

# Указатели и масиви

Пламен Танов

Ненко Табаков

Технологично училище „Електронни системи“

Технически университет – София

версия 0.1

# Въведение

- Указателите са променливи, които съдържат адреса (в паметта) на други променливи
- Указателите са широко използвани в C програмите – в дадени случаи употребата им е единственият начин за решаване на задача
- Използването им води до по-ефективен и компактен код
- **ANSI C** дефинира ясни правила, как да се използват указатели. Невнимателната им употреба често води до създаване и ползване на указатели, които съдържат неочаквани адреси

# Организация на паметта

Паметта в компютъра се разделя на клетки, които са последователно подредени и всяка от тях има адрес. Клетките в паметта може да се използват индивидуално (например **char**) или в групи (**double**).

Указател е група от клетки, които съдържат адрес на клетка в паметта.



# Дефиниране на указател<sub>1</sub>

- Указатели се дефинират по следния начин:

**тип \* име\_на\_указателя;**

- Унарнен оператор **&** връща адреса на променливата
- Унарнен оператор **\*** връща стойността на променливата, към която сочи указателя

```
int x = 1, y = 2, z[10];  
int *ip;    /* ip е указател към int*/  
ip = &x;    /* ip сочи към x*/  
y = *ip;    /* y приема стойност 1*/  
*ip = 0;    /* x приема стойност 0*/  
ip = &z[0]; /* ip сочи към z[0]*/
```

# Дефиниране на указател<sub>2</sub>

- Адрес може да се вземе само на обект в паметта (променлива, елемент на масив), но не и на израз или **register** променлива
- Всеки указател сочи към определен тип данни (този който е указан при декларирането им).  
Изключение правят указателите към тип **void**

```
register int x = 1;  
int *p;  
char c = '3';
```

```
p = &x;           //ГРЕШКА - променливата е register  
p = &(x+2*3);    //ГРЕШКА - само на обекти в паметта  
p = &c;          //ГРЕШКА - друг тип
```

# Дефиниране на указател<sub>3</sub>

- Указателят е променлива като всяка друга

```
int x = 1, z = 6;  
int *p1, *p2 = &z; // p2 сочи към z
```

```
p1 = p2; // p1 сочи където и p2, т.е. към z  
++*p1; // z = z + 1 = 7;
```

```
int x = 4, *px = &x, z = 6, *pz = &z;
```

≡

```
int x = 4, z = 6;  
int *px = &x, *pz;
```

```
pz = &z;
```

# Приоритет

```
( ) [ ] -> .  
! ~ ++ -- + - * (type) sizeof  
* / %  
+ -  
<< >>  
< <= > >=  
== !=  
&  
&  
|  
&&  
||  
?:  
= += -= *= /= %= &= ^= |= <<= >>=  
,
```

Унарните **&**, **+**, **-** и **\*** (пример: **&k**) имат по-висок приоритет от еквивалентните им бинарни форми (пример: **a&b**).

# Приоритети

- \* и & имат висок приоритет

```
int x = 4, *px = &x, z = 6, *pz = &z;  
*px = *px + 1; // x = x + 1 = 5  
z = *pz + 3; // z = z + 3 = 9
```

```
*pz += 5; // z = z + 5 = 14
```

```
++*px; // x = x + 1 = 6
```

```
(*px)++; // x = x + 1 = 7
```

```
*px++; // първо се изпълнява px++, след което *px
```

```
*++px;
```



# Указатели аргументи на функция

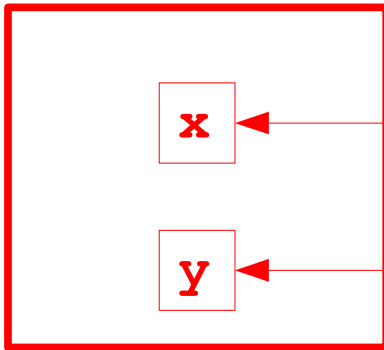
- В C аргументите на една функция се предават по стойност, затова няма директен начин една функция да промени стойността на някой от аргументите си.

```
void swap (int *a, int *b) {
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
int main () {
    int x = 2, y = 4;
    swap (&x, &y)
    printf("x=%d, y=%d", x, y);
    return 0;           //x=4, y=2
}

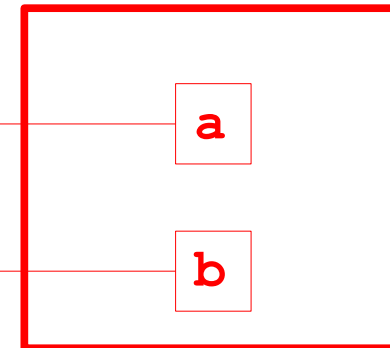
//ГРЕШНО !!!
void swap (int a, int b) {
    int temp;
    temp = a;
    a = b;
    b = temp;
}
int main () {
    int x = 2, y = 4;
    swap (x, y)
    printf("x=%d, y=%d", x, y);
    return 0;           //x=2, y=4
}
```

