stack)`.

```
--------+              +------------
        |              |                      пустой стек
        +--------------+
```

Положим в стек значение `a`

```
                |
--------+       |      +------------
        |       V      |
        +--------------+
        |       a      | <--- вершина стека
        +--------------+
```

Положим в стек значение `b`

```
--------+              +------------
        |              |
        +--------------+
        |       b      | <--- вершина стека
        +--------------+
        |       a      |
        +--------------+
```

Положим в стек значение `c`

```
--------+              +------------
        |              |
        +--------------+
        |       c      | <--- вершина стека
        +--------------+
        |       b      |
        +--------------+
        |       a      |
        +--------------+
```

Аналогично, значения "снимаются со стека" в обратном порядке: `c, b, a`.

В каждый момент времени у стека доступно для чтения (копирования) или выбрасывания только данное, находящееся на ВЕРШИНЕ стека.

Так и в нашей рекурсивной функции переменная factorial::arg
ведет себя именно как стек (этот ящик-стек имеет имя arg) -
она имеет ОДНО имя, но разные значения в разных случаях.
Переменные, которые АВТОМАТИЧЕСКИ ведут себя как стек,
встречаются только в (рекурсивных) функциях.

Стек - это часто встречающаяся в программировании конструкция.
Если подобное поведение нужно программисту, он должен промоделировать
стек при помощи массива:

```
int stack[10];
int in_stack = 0;       /* сколько элементов в стеке */

/* Занесение значения в стек */
void push(int x){
        stack[in_stack] = x;
        in_stack++;
}

/* Снять значение со стека */
int pop(){
        if(in_stack == 0){
                printf("Стек пуст, ошибка.\n");
                return (-1);
        }
        in_stack--;
        return stack[in_stack];
}
```

Обратите в нимание, что нет нужды СТИРАТЬ (например обнулять)
значения элементов массива, выкинутых с вершины стека.
Они просто больше не используются, либо затираются новым значением при
помещении на стек нового значения.

```
void main(){
        push(1);
        push(2);
        push(3);

        while(in_stack > 0){
                printf("top=%d\n", pop());
        }
}
```

СТЕК И ФУНКЦИИ

Будем рассматривать каждый ВЫЗОВ функции как помещение в специальный стек
большого "блока информации", включающего в частности
АРГУМЕНТЫ И ЛОКАЛЬНЫЕ ПЕРЕМЕННЫЕ вызываемой функции.

Оператор return из вызванной функции выталкивает со стека ВЕСЬ такой блок.

В качестве примера рассмотрим такую программу:

```
int x = 7;      /* глобальная */
int v = 333;    /* глобальная */

int factorial(int n){
        int w;  /* лишняя переменная, только для демонстрационных целей */
```

```
        w = n;

        if(n == 1)      return 1;                       /* #a */
        else            return n * factorial(n-1);      /* #b */
}
void func(){
        int x;          /* func::x */

        x = 777;        /* #c */
        printf("Вызвана функция func()\n");
}
void main(){
        int y = 12;     /* main::y */
        int z;

                        /* A */
        z = factorial(3);       /* B */
        printf("z=%d\n", z);

        func();                 /* C */
}
```

Выполнение программы начнется с вызова функции main().
В точке /* A */

```
                        |    в  з  г  л  я  д   |
                        V                       V

        --------+                       +--------
                |=======================|
                |       z = мусор       |
                |       y = 12          |           +------+---------+
                |main()                 |           |x = 7 | v = 333 |
                +-----------------------+-----------+------+---------+-----
                 СТЕК ВЫЗОВОВ ФУНКЦИЙ                 ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ
```

В каждый данный момент видимы переменные, которые находятся
a) на вершине (и только) стека вызовов функций.
b) и незаслоненные ими глобальные переменные.
В данном случае мы смотрим "сверху" и видим:

```
        main::z,  main::y,  ::x,  ::v
```

В точке /* B */ мы вызываем factorial(3).

```
        --------+                       +--------
                |=======================|
                |       w = мусор       |
                |       n = 3           |
                |factorial(3)           |
                |=======================|
                |                     +-|---> текущий оператор z = factorial(3);
                |       z = мусор       |
                |       y = 12          |           +------+---------+
                |main()                 |           |x = 7 | v = 333 |
                +-----------------------+-----------+------+---------+-----
                 СТЕК ВЫЗОВОВ ФУНКЦИЙ                 ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ
```

При новом взгляде видимы:

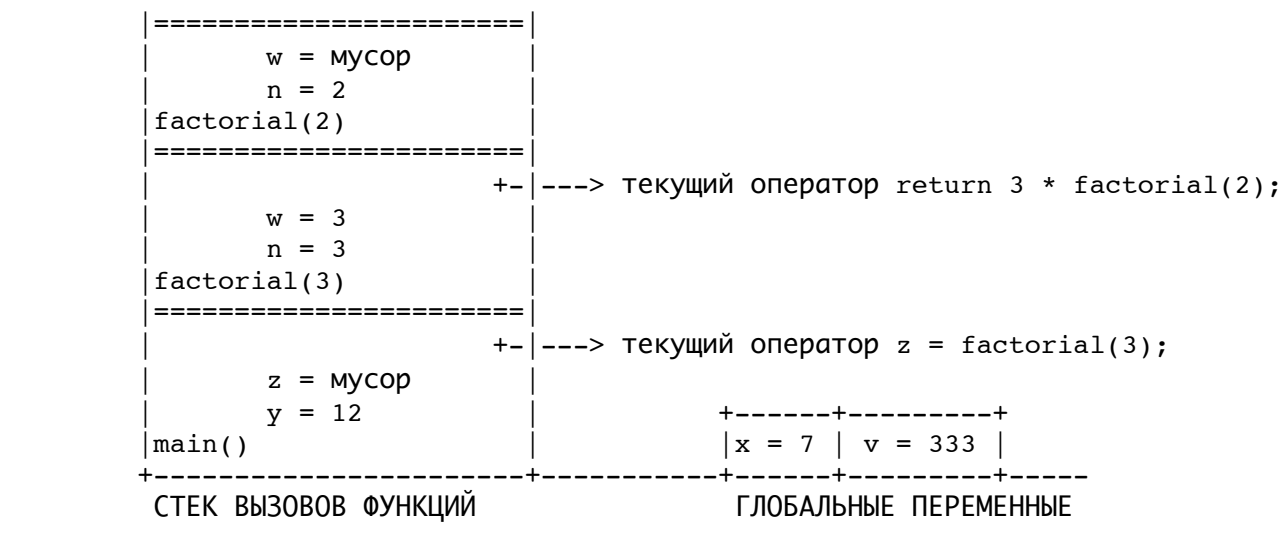
```
        factorial(3)::w,  factorial(3)::n,  ::x,  ::v
```

Стали невидимы:

```
        main::z,  main::y
```

Строка "текущий оператор ..." указывает место, с которого надо возобновить
выполнение функции, когда мы вернемся в нее.

Когда выполнение программы в функции `factorial(3)` дойдет до точки
`/* #b */` будет выполняться `return 3 * factorial(2)`.
В итоге будет вызвана функция `factorial(2)`.

