

Строки Многомерные массивы

Урок 8 Программирование на языке Си (базовый уровень)





План курса

1

Введение в язык Си

2

Типы данных
Операторы и выражения

3

Ветвления и побитовые
операции

4

Циклы

5

Буферный ввод-вывод
Функции

6

Рекурсия
Вещественные типы
данных

7

Указатели
Массивы

8

Строки
Многомерные массивы

9

Структурные типы данных
Файлы

10

Многомодульные
программы

11

Аргументы командной строки
Препроцессор
Отладка программ



На этой лекции вы узнаете:

- Что такое строки?
- Преобразование строки в массив байт
- Функции библиотек `stdlib.h` и `string.h`
- Массивы указателей, многомерные и VLA
- Сортировка `qsort` и чтение деклараторов



Строки



Что такое строки?

Строка в языке Си — это массив из символов типа **char**, в конце которого стоит символ с кодом 0 (**'\0'**).

Все программы и библиотечные функции, работающие со строками, основаны на том, что в конце строки обязательно есть символ 0.

Особенности строк:

- Все строки состоят из символов с кодом отличным от 0;
- Длина строки не содержится в самой строке;
- Нет никаких ограничений на длину строки.



Примеры объявления строк

```
char s[10]; // Строка из 9 значимых символов  
char s[N]; // Строка из N-1 значимых символов, в строке всегда есть  
символ \0
```

```
char st[] = "hello"; // st[0]='h' st[1]='e' st[2]='l' st[3]='l'  
st[4]='o' st[5]='\0'  
printf("%u", sizeof(st)); // 6
```

```
char st[10] = "hello"; // st[0]='h' st[1]='e' st[2]='l' st[3]='l'  
st[4]='o' st[5]='\0'  
printf("%u", sizeof(st)); // 10
```



Ввод и вывод строк

Для ввода и вывод строк можно также использовать функции `printf` и `scanf`, указав соответствующие спецификаторы в параметре форматной строки.

```
char s[10];  
scanf("%s",s); // считать строку до первого пробельного символа или \n  
printf("%s",s); // напечатать строку
```

 **Внимание!** При выводе строки учитывается символ `'\0'`



Ввод строки с ограничением длины


Функция `scanf` последовательно считывает в строку `s` символы вводимой строки и добавляет в конец строки символ `'\0'`.

Ввод выполняется до первого символа пробел, символа перевода на следующую строку и т.п.

Необходимо позаботиться о размере массива, куда считывается строка, сама функция никак не контролирует это.

Можно ограничить размер считываемой строки.

```
char s[10];  
scanf("%9s", s); // считать строку не более 9 символов
```

 **Внимание!** Если строка больше, чем выделенное под нее место, то произойдет аварийное завершение программы.



Пример посимвольного ввода и вывода строки

- Считываем строку до первого символа “перенос строки” и заносим все в массив. Обратите внимание на скобки внутри while.
- Необходимо поставить признак конца строки после считывания.
- Напечатать строку можно также с помощью функции посимвольного вывода putchar().

```
char s[10], c;  
int i=0;  
while( (c=getchar())!='\n' && i<9 )  
    s[i++]=c;  
s[i]='\0';
```

```
i=0;  
while( s[i] ) // s[i] != 0  
    putchar(s[i++]); // s + i++;
```



Множество сканирования scanf



Шаблон `scanf` считываемой строки

Функция `scanf` позволяет использовать некоторое подобие регулярного выражения для задания шаблона считываемой строки. В форматной строке можно задать допустимые символы — множество сканирования. Можно определить символы, которые будут считываться и присваиваться элементам соответствующего символьного массива. Например:

```
char s[100];  
scanf("%[a-z]", s); // считать стр буквы  
printf("%s\n", s);
```

```
helloWORLD  
  
hello
```

```
scanf("%[0-9]", s); // считать цифры  
printf("%s\n", s);
```

```
123WORLD  
  
123
```



Пример шаблонов `scanf` считываемой строки

<pre>scanf("%[^\\n]", s); // все кроме \\n printf("%s\\n", s);</pre>	<pre>Hello world Hello world</pre>
<pre>char s1[100], s2[100]; scanf("%s%s", s1, s2); printf("s1 = %s s2 = %s\\n", s1, s2);</pre>	<pre>Hello world s1 = Hello s2 = world</pre>
<pre>char s1[100], s2[100]; scanf("%[0-9]=%[a-z]", s1, s2); printf("s1 = %s s2 = %s\\n", s1, s2);</pre>	<pre>123=hello s1 = 123 s2 = hello</pre>

В данном примере ожидается строка начинающаяся с цифр, потом идет символ '=', а затем маленькие английские буквы.