# Пример

```
int main ():
```



```
void swap (int *a, int *b):
```

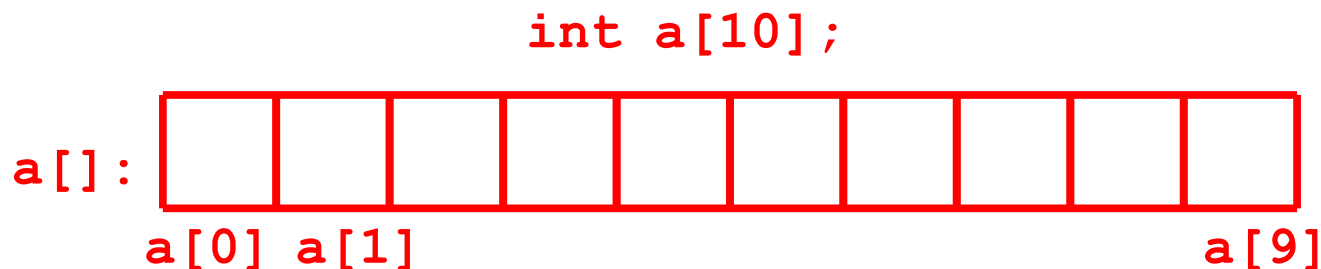


# Пример

```
/* getint: get next integer from input into *pn */
int getint(int *pn) {
    int c, sign;
    while (isspace(c = getch())) /* skip white space */
        ;
    if (!isdigit(c) && c != EOF && c != '+' && c != '-') {
        ungetch(c); /* it is not a number */
        return 0;
    }
    sign = (c == '-') ? -1 : 1;
    if (c == '+' || c == '-')
        c = getch();
    for (*pn = 0; isdigit(c), c = getch())
        *pn = 10 * *pn + (c - '0');
    *pn *= sign;
    if (c != EOF)
        ungetch(c);
    return c;
}
```

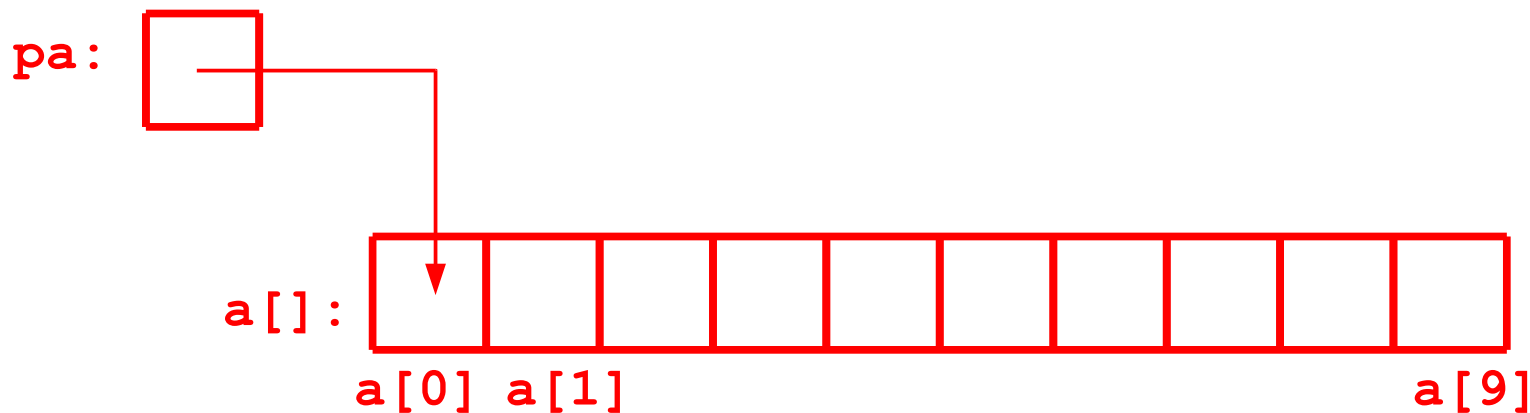
# Указатели и масиви<sub>1</sub>

- В С има силна връзка между указатели и масиви. Всяка операция, която може да бъде извършена с масиви, може да се направи и с указатели (към елементите на масива)
- Правило е, че версията с указатели работи по-бързо, но е по-неразбираема при четене от човек