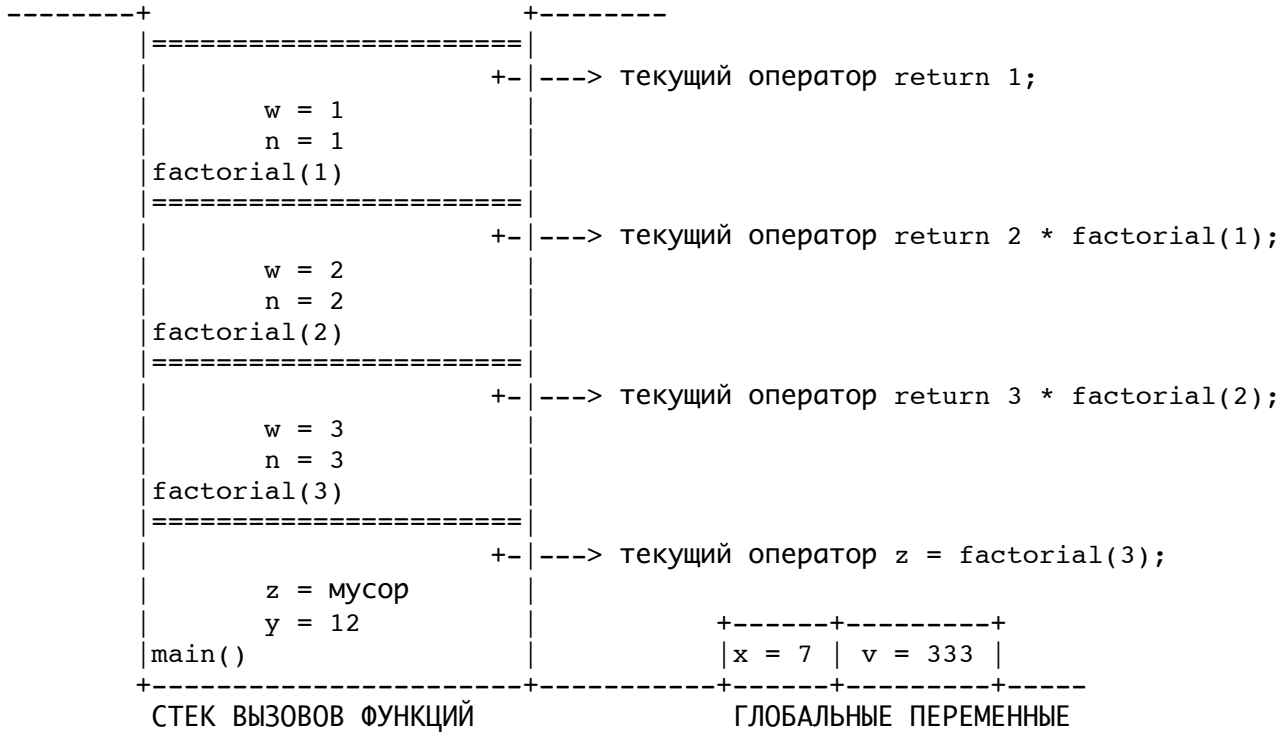


Когда выполнение программы в функции `factorial(2)` дойдет до точки
`/* #b */` будет выполняться `return 2 * factorial(1)`.
В итоге будет вызвана функция `factorial(1)`.

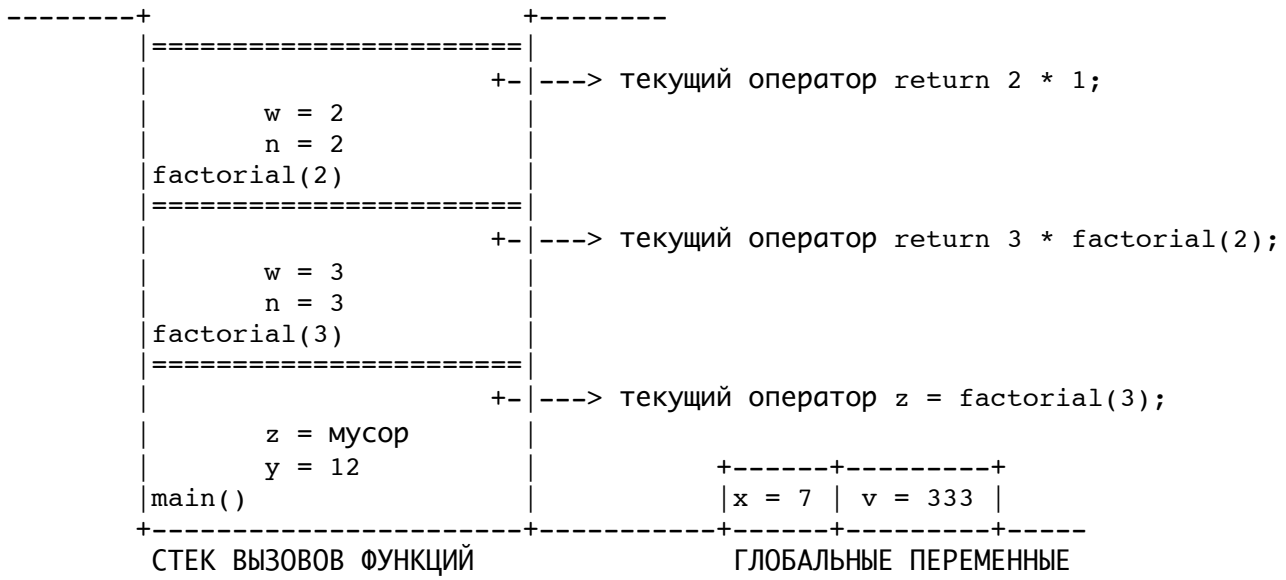
Затем в `factorial(1)` выполнение программы дойдет до точки `/* #a */`
и будет производиться `return 1`.

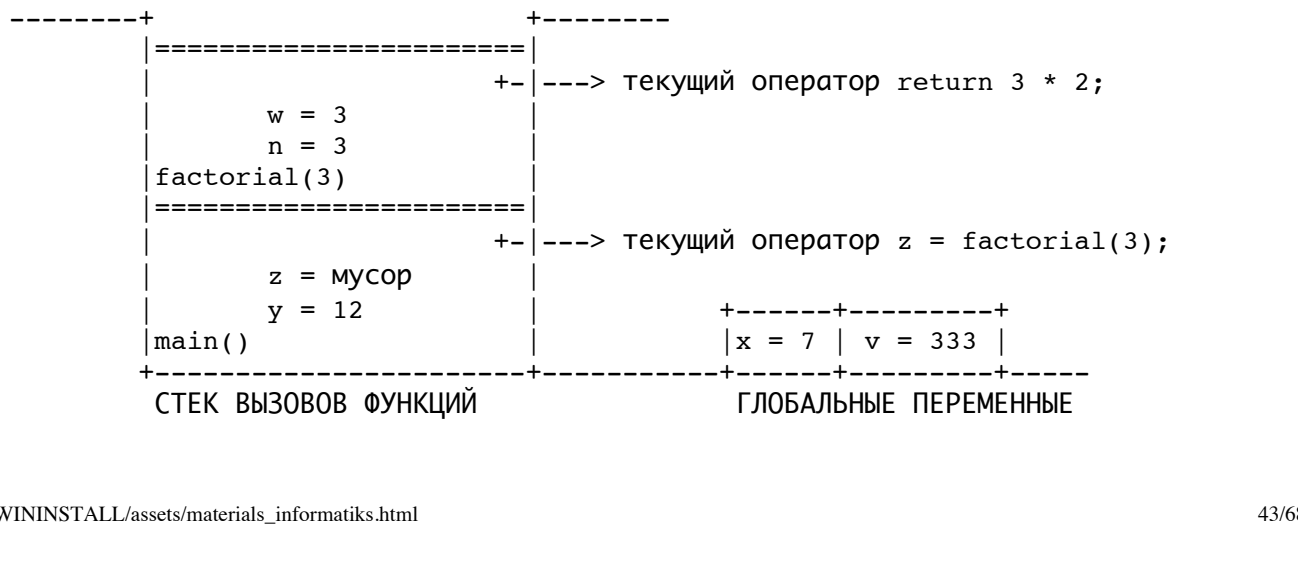
При `return` вычеркивается ОДИН блок информации со стека вызовов функций,
и возобновляется выполнение "текущего оператора" в функции,
ставшей НОВОЙ вершиной стека вызовов.
Заметьте, что в данной ситуации вызванные функции `factorial(m)` еще не завершились.