Что будет напечатано?

```
char s[100];  
scanf("%5s",s);//чтение 5 символов  
printf("%s\n",s);
```

Hello! 79 world
?

```
int i;  
scanf("%s%d",s, &i);// чтение строки и  
целого  
printf("%s %d\n",s,i);
```

Hello 79 world
?



**Поставьте видео
на паузу и
решите задачу**



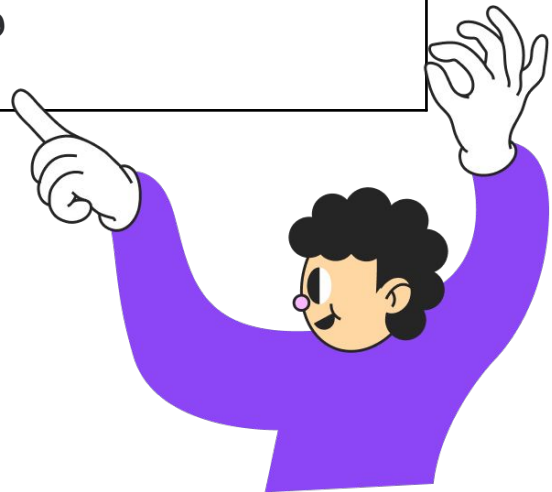
Ответ: Что будет напечатано

```
char s[100];  
scanf("%5s",s);//чтение 5 символов  
printf("%s\n",s);
```

```
Hello! 79 world  
  
Hello
```

```
int i;  
scanf("%s%d",s, &i);// чтение строки и  
целого  
printf("%s %d\n",s,i);
```

```
Hello 79 world  
  
Hello 79
```





Проверка работы функции **scanf**

Функция `scanf` возвращает количество успешно распознанных спецификаторов в качестве результата или `-1` если на вход передан конец строки или произошла ошибка.

```
char s1[100], s2[100];  
int r;  
r=scanf("%[0-9]=%[a-z]",s1,s2);  
printf("r = %d\n",r);
```

```
123=hello  
r = 2
```

```
r=scanf("%[0-9]=%[a-z]",s1,s2);  
printf("r = %d\n",r);
```

```
123=123  
r = 1
```

```
r=scanf("%[0-9]=%[a-z]",s1,s2);  
printf("r = %d\n",r);
```

```
hello=123  
r = 0
```



Задачи на строки



Определение длины строки через `Sizeof` и `strlen`

Для работы со строками существует стандартная библиотека `<string.h>`. Функция `int strlen(const char *src)` - возвращает длину строки.

```
#include <stdio.h>
#include <string.h>

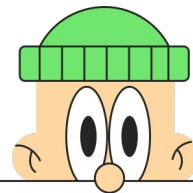
int main(void) {
    char st[10] = "hello";
    printf("Sizeof = %u\n", sizeof(st));
    printf("Strlen = %u\n", strlen(st));
    return 0;
}
```

```
Sizeof = 10
Strlen = 5
```



Что будет напечатано?

```
char st1[10] = "hello";  
char st2[10] = "hello";  
if(st1 == st2)  
    printf("Yes");  
else  
    printf("No");
```



**Поставьте видео
на паузу и
решите задачу**




Как не нужно сравнивать строки

```
char st1[10] = "hello";  
char st2[10] = "hello";  
if(st1 == st2)  
    printf("Yes");  
else  
    printf("No");
```

No

Для работы со строками существует стандартная библиотека `<string.h>`.

Функция `int strcmp(const char *a, const char *b)` сравнивает строки.

 **Внимание!** Для сравнения строк нельзя использовать операции `==`, `!=` и т.п., поскольку при этом происходит сравнение указателей на начало соответствующих строк, а не самих строк.