# Пример

```
int a[10];  
int *pa;  
int x;  
pa = &a[0]; /* pa сочи към първия елемент от масива*/  
x = *pa; /* x приема стойността на първия елемент от масива*/  
x = pa[0]; /* еквивалентно на горния ред (x = a[0])*/
```



# Указатели и масиви<sub>2</sub>

Елементите на един масив могат да се достигат чрез индексите му или чрез адресна аритметика. По дефиниция името на масива е указател към първия елемент от него.

```
int a[10];  
int *pa;  
int x;
```



```
int a[10];  
int *pa;  
int x;
```



```
int a[10];  
int *pa;  
int x;
```

```
pa = &a[0];  
x = a[5];
```

```
pa = a;  
x = *(a+5);
```

```
pa = a;  
x = pa[5];
```

# Пример

```
int a[10];  
int *pa;  
int x;
```

```
x = *(pa + 1);  
i = 5;  
x = *(pa + i);
```

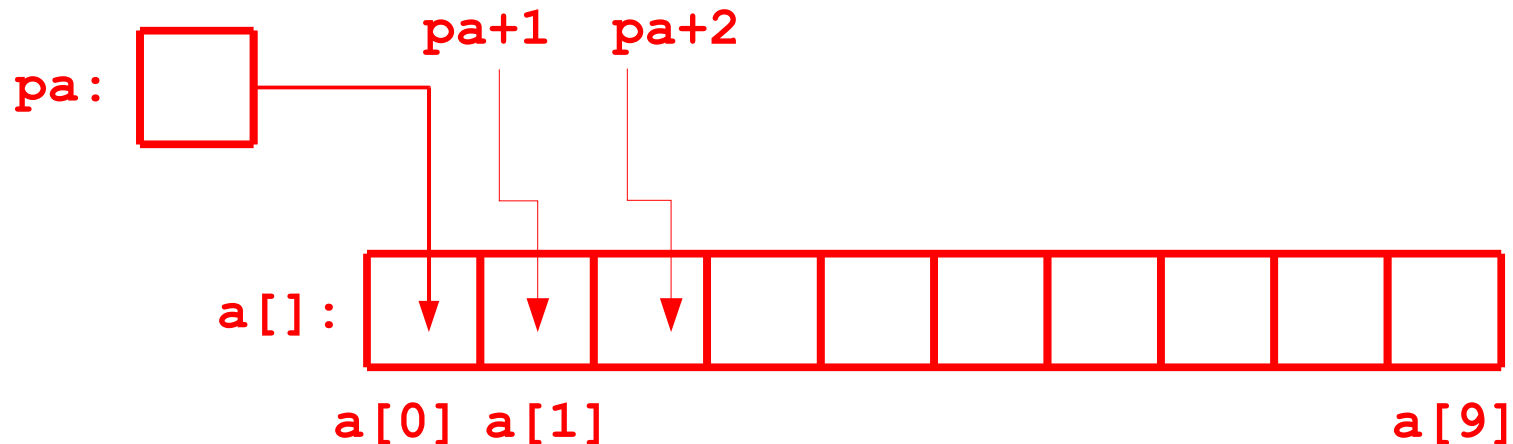
≡

```
int a[10];  
int *pa;  
int x;
```

```
x = a[1];  
i = 5;  
x = a[i]
```

```
for (i = 0; i < 10; i++)  
    printf ("%d", *(pa+i));
```

```
for (i = 0; i < 10; i++)  
    printf ("%d", a[i]);
```