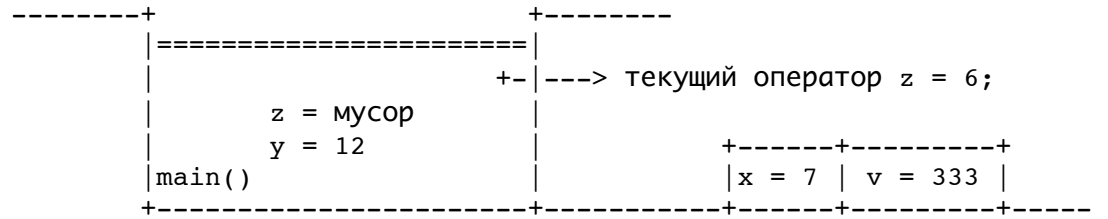
В них еще ЕСТЬ что сделать: вычислить выражение в return,
и собственно выполнить сам return. Вычисления будут продолжены.

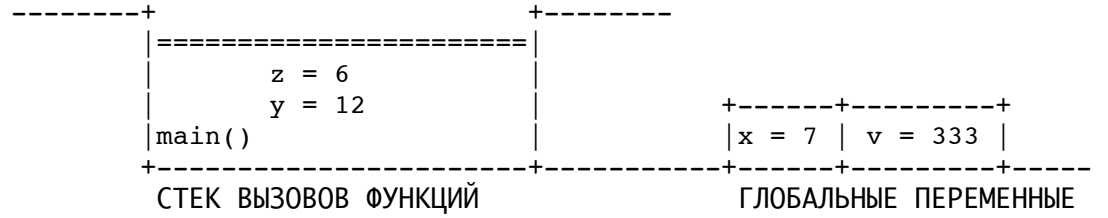


Начинается выталкивание функций со стека и выполнение операторов return;

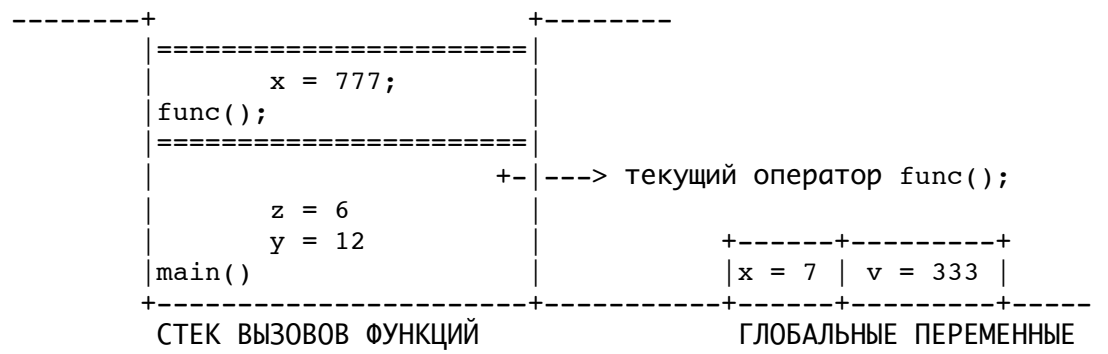








Наконец, в точке /* c */ будет вызвана функция func().
Рассмотрим точку /* #c */ в ней.



В данном месте нас интересует - какие переменные видимы?
Видимы:

```
func::x = 777
::v     = 333
```

И все.

```
::x                заслонен локальной переменной.
main::y,  main::z  невидимы, так как находятся
                   не на вершине стека вызовов функций
```

Многие функции более естественно выражаются через рекурсию.
Хотя, часто это приводит к излишним вычислениям по сравнению с итерациями
(то есть циклами). Вот пример - числа Фибоначчи.

```
int fibonacci(int n){
        if(n==1 || n==2) return 1;

        /* else */

        return fibonacci(n-1) + fibonacci(n-2);
}
void main(){
        printf("20ое число Фибоначчи равно %d\n", fibonacci(20));
}
```

Поскольку тут отсутствует массив для запоминания промежуточных
результатов, то этот массив на самом деле неявно моделируется
в виде локальных переменных внутри стека вызовов функций.
Однако этот способ плох - в нем слишком много повторяющихся

действий. Добавим оператор печати - и посчитаем, сколько раз была вызвана fibonacci(1) ?

```
int called = 0;

int fibonacci(int n){
        if(n==1){
                called++;
                return 1;

        } else  if(n==2)
                return 1;

        return fibonacci(n-1) + fibonacci(n-2);
}
void main(){
        printf("20ое число Фибоначчи равно %d\n", fibonacci(20));
        printf("fibonacci(1) была вызвана %d раз\n", called);
}
```

Она была вызвана... 2584 раза!

14.c

```
/* Рисуем хитрую геометрическую фигуру */
#include <stdio.h>

const int LINES = 15;

void draw(int nspaces, int nstars, char symbol){
        int i;

        for(i=0; i < nspaces; i++)
                putchar(' ');
        for(i=0; i < nstars; i++)
                putchar(symbol);
}

void main(){
        int nline, nsym;
        char symbols[3];        /* Массив из трех букв */

        symbols[0] = '\\';
        symbols[1] = 'o';
        symbols[2] = '*';

        for(nline=0; nline < LINES; nline++){
                for(nsym = 0; nsym < 3; nsym++)
                        draw(nline, nline, symbols[nsym]);

                /* Переход на новую строку вынесен
                   из функции в главный цикл */
                putchar('\n');
        }
}
```

15.c

```
/* Задача:
        нарисовать таблицу вида

        кот     кот     кот     кошка   кошка   кот
```

```
        кот     кот     кошка   кошка   кот     ...

   Где идет последовательность
        кот, кот, кот, кошка, кошка...
   повторяющаяся много раз и располагаемая по 6 зверей в строку.
*/

#include <stdio.h>      /* магическая строка */

/* Объявление глобальных переменных.
   В данном случае - констант.
 */

int TOMCATS     = 3;            /* три кота  */
int CATS        = 2;            /* две кошки */
int ANIMALS_PER_LINE    = 6;    /* шесть зверей в каждой строке */
int LINES       = 25;           /* число выводимых строк */

/* и нам понадобится еще одна переменная - общее число зверей.
   Ее мы вычислим через уже заданные, поэтому тут мы ее объявим...
   но вычислить что-либо можно только внутри функции.
   В нашем случае - в функции main().
*/
int ANIMALS;                    /* общее число зверей */

int nth_in_line = 0;            /* номер зверя в текущей строке */
/* Эта переменная не может быть локальной в функции, так как
 * тогда она уничтожится при выходе из функции. Нам же необходимо,
 * чтобы ее значение сохранялось. Поэтому переменная - глобальная.
 */

/* Функция, которая считает число зверей в одной строке
   и либо переводит строку, либо переводит печать в
   следующую колонку (табуляцией).
*/
void checkIfWeHaveToBreakLine(){
        nth_in_line++;  /* текущий номер зверя в строке (с единицы) */

        if(nth_in_line == ANIMALS_PER_LINE){
        /* Если строка заполнена зверьми... */
                putchar('\n');      /* новая строка */
                nth_in_line = 0;    /* в новой строке нет зверей */
        } else {
                putchar('\t');      /* в следующую колонку */
        }
}

void main(){
        int nanimal;    /* номер зверя */
        int i;          /* счетчик     */

        ANIMALS = ANIMALS_PER_LINE * LINES;
        nanimal = 0;

        while(nanimal < ANIMALS){

                for(i=0; i < TOMCATS; i++){
                        /* Оператор printf() выводит СТРОКУ СИМВОЛОВ.
                           СТРОКА заключается в двойные кавычки
                           (не путать с одиночными для putchar().
                         */
                        printf("кот");
```

```
                        nanimal++;      /* посчитать еще одного зверя */

                        /* и проверить - не надо ли перейти на новую строку ? */
                        checkIfWeHaveToBreakLine();
                }
                for(i=0; i < CATS; i++){
                        printf("кошка");
                        nanimal++;      /* посчитать еще одного зверя */

                        /* и проверить - не надо ли перейти на новую строку ? */
                        checkIfWeHaveToBreakLine();
                }
        }
        /* putchar('\n'); */
}
```