Нахождение длины строки

Реализовать строковую функцию `int strlen(const char *src)`

```
#include <stdio.h>

int strlen(const char *src)
{
    int len=0;
    while (*src++) len++;
    return len;
}

int main(int argc, char **argv)
{
    char* str={"Hello!"};
    printf("%d\n",strlen(str));
    return 0;
}
```



Копирования одной строки в другую строку

Копирования строки `src` (включая `'\0'`) в строку `dst` . Функция возвращает указатель на первый символ строки `dst`.

```
#include <stdio.h>

char *strcpy (char *dst, char *src)
{
    char *ptr = dst;
    while(*dst++=*src++);
    return ptr;
}

int main(int argc, char **argv)
{
    char str1[]={"Hello!"}; //char* str1 = {"Hello!"};
    char str2[]={"World!"}; //char* str2={"World!"}
    printf("%s\n",strcpy(str2,str1));
    printf("%s\n",str2);
    return 0;
}
```



Сравнение строк

Реализовать строковую функцию `int strcmp(const char *a, const char *b)`

Функция сравнивает в лексикографическом порядке строку cs со строкой ct.

- Если строка cs меньше строки ct, возвращается значение < 0 ,
- если строка cs больше строки ct, возвращается значение > 0 ,
- в случае равенства строк возвращается значение 0.

```
#include <stdio.h>

int strcmp(const char *a, const char *b){
    while ( *a && *b && *a == *b )
        ++a, ++b;
    return *a - *b;
}
```



Пример сравнения строк

```
void Print(char* str,int res)
{
    printf(str, res==0 ? "equal to" :  res<0 ? "less" : "greater than");
}

int main(void){
    char *a = "abcde",*b = "xyz",*c = "abcd",*d = "xyz";
    printf("A = %s\nB = %s\nC = %s\nD = %s\n\n", a, b, c, d);
    Print("A is %s B\n",strcmp(a,b));
    Print("A is %s D\n",strcmp(a,b));
    Print("B is %s C\n",strcmp(a,b));
    Print("B is %s D\n",strcmp(a,b));
    Print("C is %s D\n",strcmp(a,b));
    return 0;
}
```



Количество слов в тексте

Посчитать количество слов в тексте, слова разделены одним или несколькими пробелами.

```
#include <stdio.h>
int main(void) {
    char s[100];
    int count=0;
    while (scanf("%s",s)==1)
    {
        count++;
        printf("In this text %d words\n",count);
    }
    return 0;
}
```

```
hello      world.      I
love      peace

In this text 5 words
```




Сумма цифр в строке

Реализовать функцию, которая возвращает сумму цифр в переданной ей строке

```
#include <stdio.h>

int str_sum_digits(const char*s) {
    int i = 0;
    int sum=0;
    while (s[i++]!='\0')
        if(s[i]>'0' && s[i]<='9')
            sum+=s[i]-'0';
    return sum;
}
```

```
int main(void) {
    char s[100];
    scanf("%s",s);
    printf("str_sum_digits=
%d\n",str_sum_digits(s));
    return 0;
}
```



Преобразование строки в массив байт



Преобразование строки в массив вспомогательные функции

Реализовать функцию, которая преобразует переданную строку в массив байт, возвращает количество байт

```
int StrToHexMas(char* Str,uint8_t* Hex)
```

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>

//преобразование hex-цифры в дес-число
int CharToHex(char c)
{
    int result=-1;
    if (c>='0' && c<='9')
        result=c-'0';
    else if(c>='A' && c<='F')
        result=c-'A'+10;
    else if(c>='a' && c<='f')
        result=c-'a'+10;
    return result;
}
```

```
int StrToHexMas(char* Str,uint8_t* mas);

int main(int argc, char **argv)
{
    uint8_t arr[10];
    int len = StrToHexMas("AAa a 1 15",arr);
    printf("%s\n","AAa a 1 15");
    printf("%d\n",len);
    for(int i=0;i<len;i++)
        printf("%02x,",arr[i]);
    return 0;
}
```



Преобразование строки в массив