# Указатели и масиви<sup>3</sup>

- `a[i]` е еквивалентно на `*(a+i)`
- `&a[i]` е еквивалентно на `(a + i)`
- Важна разлика между името на масив и указател е, че указателят е променлива и изразът:

```
int a[10];  
int *pa = a;
```

```
pa++; //указателят сочи към a[1] (следващия елемент)  
a++; //ГРЕШКА: това е масив, a винаги сочи към нулевия елемент
```

```
for (i = 0; i<10; i++)  
    printf ("%d", *(a+i));
```



# Пример

```
/* strlen: return length of string s */
int strlen(char *s) {
    int n;
    for (n = 0; *s != '\0', s++)
        n++;
    return n;
}

int main () {
    char hw[] = "hello world";
    printf ("%d", strlen(hw));
}
```

# Адресна аритметика

Ако **p** е указател към елемент от масив, то **p++** увеличава (инкрементира) стойността на **p** с 1, т.е. той вече сочи към следващия елемент от масива. По същия начин **p+=i** увеличава стойността на **p** с **i** и **p** ще сочи към **i**<sup>тия</sup> елемент спрямо този, който е сочил преди прибавянето.

Тези и подобни конструкции се наричат адресна аритметика.

```
int main () {
    int a[10];
    int *pa;
    int i = 5;
    pa = a; /* pa сочи към първия (a[0]) елемент от масива*/
    pa++; /* pa сочи към втория (a[1]) елемент от масива*/
    pa+=i; /* pa сочи към седмия (a[6]) елемент от масива*/
}
```

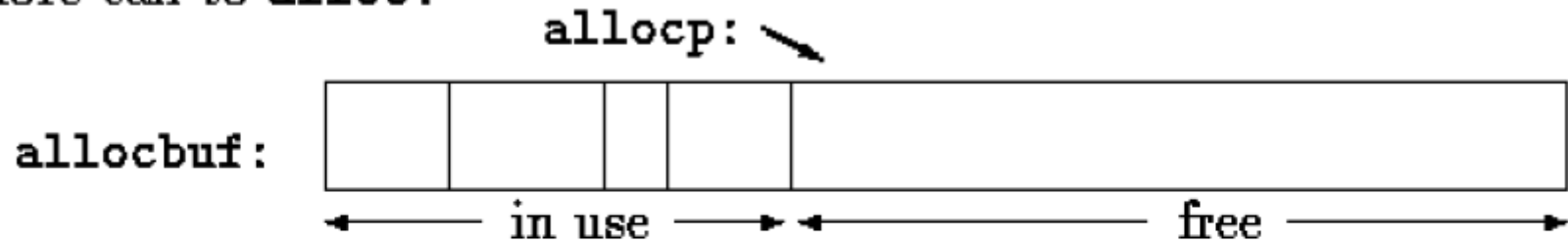
# Пример

```
/* strlen: return length of string s */
int strlen(char *s) {
    char *p = s;
    while (*p != '\0')
        p++;
    return p - s;
}

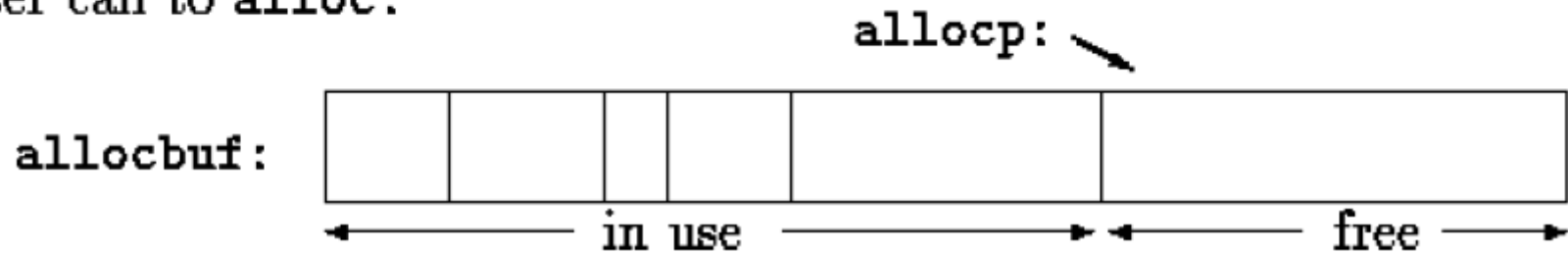
int main () {
    char hw[] = "hello world";
    printf ("%d", strlen(hw));
}
```

# Пример<sub>1</sub>

before call to alloc:



after call to alloc:



# Пример<sub>2</sub>

```
#define ALLOCSIZE 10000 /* size of available space */
static char allocbuf[ALLOCSIZE]; /* storage for alloc */
static char *allocp = allocbuf; /* next free position */

char *alloc(int n) { //return pointer to n characters
    if (allocbuf + ALLOCSIZE - allocp >= n) { /* it fits */
        allocp += n;
        return allocp - n; /* old p */
    } else /* not enough room */
        return 0;
}

void afree(char *p) { //free storage pointed to by p
    if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
        allocp = p;
}
```

# Символни указатели и функции<sub>1</sub>

- `"I am a string"` е константа, чийто край се определя със знака `'\0'`
- Когато един стринг, който е константа, се предава като аргумент на функция, достъпът до елементите му става чрез символен указател (указател към **char**)

```
int strlen(char *s) {
    char *p = s;
    while (*p != '\0')
        p++;
    return p - s;
}
int main () {
    printf ("%d", strlen("I am a string"));
}
```

# Символни указатели и функции<sub>2</sub>

```
char *pa;  
pa="I am a string"; /*pa сочи към стринг, който е константа*/
```

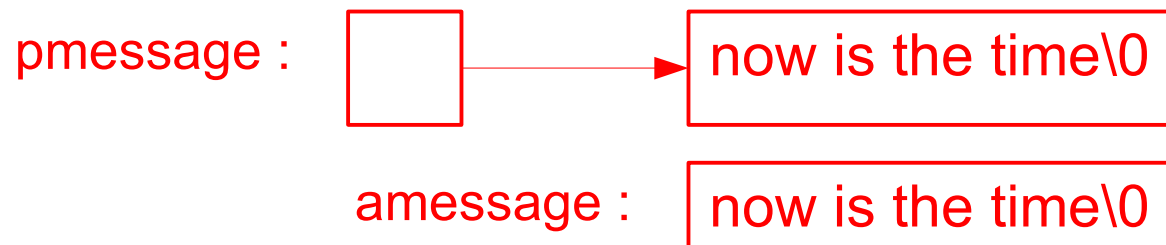
В този случай указателят **pa** само сочи към стринга. Това не е копие на символния низ. Следващите две дефиниции не са еквивалентни:

```
char amessage[] = "now is the time";// масив  
char *pmessage = "now is the time";//указател към константа  
  
amessage[1] = 'Z';//променяме елемент на масив  
pmessage[1] = 'Z';/* ГРЕШКА – опит за промяна на константа  
Компиляторът не съобщава за нея, но при изпълнение  
последниците са непредвидими! */  
pmessage = amessage;
```

# Символни указатели и функции<sub>3</sub>

**amessage** е масив, достатъчно голям да съдържа последователността от символи, които завършват с терминаращия символ '`\0`'. Някои от елементите в масива може да променят стойностите си, но **amessage** винаги ще сочи към едно и също място в паметта (първия елемент на масива).

**pmessage** е указател, който е инициализиран да сочи към стринг (константен). В последствие той може да бъде променен да сочи към друг адрес.