16.c

```
/*
        Та же задача, но еще надо печатать номер каждого зверя.
        Ограничимся пятью строками.
*/

#include <stdio.h>      /* магическая строка */

int TOMCATS     = 3;            /* три кота  */
int CATS        = 2;            /* две кошки */
int ANIMALS_PER_LINE    = 6;    /* шесть зверей в каждой строке */
int LINES       = 5;            /* число выводимых строк */
int ANIMALS;                    /* общее число зверей */

int nth_in_line = 0;            /* номер зверя в текущей строке */

void checkIfWeHaveToBreakLine(){
        nth_in_line++;

        if(nth_in_line == ANIMALS_PER_LINE){
                putchar('\n');
                nth_in_line = 0;
        } else
                printf("\t\t");         /* @ */

        /* Одинокий оператор может обойтись без {...} вокруг него */
}

void main(){
        int nanimal;
        int i;

        ANIMALS = ANIMALS_PER_LINE * LINES;
        nanimal = 0;

        while(nanimal < ANIMALS){

                for(i=0; i < TOMCATS; i++){
                /* Формат %d выводит значение переменной типа int
                   в виде текстовой строки.
                   Сама переменная должна быть в списке после формата
                   (список - это перечисление переменных через запятую).
                   Переменных ИЛИ выражений (формул).

                   Давайте выводить по ДВЕ табуляции --
```

```
                это место отмечено в функции checkIfWeHaveToBreakLine()
                как @.

                Еще раз внимание - один символ мы выводим как
                     putchar('a');
                Несколько символов - как
                     printf("abcdef");

                Одиночные кавычки - для одной буквы.
                Двойные кавычки   - для нескольких.
             */

                     printf("кот%d", nanimal);
                     nanimal++;

                     checkIfWeHaveToBreakLine();
             }
             for(i=0; i < CATS; i++){
                     printf("кошка%d", nanimal);
                     nanimal++;

                     checkIfWeHaveToBreakLine();
             }
        }
}
```

17.c

```
/* Задача: напечатать корни из чисел от 1 до 100.

   Новая информация:
        Нам понадобится новый тип данных - ДЕЙСТВИТЕЛЬНЫЕ ЧИСЛА.
        Это числа, имеющие дробную часть (после точки).
        Как мы уже знаем, целые - это   int.
                          буква - это   char.
                          действительное число - это double.
        (есть еще слово float, но им пользоваться не рекомендуется).

   Для вычисления корня используется итерационный алгоритм Герона.

             q = корень из x;

             q[0]   := x;
             q[n+1] := 1/2 * ( q[n] + x/q[n] );

   Главное тут не впасть в ошибку, не клюнуть на q[n] и не
   завести массив. Нам не нужны результаты каждой итерации,
   нам нужен только конечный ответ. Поэтому нам будет вполне
   достаточно ОДНОЙ, но изменяющейся в цикле, ячейки q.
*/

#include <stdio.h>

/* Еще одно новое ключевое слово - const. Обозначает константы.
   В отличие от переменных, такие имена нельзя изменять.
   То есть, если где-то потом попробовать написать epsilon = ... ;
   то это будет ошибкой.
 */
const double epsilon = 0.0000001;       /* точность вычислений */

/* Функция вычисления модуля числа */
double doubleabs(double x){
```

```
        if(x < 0) return -x;
        else      return x;
}

/* Функция вычисления квадратного корня */
double sqrt(double x){

        double sq = x;

        /* Такая конструкция есть просто склейка двух строк:
                double sq;
                sq = x;
           Называется это "объявление переменной с инициализацией".
         */

        while(doubleabs(sq*sq - x) >= epsilon){
                sq = 0.5 * (sq + x/sq);
        }
        return sq;
}

void main() {
        int n;

        for(n=1; n <= 100; n++)
                printf("sqrt(%d)=%lf\n",
                             n,  sqrt((double) n)
                     );

}

/*
   Здесь в операторе printf() мы печатаем ДВА выражения.
        ФОРМАТ          ЗНАЧЕНИЕ
        ------          --------
        %d      --      n
        %lf     --      sqrt((double) n)

        По формату %d печатаются значения типа int.
        По формату %c печатаются значения типа char.
        По формату %lf (или %g) печатаются значения типа double.

   Что значит "напечатать значение выражения sqrt(xxx)" ?
   Это значит:
        - вызвать функцию sqrt() с аргументом, равным xxx;
        - вычислить ее;
        - возвращенное ею значение напечатать по формату %lf,
          то есть как действительное число.

   Заметьте, что тут возвращаемое значение НЕ присваивается
   никакой переменной, мы не собираемся его хранить.

   Точно так же, как в операторе x = 12 + 34;
   12 и 34 не хранятся ни в каких переменных,
   а оператор

        printf("%d\n", 12);

   печатает ЧИСЛО 12, а не переменную.

   Точно так же, как можно писать

        double z;
```

```
        z = sqrt(12) + sqrt(23);
```

где значение, вычисленное каждой функцией, НЕ хранится
в своей собственной переменной (такая переменная на самом
деле существует в компьютере, но программисту она не
нужна и недоступна).
Или

```
        z = sqrt( sqrt(81));

        (корень из корня из 81 --> даст 3)
```

Далее, что означает конструкция (double) n ?
Функция sqrt() требует аргумента типа double.
Мы же предлагаем ей целый аргумент

```
        int n;
```

Целые и действительные числа представлены в памяти
машины ПО-РАЗНОМУ,
поэтому числа

```
        12   и   12.0  хранятся в памяти ПО-РАЗНОМУ.
```

Машина умеет преобразовывать целые числа в действительные
и наоборот, надо только сказать ей об этом.
Оператор (double) x
называется "приведение типа к double".

Заметим, что часто преобразование типа
выполняется автоматически.

Так, например, при сложении int и double
int автоматически приводится к double, и результат
имеет тип double.

```
        int    var1;
        double var2, var3;

        var1 = 2;
        var2 = 2.0;
        var3 = var1 + var2;
```

что означает на самом деле

```
        var3 = (double) var1 + var2;
```

var3 станет равно 4.0

Более того, к примеру тип char - это тоже ЦЕЛЫЕ ЧИСЛА из интервала
0...255. Каждая буква имеет код от 0 до 255.

```
*/
```

* 18_POINTERS.txt *

```
УКАЗАТЕЛИ
=========
void f(int x){
        x = 7;
}
```

```
main(){
        int y = 17;
        f(y);
        printf("y=%d\n", y);       /* печатает: y=17 */
}
```

В аргументе x переда?тся КОПИЯ значения y,
поэтому x=7; не изменяет значения y.
Как все же сделать, чтобы вызываемая функция
могла изменять значение переменной?