```
//данные идут последовательно, не более двух
символов
int StrToHexMas(char* Str,uint8_t* mas)
{
    int Result = 0; //полученное число
    int data    = 0; //временная переменная
    int i       = 0; //счетчик символов по
строке
    int index   = 0; //счетчик данных в массиве
    int StrLenght = strlen(Str);
    printf("%d\n",StrLenght);
    while(i<StrLenght)//выполняем цикл, пока
есть символы в строке
    {
        Result=0;    //обнуляем число
        data = CharToHex(Str[i++]);
        //анализируем очередной символ
        if(data>=0)  //если это значащий символ
        {
```

```
Result = data;
        if(i<StrLenght)//проверка на
выход за границы массива
        {
            data =
CharToHex(Str[i++]);//анализируем
очередной символ
            if(data>=0) //если это данные
            {
                Result *= 16;
                Result += data;
            }
            mas[index++]=Result; //кладем
число в массив
        }
    }
    return index;
}
```



Библиотека `string.h`



Работа с памятью

Имя	Примечания
<u>memcpy</u>	копирует n байт из области памяти src в dest, которые не должны пересекаться
<u>memmove</u>	копирует n байт из области памяти src в dest, которые в отличие от memcpy могут перекрываться
<u>memchr</u>	возвращает указатель на первое вхождение значения символа
<u>memcmp</u>	сравнивает первые n символов в областях памяти
<u>memset</u>	заполняет область памяти указанным байтом



Заголовочный файл string.h работа со строками

Имя	Примечания
<u>*strcpy</u>	копирует строку из одного места в другое
<u>*strncpy</u>	копирует до n байт строки из одного места в другое
<u>strlen</u>	возвращает длину строки
<u>*strpbrk</u>	находит первое вхождение любого символа
<u>*strstr</u>	находит первое вхождение строки



Заголовочный файл string.h работа со строками

Имя	Примечания
<u>strcat</u>	дописывает строку src в конец dest
<u>strncat</u>	дописывает не более n начальных символов строки
<u>*strchr</u>	возвращает адрес символа в строке
<u>*strrchr</u>	возвращает адрес символа, начиная с хвоста
<u>strcmp</u>	лексикографическое сравнение строк
<u>strncmp</u>	лексикографическое сравнение первых n байтов строк



Пример функций `strcat`, `strcpy`, `strncpy` библиотеки `string.h`

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    char dst[30] = "Hello ";
    char src[30] = "GB!!!";
    strcat(dst, src);
    printf("%s\n", dst);    //Hello GB!!!
    strcpy(src, dst);
    printf("%s\n", src);    //Hello GB!!!
    int n = strlen(src); // количество копируемых символов
    strncpy(dst, src, n-1);

    //функция strncpy НЕ заканчивает скопированную строку нулевым символом,
    //что может привести к переполнению буфера.
    //Поэтому после копирования следует вручную устанавливать нулевой символ
```



Пример функций `strstr` библиотеки `string.h`

```
dst[n-1] = '\\0';
printf("%s %d\\n", dst,n);    //Hello GB!! 11
char substr[14] = "GB!";
char *substr_ptr = strstr(dst, substr);
// если подстрока найдена
if(substr_ptr)
{
    // вычисляем позицию подстроки в строке
    long pos = substr_ptr - dst;
    printf("Substring index: %ld\\n", pos); // Substring index: 6
}
else    // если подстрока не найдена
{
    printf("Substring not found\\n");
}
return 0;
}
```



Библиотека `stdlib.h`



Заголовочный файл `stdlib.h` в соответствии со стандартами Си

Имя	Преобразование типов	C89	C99
<u>atof</u>	строка в число двойной точности (double; НЕ float)	Да	Да
<u>atoi</u>	строка в целое число (integer)	Да	Да
<u>atol</u>	строка в длинное целое число (long integer)	Да	Да
<u>atoll</u>	строка в длинное целое число (long long integer)	Нет	Да
<u>strtod</u>	строка в число двойной точности (double)	Да	Да
<u>strtof</u>	строка в число одинарной точности (float)	Нет	Да
<u>strtol</u>	строка в длинное целое число (long integer)	Да	Да
<u>strtold</u>	строка в длинное число двойной точности (long double)	Нет	Да
<u>strtoll</u>	строка в длинное целое число (long long integer)	Нет	Да
<u>strtoul</u>	строка в беззнаковое длинное целое число (unsigned long integer)	Да	Да
<u>strtoull</u>	строка в беззнаковое длинное целое число (unsigned long long integer)	Нет	Да



Функции atoi, atof

```
#include <stdio.h>    //Для printf()
#include <stdlib.h>    //Для atoi()
int main (void)
{
    char *Str = "652.2345brtt 7";
    //Строка для преобразования
    char *ptr=Str;
    //Преобразование строки в число типа
    int
    int Num = atoi (Str);
    //Вывод результата преобразования
    printf ("%d\n",Num);
```

```
float Num1 = atof (Str);
    //Вывод результата
    преобразования
    printf ("%f\n",Num1);
    ptr+=13;
    int Num3 = atoi (ptr);
    //Вывод результата
    преобразования
    printf ("%d\n",Num3);
    return 0;
}
```



Заголовочный файл `stdlib.h` функции

Имя	Генерация псевдослучайных последовательностей	C89	C99
<code>rand</code>	генерирует псевдослучайное значение	Да	Да
<code>srand</code>	устанавливает начальное значение генератора псевдослучайных чисел	Да	Да
	Выделение и освобождение памяти		
<code>malloc</code> <code>calloc</code> <code>realloc</code>	выделяет память из кучи	Да	Да
<code>free</code>	освобождает память обратно в кучу	Да	Да
	Сортировка и поиск		
<code>qsort</code>	сортировка массива	Да	Да
	Многобайтовые операции/ широкие символы	Нет	Да
	Контроль процесса выполнения программы	Да	Да
	Математика		Да



Массивы указателей, многомерные и VLA



Массивы указателей

Массив указателей — это массив, в котором каждый элемент ссылается на объект одного типа.

Обычно массивы указателей ссылаются на строки: каждый элемент массива содержит адрес начального (нулевого) символа строки.

Очевидное преимущество такого способа хранения — это разная длина хранящихся строк.

```
const uint8_t *ps[] = {"one", "two", "three", NULL};  
// NULL признак конца  
for(uint32_t i=0; ps[i] ;i++)  
    printf("%s\n", ps[i]);
```




VLA массивы

В C99 можно описывать локальные массивы неконстантного размера (Variable-length array) и выделять для них динамическую память.

Без использования указателей VLA массивы могут быть использованы как локальные переменные, либо параметры функций.


```
uint32_t n;  
scanf("%u",&n); //вводим количество элементов  
int32_t ar[n]; //создаем VLA массив
```



Особенности VLA массивов

```
// Делайте так  
void func(int n, int arr[n])
```

```
// ТАК НЕЛЬЗЯ - ОШИБКА  
void func(int arr[n], int n)
```

 **Внимание!** Если массив используется в качестве параметра функции и размерность массива передаются в эту функцию, то указатель должен следовать после описания размера.



Многомерные массивы

В языке Си можно определять многомерные массивы.

Объявляются многомерные массив в общем случае так:

```
тип имя [индекс_1] [индекс_2] ... [индекс_n];
```



Пример двумерного массива

По сути, двумерный массив является фактически одномерным массивом, каждый элемент которого в свою очередь также является одномерным массивом. Двумерным он называется потому, что управляется при помощи двух индексов (индекс строки — первый при объявлении и индекс столбца).

В данном примере массив `matr` — это массив состоящий из 3 объектов, каждый из которых в свою очередь состоит из 5 элементов.

```
int matr[3][5]; // 3 строки и 5 столбцов
```

<code>matr[0][0]</code>	<code>matr[0][1]</code>	<code>matr[0][2]</code>	<code>matr[0][3]</code>	<code>matr[0][4]</code>
<code>matr[1][0]</code>	<code>matr[1][1]</code>	<code>matr[1][2]</code>	<code>matr[1][3]</code>	<code>matr[1][4]</code>
<code>matr[2][0]</code>	<code>matr[2][1]</code>	<code>matr[2][2]</code>	<code>matr[2][3]</code>	<code>matr[2][4]</code>



Пример объявления многомерного массива

Оба варианта инициализации массива эквивалентны, поскольку инициализируются все элементы многомерного массива.