Отбросим два способа:

- объявление y как глобальной
  (много глобальных переменных - плохой стиль),

- y=f(y);
  (а что если надо изменить МНОГО переменных?
  return, к несчастью, может вернуть лишь одно значение).

Используется новая для нас конструкция: УКАЗАТЕЛЬ.

Пример (@)

```
void f(int *ptr){       /* #2 */
        *ptr = 7;       /* #3 */
}

main (){
        int y=17;

        f(&y);          /* #1 */
        printf("y=%d\n", y);       /* печатает: y=7 */
}
```

Ну как, нашли три отличия от исходного текста?

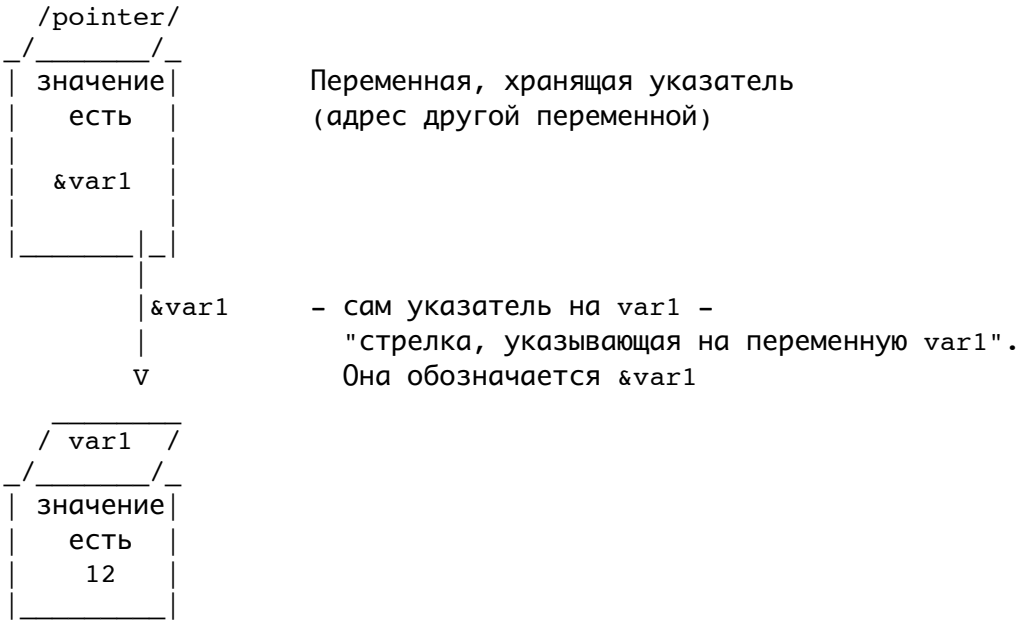
Мы вводим две новые конструкции:

```
        &y              "указатель на переменную y" или
                        "адрес переменной y"

        *ptr            означает "разыменование указателя ptr"
                        (подробнее - позже)

        int *ptr;       означает объявление переменной ptr,
                        которая может содержать в себе
                        указатель на переменную,
                        хранящую int-число.
```

Для начала определим, что такое указатель.

```
        int var1, var2, z;      /* целочисленные переменные */
        int *pointer;           /* указатель на целочисленную переменную */

        var1    = 12;
        var2    = 43;
        pointer = &var1;
```

Мы будем изображать указатель в виде СТРЕЛКИ;
это хороший прием и при практическом программировании.



Таким образом, УКАЗАТЕЛЬ - это "стрелка, указывающая на некий ящик-переменную".
Начало этой стрелки можно (в свою очередь) хранить в какой-нибудь переменной.

При этом, если стрелка указывает на переменную типа `int`,
то тип переменной, хранящей начало стрелки, есть `int *`

Если типа `char`, то тип - `char *`

АДРЕС (указатель на) можно взять только от переменной или элемента массива,
но не от выражения.

```
int x;
int arr[5];

Законно     &x          стрелка на ящик "x"
            & arr[3]    стрелка на ящик "arr[3]"

Незаконно   &(2+2)      тут нет именованного "ящика",
                        на который указывает стрелка,
                        да и вообще ящика нет.
```

ИСПОЛЬЗОВАНИЕ УКАЗАТЕЛЕЙ

Указатели несколько различно ведут себя СЛЕВА и СПРАВА
от оператора присваивания.
Нас интересует новая операция, применяемая только к указателям:

```
*pointer
```

```
---------------------------------------------------------------------
СПРАВА от присваиваний и в формулах
==================================
*pointer        означает
                "взять значение переменной (лежащее в ящике),
                на которую указывает указатель,
                хранящийся в переменной pointer".
```

В нашем примере - это число `12`.

То есть *pointer означает "пройти по стрелке и взять указываемое ею ЗНАЧЕНИЕ".

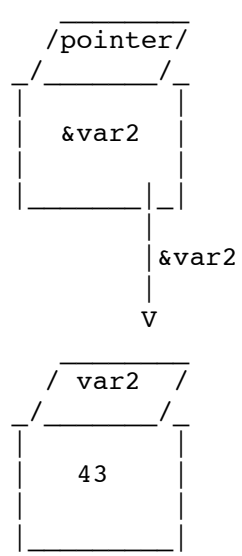
```
printf("%d\n", *pointer);
```

Печатает 12;

```
z = *pointer;        /* равноценно z = 12;        */
z = *pointer + 66;   /* равноценно z = 12 + 66;   */
```

Заставим теперь указатель указывать на другую переменную
(иначе говоря, "присвоим указателю адрес другой переменной")

```
pointer = &var2;
```



После этого z = *pointer;
означает   z = 43;

Таким образом, конструкция

```
z = *pointer;

    означает

z = *(&var2);

    означает

z = var2;
```

То есть * и & взаимно СТИРАЮТСЯ.

СЛЕВА от присваивания...

```
*pointer = 123;
```

Означает "положить значение правой части (т.е. 123)
в переменную (ящик), на который указывает указатель,
хранящийся в переменной pointer".

Пройти по стрелке и положить значение в указываемую переменную.

В данном случае *pointer обозначает
не ЗНАЧЕНИЕ указываемой переменной,
а САМУ указываемую переменную.


```
pointer  = &var2;
*pointer = 123;
```

означает

```
*(&var2) = 123;
```

означает

```
var2 = 123;
```

То есть снова * и & взаимно СТИРАЮТ друг друга.

Ещ? пример:

```
*pointer = *pointer + 66;
```

или

```
*pointer += 66;
```

Вернемся к примеру с функцией (@). Как он работает?

В строке /* #1 */
Мы вызываем функцию f(), передавая в нее
УКАЗАТЕЛЬ на переменную y ("адрес переменной y").

В строке /* #2 */
Отводится локальная переменная с именем ptr,
которая в качестве начального значения
получает значение первого аргумента функции в точке вызова -
то есть УКАЗАТЕЛЬ на y.

В строке /* #3 */
Мы видим

```
*ptr = 7;
```

что следует рассматривать как

```
*(&y) = 7;          точнее *(&main::y)=7;
```

то есть как

```
y = 7;              точнее main::y=7;
```

Что и хотелось.

При этом отметим, что само имя "y" этой переменной
внутри функции f() НЕВИДИМО и НЕИЗВЕСТНО!