Если же инициализируются не все элементы, то внутренние фигурные скобки обязательны

```
int m[4][3] = {{11, 15, 30},  
               {10, 20, 31},  
               { 5,  8,  0},  
               {17, 25, 47}};
```

```
int m[4][3] = {11, 15, 30,  
               10, 20, 31,  
               5,  8,  0,  
               17, 25, 47};
```



Пример частичной инициализации многомерного массива

```
int m[4][3] = {{11, 15, 30},  
               {10, -2},  
               {5},  
               {}};
```

При записи выше первый внутренний массив проинициализируется полностью, во втором остается в неопределенном состоянии последний элемент, в третьем подмассиве - два последних элемента останутся без инициализации, хотя память под них будет выделена, как и под четвертый подмассив.



Пример частичной инициализации многомерного массива

В такой записи:

```
int m[4][3] = {11, 15, 30,  
               10, -2,  
               5};
```

компилятор также не найдёт ошибок, хотя результат будет совсем иным, чем в предыдущей записи, а именно: пятерка проинициализирует третий элемент второго подмассива, а третий и четвертый подмассивы останутся без инициализации



Обработка массивов

Как правило для обработки массивов используются вложенные циклы. Причем для обработки двумерных массивов — два вложенных цикла, для трехмерных — три цикла и т.д.

```
int matr[3][5];  
int i,j;  
for(i=0;i<3;i++)  
for(j=0;j<5;j++)  
    matr[i][j]=i+j;
```

```
int matr[3][2] = {  
    {1,2},    //строка 0  
    {10,-5}, //строка 1  
    {0,-17}  //строка 2  
};
```




Многомерные массивы и указатели

Можно объявить указатель на двумерный массив, однако при этом теряется информация о размере строки.

```
int matr[3][2];  
int *pm;  
pm = matr;  
pm[1][1] = 123; //ТАК НЕЛЬЗЯ
```

```
int matr[3][2];  
int *pm;  
pm = matr;  
*(pm + 3) = 123; // matr[1][1]  
pm[3] = 123; // или так  
printf("%d\n",matr[1][1]);
```



Обработка массивов

А можно объявить указатель на строку матрицы

```
int matr[3][2];  
int (*pm)[2]; //указатель на строку из 2-ух int  
pm = matr;  
pm[1][1] = 123; //теперь все ок  
printf("%d\n",matr[1][1]); // 123
```



Вопрос: Что будет напечатано?

```
int m[3][3] =  
{  
    {1, 2, 3},  
    {4, 5, 6},  
    {7, 8, 9}  
};  
int *p1;  
int (*p2)[3];  
p1 = m[1];  
p2 = m + 1;  
p1++;  
p2++;  
printf ("%d %d\n", *p1, **p2);
```





Ответ: Что будет напечатано?

```
int m[3][3] =
```

```
{
```

```
    {1, 2, 3},
```

```
    {4, 5, 6},
```

```
    {7, 8, 9}
```

```
};
```

```
int *p1;
```

```
int (*p2)[3];
```

```
p1 = m[1];
```

```
p2 = m + 1;
```

```
p1++;
```

```
p2++;
```

```
printf ("%d %d\n", *p1, **p2);
```

5 7

m[i]

m [i] [j]

m	m[0]	1	2	3
	m[1]	4	5	6
	m[2]	7	8	9



Печать матрицы

Написать функцию, которая выводит на печать матрицу

```
void print_matrix_1 (int m, int n, int
*a)
{
    int i, j;
    for (i = 0; i < m; i++){
        for (j = 0; j < n; j++){
            printf ("%d ", a[i*n + j]);
        }
        printf ("\n");
    }
}
```

```
void print_matrix_2 (int m, int n, int
a[m][n])
{
    int i, j;
    for (i = 0; i < m; i++){
        for (j = 0; j < n; j++){
            printf ("%d ", a[i][j]);
        }
        printf ("\n");
    }
}
```



След матрицы

Написать функцию, которая вычисляет след матрицы.

След матрицы - это сумма элементов на главной диагонали.

```
int trace_matrix(int m, int n, int a[m][n])
{
    int sum=0;
    for (int i = 0; i < m; ++i)
        for (int j = 0; j < n; ++j)
            if (i == j)
                sum += a[i][j];
    return sum;
}

...
printf("%d\n", trace_matrix(3,3,m));
```



Сортировка qsort



Сортировка **qsort**

```
qsort(base, num , size , (int(*) (const void *, const void *)) comparator)
```

Функция **qsort** сортирует массив, на который указывает параметр *base*, используя quicksort — алгоритм сортировки Хоара. Параметр *num* задает число элементов массива, параметр *size* задает размер в байтах каждого элемента.