ПРИМЕР: обмен значений двух переменных.

```
void main(){
        int x, y;
        int temporary;  /* вспомогательная переменная */

        x=1; y=2;

        temporary=x; x=y; y=temporary;
        printf("x=%d y=%d\n", x, y);    /* Печатает x=2 y=1 */
}
```

Теперь то же с использованием адресов и указателей:

```
void swap(int *a, int *b){
        int tmp;

        tmp = *a; *a = *b; *b = tmp;
}

void main(){
        int x, y;

        x = 1; y = 2;
        swap(&x, &y);
        printf("x=%d y=%d\n", x, y);
}
```

Ещ? пример:

```
        int x;
        int *ptr1, *ptr2;

        ptr1 = &x; ptr2 = &x;
        *ptr1 = 77;
        printf("%d\n", *ptr2);          /* Печатает 77 */
```

То есть на одну переменную МОГУТ указывать несколько указателей.

Ещ? пример:

```
        int x;
        int *ptr1;              /* Не инициализирована */

        x = *ptr1;
```

В ptr1 нет указателя ни на что, там есть мусор.
Указатель указывает "в никуда" (пальцем в небо).
Скорее всего произойд?т сбой в работе программы.

Мораль: ВСЕГДА инициализируй переменные, указатели в том числе.

МАССИВЫ

Язык Си работает с именами массивов специальным образом.
Имя массива "a" для

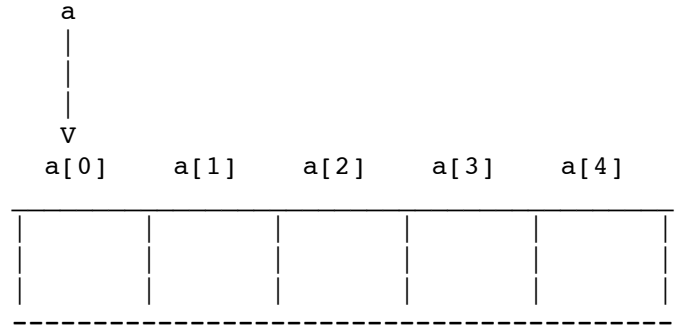
```
int a[5];
```

является на самом деле указателем на его нулевой элемент.

То есть у нас есть переменные (ящики)
с именами

```
a[0]    a[1]    ...     a[4].
```

При этом само имя a при его использовании в программе означает &a[0]



Поэтому

```
int a[5];

/* Переда?тся не КОПИЯ самого массива, а копия УКАЗАТЕЛЯ на его начало */

void f(int *a){         /* или f(int a[]), что есть равноценная запись */
        printf("%d\n", a[1]);
        a[2] = 7;
}

main (){
        a[1] = 777;
        f(a);           /* аргумент - массив */
        printf("%d\n", a[2]);
}
```

Вызов f(a); сделает именно ожидаемые вещи.
В этом примере мы видим два правила:

ПРАВИЛО_1:
    При передаче в функцию имени массива
    в аргумент функции копируется не весь массив (жирновато будет),
    а указатель на его 0-ой элемент.

ПРАВИЛО_2:
    Указатель на начало массива
    МОЖНО индексировать как сам массив.
    Это вторая операция, помимо *pointer,
    применимая к указателям: pointer[n].

Второе правило влечет за собой ряд следствий.

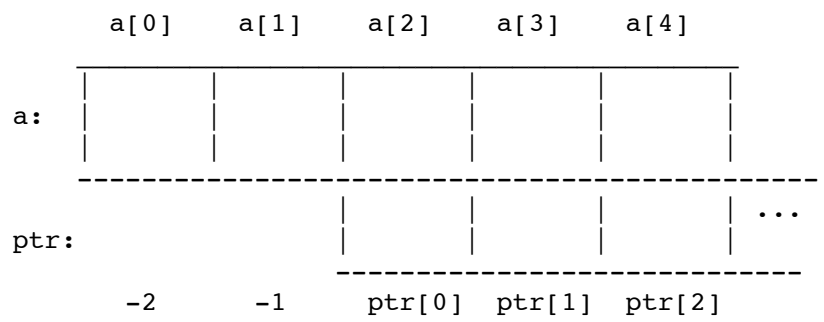
```
        int a[5];       /* массив */
        int *ptr;       /* указательная переменная */

        ptr = a;        /* законно, означает ptr = &a[0]; */
```

Теперь

```
        ptr[0] = 3;     /* означает a[0] = 3;   */
        ptr[1] = 5;     /* означает a[1] = 5;   */
```

Более того. Возьмем теперь

```
        ptr = &a[2];
```



Мы как бы "приложили" к массиву a[] массив ptr[].

В котором

```
        ptr[0]  есть    a[2]
        ptr[1]  есть    a[3]
        ptr[2]  есть    a[4]
        ptr[3]          находится за концом массива a[], МУСОР
```

Более того, допустимы отрицательные индексы!

```
        ptr[-1] есть    a[1]
        ptr[-2] есть    a[0]
        ptr[-3]         находится перед началом массива a[], МУСОР
```

Итак: индексировать можно И массивы И указатели.

Кстати, для имени массива a[]
        *a означает то же самое, что и a[0].

Это обратное следствие из схожести массивов и указателей.

19.c

```
/* Задача: написать функцию инвертирования порядка символов
   в массиве char.
        A B C D ---> D C B A
   В решении можно использовать рекурсию.
 */

/* Мы приведем рекурсивное и нерекурсивное решения (два варианта) */
#include <stdio.h>

/* Сначала - несколько служебных функций. */
```

```
/* ФУНКЦИЯ ПОДСЧЕТА ДЛИНЫ СТРОКИ.
   Как уже объяснялось, строка текста - это массив char,
   в конце которого помещен символ '\0'.
   Сам символ \0 не считается.
 */
int strlen(char s[]){                   /* функция от массива букв            */
        int counter = 0;                /* счетчик и одновременно индекс      */

        while(s[counter] != '\0')       /* пока не встретился признак конца текста */
                counter++;              /* посчитать символ                   */
        return counter;                 /* сколько символов, отличных от '\0' */
}

/* ФУНКЦИЯ ПЕЧАТИ СТРОКИ.
   Печатаем каждый элемент массива как символ при помощи putchar(c).
   Как только встречаем элемент массива, равный '\0' - останавливаемся.
   Заметьте, что при наличии завершающего символа нам НЕ НАДО
   передавать в функцию размер массива, он нам неинтересен.

   В конце эта функция переводит строку.
*/
int putstr(char s[]){
        int i = 0;      /* индекс */

        while(s[i] != '\0'){
                putchar(s[i]);
                i++;
        }
        putchar('\n');
        return i;
}

/* ТЕПЕРЬ МЫ ЗАНИМАЕМСЯ ФУНКЦИЕЙ ИНВЕРТИРОВАНИЯ.
   Для этого нам нужна вспомогательная функция:
   сдвиг элементов массива на 1 влево.

        Исходный массив: A B C D E F
                         <----------
        Результат:       B C D E F F
                                   -
        Последний элемент удваивается.

        n - размер массива.
        Функция работает так:

        Исходный массив: A B C D E F  n=6
        После i=1        B B C D E F
        После i=2        B C C D E F
        После i=3        B C D D E F
        После i=4        B C D E E F
        После i=5        B C D E F F
        i=6 ==> остановка.

*/
void shiftLeft(char s[], int n){
        int i;

        for(i=1; i < n; i++)
                s[i-1] = s[i];
}

/* Функция инвертирования.
```

```
    Идея такова:
        - если длина массива меньше или равна 1, то инвертировать нечего,
          ибо массив состоит из 1 или 0 элементов.
        - если длина массива > 1, то
          a) Спасти нулевой элемент массива.
                A B C D E F
                |
                |
                V
               tmp

          b) Сдвинуть массив влево
                B C D E F F

          c) В последний элемент массива поместить спасенный нулевой элемент.
                          tmp
                           |
                           V
                B C D E F A


          d) Инвертировать начало массива длиной n-1.
               {B C D E F}A

             Получится:
                F E D C B A

             Что и требовалось.

        s[] - массив,
        n   - его длина.
*/
void reverse(char s[], int n){
        char tmp;

        if(n <= 1)              /* нечего инвертировать */
                return;

        tmp = s[0];             /* спасти */
        shiftLeft(s, n);        /* сдвинуть */
        s[n-1] = tmp;           /* переместить */

        reverse(s, n-1);        /* инвертировать начало */
}

/* ВТОРАЯ ВЕРСИЯ нерекурсивна. Рекурсия заменена циклом.
   Длина начала массива, которую надо инвертировать,
   вынесена на переменную length.
*/
void reverse1(char s[], int n){
        char tmp;
        int length;

        for(length=n; length > 1; --length){
                tmp = s[0];
                shiftLeft(s, length);
                s[length-1] = tmp;
        }
}

char testString[] = "abcdefghijklmnopqrstuvwxyz";

/* Если не задать размер массива, он будет вычислен компилятором автоматически.
   Он будет равен числу букв внутри "..." ПЛЮС одна ячейка для невидимого
```

```
   символа '\0' на конце.
   В данном случае это 27.
*/

void main(){
        int len;

        len = strlen(testString);
        /* вычислить длину строки: 26 ('\0' на конце не считается) */

        printf("Строка для теста: \"%s\", ее длина %d\n",
                                  testString,    len);
        /* Обратите внимание на два момента:
                - строку (массив char) следует печатать по формату %s
                - чтобы внутри "..." напечатать символ "
                  надо изобразить его как \"

           А чтобы в putchar напечатать символ ' надо писать putchar('\'');
         */

        /* Первая инверсия */
        reverse(testString, len);
        putstr("Инвертированная строка:");
        putstr(testString);

        /* Вторая инверсия - возвращает в исходное состояние */
        reverse1(testString, len);
        putstr("Инвертированная в исходное состояние строка:");
        putstr(testString);
}
```

19_1.c

```
/* Еще более простой вариант решения:
   просто обменивать элементы местами.

        A B C D E F G H I J
        J B C D E F G H I A
        |                 |     эти
        J B C D E F G H I A
        J I C D E F G H B A
          |             |       потом эти
        J I C D E F G H B A
        J I H D E F G C B A
            |         |         потом эти
          ---->     <-----

        J I H D E F G C B A
        J I H G E F D C B A
              |     |
        J I H G E F D C B A
                | |
        J I H G F E D C B A

                стоп.
*/

#include <stdio.h>

/* Обмен значений двух переменных типа char */
void swap(char *s1, char *s2){
        char c;
```

```
        c = *s1; *s1 = *s2; *s2 = c;
}

void reverse(char s[], int n){
        int first, last;

        first = 0;              /* индекс первого элемента массива */
        last = n-1;             /* индекс последнего элемента массива */

        while(first < last){    /* пока first левее last */
                swap(&s[first], &s[last]);
                first++;        /* сдвинуть вправо */
                last--;         /* сдвинуть влево  */
        }
}

char testString[] = "abcdefghijklmnopqrstuvwxyz.";

void main(){
        int len;

        len = strlen(testString);       /* Есть такая стандартная функция */
        reverse(testString, len);
        printf("Инвертированная строка: %s\n", testString);
}
```

19_2.c

```
/* Еще один вариант решения:
   сформировать ответ в дополнительном массиве,
   а потом скопировать его на прежнее место.
*/

#include <stdio.h>

char testString[] = "abcdefghijklmnopqrstuvwxyz.";

/* Конструкция sizeof(массив)/sizeof(массив[0])
   выдает размер массива, даже если он не был явно объявлен.
   Эта конструкция применяется (чаще всего) для задания массива
   с размером, равным размеру уже объявленного массива.
 */

char tempString[ sizeof(testString) / sizeof(testString[0]) ];

void reverse(char s[], int n){
        int i;

        /* вывернуть, результат в tempString[] */
        for(i=0; i < n; i++)
                tempString[n-1-i] = s[i];

        tempString[n] = '\0';   /* признак конца строки */

        /* скопировать на старое место */
        for(i=0; i < n; i++)
                s[i] = tempString[i];

        s[n] = '\0';            /* признак конца строки */
}

void main(){
        int len;
```

```
        len = strlen(testString);       /* Есть такая стандартная функция */
        reverse(testString, len);
        printf("Инвертированная строка: %s\n", testString);
}
```

19_3.c

```
/* Задача инвертирования массива целых чисел */

#include <stdio.h>

int arr[]  = {1, 5, 10, 15, 20, 25, 30};
int arrLen = sizeof(arr) / sizeof(arr[0]);   /* размер массива */

/* Распечатка массива в строку */
void printit(int row[], int n){
        int i;

        for(i=0; i < n; i++){
                printf("%d", row[i]);

                if(i == n-1) putchar('\n');
                else         putchar(' ');
        }
}
/* Печать отступа. Отладочная функция */
void printShift(int n){
        n = arrLen - n;
        while(n > 0){
                printf("   ");
                n--;
        }
}
/* Сдвиг массива */
void shiftleft(int row[], int n){
        int i;

        for(i=1; i < n; i++)
                row[i-1] = row[i];
}
/* Инвертирование */
void reverse(int  row[], int n){
        int pocket;

        printShift(n);                          /* трассировка */
        printf("CALLED reverse(row, %d)\n", n); /* трассировка */

        if(n <= 1){
                printShift(n);                  /* трассировка */
                printf("return from reverse(row, %d);\n", n); /* трассировка */
                return;
        }

        pocket = row[0];
        shiftleft(row, n);
        row[n-1] = pocket;

        printShift(n);                          /* трассировка */
        printit(arr, arrLen);                   /* трассировка */

        reverse(row, n-1);
```

```
        printShift(n);                          /* трассировка */
        printf("all done; return from reverse(row, %d);\n", n); /* трассировка */
}
void main(){
        reverse(arr, arrLen);
        printit(arr, arrLen);
}
```

20.c

```
/* Задача: написать функцию для распечатки массива целых чисел
   в виде таблицы в columns столбцов.
   При этом порядок элементов должен быть таков:

   0    4    8
   1    5    9
   2    6    10
   3    7
*/

/* Пусть в массиве n элементов.
   Если n < columns, то мы получаем такую ситуацию (n=2, columns=4)

        0       1       пусто       пусто

   Поэтому

        if(n < columns) columns = n;

   Далее, прямоугольная таблица с columns столбцами и lines строками
   может содержать максимум columns*lines элементов. Поэтому:

        columns*lines >= n

   Вычислим число строк.
   Нам нужно минимальное целое число строк, такое что

        lines >= n/columns

   Такое число вычисляется по формуле

        lines = (n + (columns - 1)) / columns;

   где деление целочисленное.

   Далее надо только вывести формулу для индекса элемента в массиве
   в зависимости от номера строки (y) и столбца (x).

   index(x, y) = (x * lines + y);
                 причем если index >= n, то ничего не выводить
*/

#include <stdio.h>

int array[100];

void printArray(int a[], int n, int columns){
        int lines;      /* число строк */

        int x, y;       /* номер колонки, номер строки - с нуля */
        int index;      /* индекс элемента в массиве */
```

```
        if(n < columns) columns = n;
        lines = (n + (columns-1)) / columns;

        /* Используем вложенные циклы: по строкам, а внутри - по столбцам */
        for(y=0; y < lines; y++){
                for(x=0; x < columns; x++){
                        index = x * lines + y;
                        if(index >= n)  /* элемент за концом массива */
                                break;  /* прервать строку */

                        /* break выводит только
                           из внутреннего цикла (по столбцам) */

                        /* сделать отступ в следующую колонку */
                        if(x != 0) putchar('\t');

                        printf("%02d|%d", index, a[index]);
                        /* Формат %02d заставляет печатать целое число
                           с использованием ДВУХ цифр, причем если
                           число состоит из одной цифры, то спереди
                           приписывается нуль.
                         */
                }
                putchar('\n');  /* перейти к следующей строке */
        }
}
void main(){
        int i, cols;

        /* Инициализация значений элементов массива */
        for(i=0; i < 100; i++)
                array[i] = i + 1;

        for(cols=4; cols <= 13; cols++){
                printf("\t\t* * * ТАБЛИЦА В %d СТОЛБЦОВ * * *\n", cols);
                printArray(array, 77, cols);
                putchar('\n');
        }
}
```