Функция, на которую указывает параметр *compare*, сравнивает элементы массива с ключом. Формат функции *compare* следующий:

```
int func_name(const void *arg1, const void *arg2)
```

Она должна возвращать следующие значения:

- Если *arg1* меньше, чем *arg2*, то возвращается отрицательное целое.
- Если *arg1* равно *arg2*, то возвращается 0.
- Если *arg1* больше, чем *arg2*, то возвращается положительное целое.



Сортировка массива. Функция сравнения **comparator**

Массив сортируется по возрастанию таким образом, что наименьший адрес соответствует наименьшему элементу.

```
/* сравнение двух целых V1 */
#include <stdio.h>
#include <stdlib.h>
int comparator (const int *a, const int *b) {
    return *a - *b;
}
int main(void)
{
    int a[3] = {3,1,2};
    qsort(a, 3, sizeof (int), (int(*) (const void *, const void *))
comparator);
    printf("%d,%d,%d",a[0],a[1],a[2]);
}
```



Объявление типа `comparator_type` функции `comparator`

Добавим typedef в версию V2

```
#include <stdio.h>
#include <stdlib.h>
typedef int(*comparator_type) (const void *, const void *);
/* сравнение двух целых V2 */
int comparator (const int *a, const int *b) {
    return *a - *b;
}
int main(void)
{
    int a[3] = {3,1,2};
    qsort(a, 3, sizeof (int), (comparator_type) comparator);
    printf("%d,%d,%d",a[0],a[1],a[2]);
}
```



Приоритеты операций с указателями



Что будет напечатано?

```
int a = 10;  
int *pa = &a;  
int b = *pa + 20;    // операция со  
                      // значением, на который указывает  
                      // указатель  
pa++;                // операция с  
                      // самим указателем  
printf("b=%d \n", b);
```

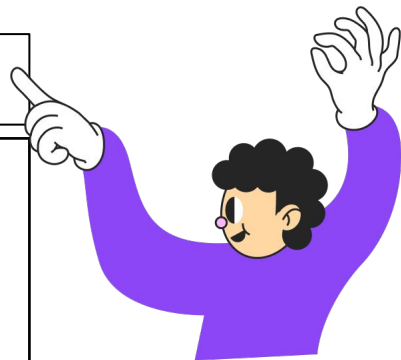


Особенности арифметических операций с указателями

При работе с указателями надо отличать операции с самим указателем и операции со значением по адресу, на который указывает указатель.

b=30

```
int a = 10;
int *pa = &a;
int b = *pa + 20;    // операция со значением, на который
                     // указывает указатель
pa++;                // операция с самим указателем
printf("b=%d \n", b); // 30
```



То есть в данном случае через операцию разыменования `*pa` получаем значение, на которое указывает указатель `pa`, то есть число 10 и выполняем операцию сложения, то есть в данном случае обычная операция сложения между двумя числами, так как выражение `*pa` представляет число.



Особенности, указателей с операциями инкремента и декремента

Операции *, ++ и -- имеют одинаковый приоритет и при размещении рядом выполняются справа налево.

```
int a = 10;
int *pa = &a;
printf("pa: address=%p \t value=%d \n", (void*)pa, *pa);
int b = *pa++;          // инкремент адреса указателя

printf("b: value=%d \n", b);
printf("pa: address=%p \t value=%d \n", (void*)pa, *pa);
```

```
pa: address=0x7ffc3ee33bd8    value=10
b: value=10
pa: address=0x7ffc3ee33bdc    value=10
```



Приоритет арифметических операций с указателями

В выражении `b = *pa++;` сначала к указателю присваивается единица (то есть к адресу добавляется 4, так как указатель типа `int`).

Изменим выражение:

`b = (*pa)++;`

Здесь сначала выполняется операция разыменования и получение значения, затем это значение увеличивается на 1. Теперь по адресу в указателе находится число 11.