20_1.c

```
#include <stdio.h>

main(){
        int x, y;
        int COLUMNS = 11;
        int LINES = 10;

        int value;

        /* цикл по строкам */
        for(y=0; y < LINES; y++){

                /* цикл по столбцам */
                for(x=0; x < COLUMNS; x++){

                        /* что напечатать */
                        value = LINES * x + y;

                        /* если это не нулевой столбец, то перейти
                           в следующую колонку
```

```
                         */

                        if(x > 0) putchar('\t');

                        /* ... и в ней напечатать значение */
                        printf("%d", value);

                }
                putchar('\n');  /* новая строка */
        }
}
```

20_2.c

```
/*
        elem(x, y) = LINES * x + y;

                тогда

        elem(0, y+1) - elem(COLUMNS-1, y) = 1 + LINES - COLUMNS*LINES;
        elem(x+1, y) - elem(x, y)         = LINES;
*/

#include <stdio.h>

int A = 150;            /* Количество элементов      */
int COLUMNS = 7;        /* Количество столбцов       */
int LINES;              /* Количество строк          */
int value;              /* Значение в текущей клетке */

int OFFSET_NEXT_COLUMN;
int OFFSET_NEXT_LINE;

/* Рисуем строку таблицы */
void line(){
        int col;        /* номер колонки */

        for(col=0; col < COLUMNS; col++){
                if(value >= A)       /* отсутствующий элемент */
                        printf("* ");
                else    printf("%03d ", value);

                /* Увеличение при переходе в соседнюю колонку */
                value += OFFSET_NEXT_COLUMN;    /* 1 */
        }
        /* Перейти к следующей строке */
        putchar('\n');

        /* Увеличение при переходе из конца одной строки к началу следующей.
           Заметим, что к value уже прибавлено OFFSET_NEXT_COLUMN из точки 1,
           поэтому при переходе в начало следующей строки в сумме прибавляется
           OFFSET_NEXT_COLUMN + OFFSET_NEXT_LINE равное
                1 - LINES*COLUMNS + LINES,
           что соответствует формуле.
         */
        value += OFFSET_NEXT_LINE;              /* 2 */
}

int main(){
        int nline;  /* Номер строки */

        LINES = (A + (COLUMNS - 1)) / COLUMNS;
```

```
        OFFSET_NEXT_COLUMN = LINES;
        OFFSET_NEXT_LINE = 1 - LINES*COLUMNS;

        for(nline=0; nline < LINES; nline++)
                line();

        /* возврат 0 из main() означает "программа завершена успешно" */
        return 0;
}
```

21.c

```
/* ДВУМЕРНЫЕ МАССИВЫ */

/*
        Двумерный массив представляет собой двумерную
        прямоугольную таблицу из нумерованных переменных.

        Он объявляется так:

                int array[LINES][COLUMNS];

        А индексируется так:

                    array[y][x]

                где 0 <= y <= LINES - 1
                    0 <= x <= COLUMNS - 1

        +-------------+-------------+-------------+------> ось x
        | array[0][0] | array[0][1] | array[0][2] | ...
        +-------------+-------------+-------------+
        | array[1][0] | array[1][1] | array[1][2] | ...
        +-------------+-------------+-------------+
        | array[2][0] | array[2][1] | array[2][2] | ...
        +-------------+-------------+-------------+
        | ...           ...           ...
        V
       ось y

Пока, на данной стадии знания Си,
я рекомендую вам объявлять двумерные массивы как глобальные
и не пытаться передавать их имена в функции как аргументы.
*/

/* Приведем пример, который заводит двумерный массив букв,
   рисует в нем некую геометрическую фигуру,
   и печатает этот массив.

   Здесь мы приводим алгоритм Брезенхема для рисования прямых,
   объяснения КАК он это делает мы опустим. Пардон.
*/

#define LINES   31      /* число строк    */
#define COLUMNS 79      /* число столбцов */

char field[LINES][COLUMNS];

/* В данной программе массив НЕ является параметром,
   мы работаем с ним как с глобальной переменной.
   Функция рисования прямой линии, алгоритм Брезенхема.
 */
```

```
void line(int x1, int y1, int x2, int y2, char sym){
        int dx, dy, i1, i2, i, kx, ky;
        int d;      /* "отклонение" */
        int x, y;
        int flag;

        dy = y2 - y1;
        dx = x2 - x1;
        if (dx == 0 && dy == 0){
                field[y1][x1] = sym;    /* единственная точка */
                return;
        }
        kx = 1; /* шаг по x */
        ky = 1; /* шаг по y */

        /* Выбор тактовой оси */
        if( dx < 0 ){ dx = -dx; kx = -1; } /* Y */
        else if(dx == 0)        kx = 0;    /* X */

        if(dy < 0) { dy = -dy; ky = -1; }

        if(dx < dy){ flag = 0; d = dx; dx = dy; dy = d; }
        else         flag = 1;

        i1 = dy + dy; d = i1 - dx; i2 = d - dx;
        x = x1; y = y1;

        for(i=0; i < dx; i++){
                field[y][x] = sym;      /* нарисовать точку */

                if(flag) x += kx; /* шаг по такт. оси   */
                else     y += ky;

                if( d < 0 ) /* горизонтальный шаг      */
                         d += i1;
                else{       /* диагональный шаг        */
                         d += i2;
                         if(flag) y += ky; /* прирост высоты */
                         else     x += kx;
                }
        }
        field[y][x] = sym; /* последняя точка  */
}

int main(){
        int x, y;

        /* Заполнить поле пробелами */
        for(y=0; y < LINES; y++)
                for(x=0; x < COLUMNS; x++)
                        field[y][x] = ' ';

        /* Нарисовать картинку */
        line(0,0,               0,         LINES-1, '*');
        line(0,0,               COLUMNS-1, 0,       '*');
        line(COLUMNS-1, 0,      COLUMNS-1, LINES-1, '*');
        line(0, LINES-1,        COLUMNS-1, LINES-1, '*');

        line(0,0,               COLUMNS-1, LINES-1, '\\');
        line(COLUMNS-1,0,       0,LINES-1,          '/');

        /* Распечатать массив */
        for(y=0; y < LINES; y++){
                for(x=0; x < COLUMNS; x++)
                        putchar(field[y][x]);
```

```
                putchar('\n');
        }
        return 0;
}
```

Популярность: **21**, Last-modified: Wed, 05 Mar 2008 22:36:19 GMT