Особенности, указателей с операциями инкремента и декремента

Операции *, ++ и -- имеют одинаковый приоритет и при размещении рядом выполняются справа налево.

```
int a = 10;
int *pa = &a;
printf("pa: address=%p \t value=%d \n", (void*)pa, *pa);
int b = (*pa)++;      // инкремент адреса указателя

printf("b: value=%d \n", b);
printf("pa: address=%p \t value=%d \n", (void*)pa, *pa);
```

```
pa: address=0x7ffdefba9e08    value=10
b: value=10
pa: address=0x7ffdefba9e08    value=11
```




Деклараторы



Что такое декларатор?

Декларатор — это синтаксическая конструкция состоящая из комбинации модификаторов типа, таких как указатели, массивы, функции.

<базовый тип> <декларатор> [= <инициализатор>] ;

Декларатор обычно содержит имя определяемого объекта, но в общем случае может не содержать имя определяемого объекта.

Анонимные деклараторы допускаются в операции приведения типа и при описании формальных параметров в прототипах функций.

char (*(*x[3])())[10]; — на самом деле это значит объявить x как массив 3 указателей на функцию, возвращающую указатель на массив из 10 char.



Операции деклараторов

Декларатор можно рассматривать как некоторое выражение над типом.

В таком выражении есть три операции:

[] постфиксная — массив из заданного количества элементов

() постфиксная — функция с заданными параметрами

* префиксная — указатель

() группировка членов в выражении



Приоритеты операций деклараторов

Постфиксные операции имеют самый высокий приоритет и читаются слева направо от определяемого имени. Это `[]` — массив из заданного количества элементов и `()` — функция с заданными параметрами

Префиксная операция имеет более низкий приоритет и читается справа налево. Это `*` — указатель

Скобки могут использоваться для изменения порядка чтения.

Декларатор читается, начиная от имени определяемого объекта следуя правилам приоритетов операций.

Имя определяемого объекта — это первое имя после базового типа.

<базовый тип> **<декларатор>** `[= <инициализатор>] ;`



Пример чтения декларатора

```
char * ( * (*var) () ) [10];  
  ^   ^  ^  ^  ^   ^   ^  
  7   6  4  2  1   3   5
```

В этом примере шаги пронумерованы по порядку и интерпретируются следующим образом:

1. Идентификатор var объявлен как
2. указатель на
3. функцию, возвращающую
4. указатель на
5. массив из 10 элементов, которые являются
6. указателями на
7. значения типа char

Задача: что означает данный декларатор

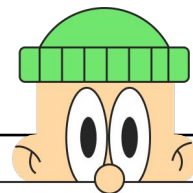
```
int a[3][4];
```

```
char **b;
```

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

```
int *f();
```

```
int (*f());
```



**Поставьте видео
на паузу и
решите задачу**

Ответ: что означает данный декларатор

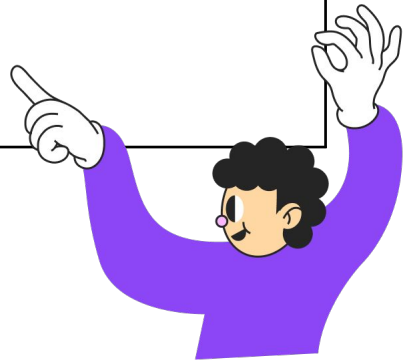
`int a[3][4];` - массив из 3 элементов типа массива из 4 элементов `int`

`char **b;` - указатель на указатель на `char`

`int (*a)[10];` - указатель на массив из 10 элементов типа `int`

`int *f();` - функция, возвращающая указатель на `int`

`int (*f)();` - `f` это указатель на функцию, возвращающую значение `int`





Задача: что означает данный декларатор

`char *c[];` - массив из неопределённого количества элементов типа указатель на тип `char`

`int *d[10];` - массив из 10 элементов типа указатель на тип `int`


`int (*a)[5];` - указатель на массив из 5 элементов типа `int`

`int *(*f)();` - указатель на функцию, возвращающую указатель на `int`



На этой лекции вы узнали:

- Что такое строки?
- Преобразование строки в массив байт
- Функции библиотек `stdlib.h` и `string.h`
- Массивы указателей, многомерные и VLA
- Сортировка `qsort` и чтение деклараторов

Спасибо 
за внимание

Когда мы делаем дело, нет времени на
беспокойство! Когда мы беспокоимся, нет
времени на дело!