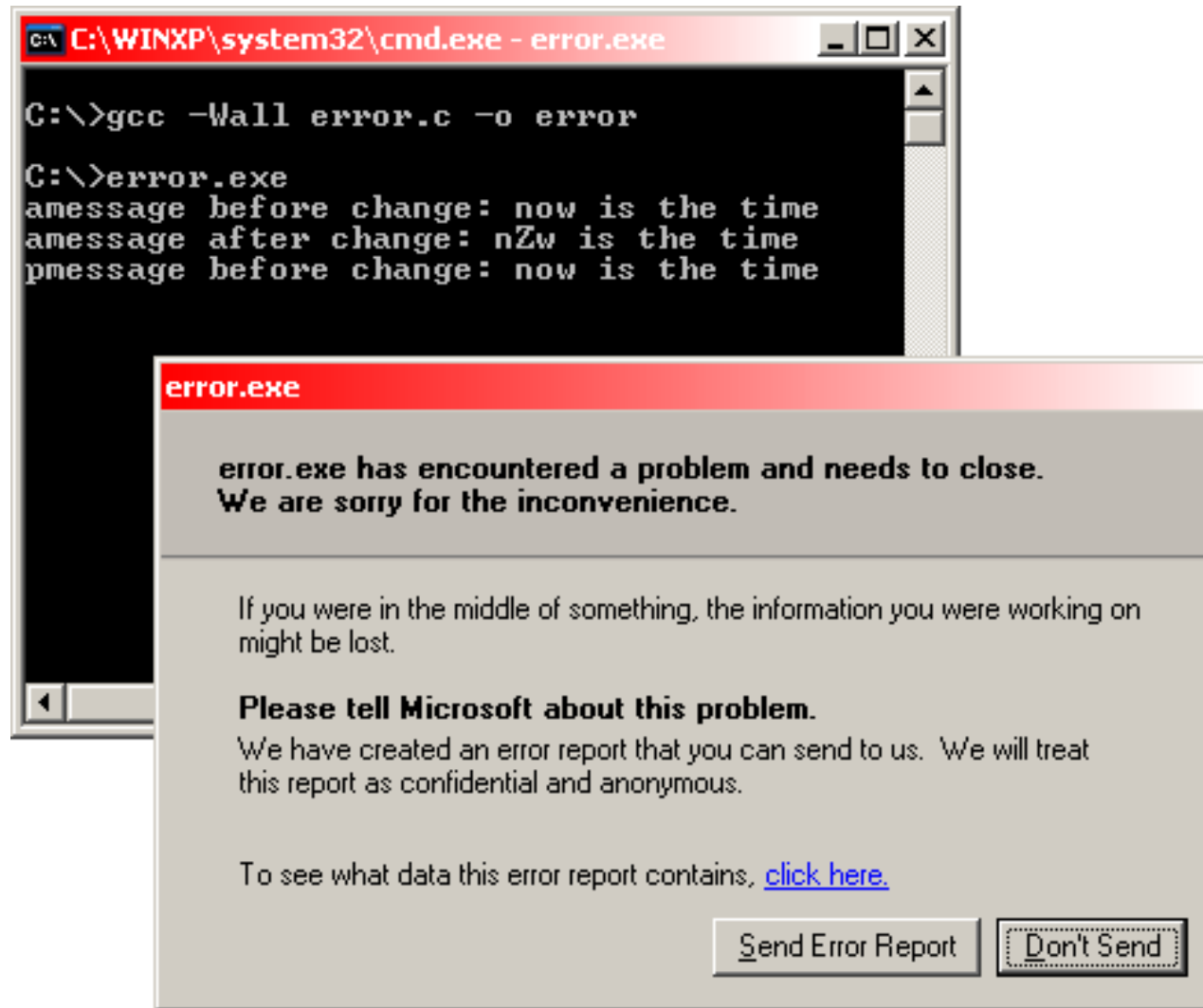


# Пример<sub>1</sub>

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

```
int main() {  
    char amessage[] = "now is the time";  
    char *pmessage = "now is the time";  
  
    printf("amessage before change: %s\n", amessage);  
    amessage[1] = 'Z';  
    printf("amessage after change: %s\n", amessage);  
  
    printf("pmessage before change: %s\n", pmessage);  
    pmessage[1] = 'Z';  
    printf("pmessage after change: %s\n", pmessage);  
    return 0;  
}
```

# Пример<sub>2</sub>



# Пример

```
/* strcpy: copy t to s; array subscript version */
void strcpy(char *s, char *t) {
    int i;
    i = 0;
    while ((s[i] = t[i]) != '\0')
        i++;
}
```

```
/* strcpy: copy t to s; pointer version */
void strcpy(char *s, char *t) {
    while ((*s = *t) != '\0') {
        s++;
        t++;
    }
}
```

# Пример

```
/* strcpy: copy t to s; pointer version 2 */  
void strcpy(char *s, char *t) {  
    while ((*s++ = *t++) != '\0')  
        ;  
}
```

```
/* strcpy: copy t to s; pointer version 3 */  
void strcpy(char *s, char *t) {  
    while (*s++ = *t++)  
        ;  
}
```

# Пример

```
/* strcmp: return <0 if s<t, 0 if s==t, >0 if s>t */
int strcmp(char *s, char *t) {
    int i;
    for (i = 0; s[i] == t[i]; i++)
        if (s[i] == '\0') //или if (!s[i])
            return 0;
    return s[i] - t[i];
}
```

```
int strcmp(char *s, char *t) {
    for ( ; *s == *t; s++, t++)
        if (*s == '\0')
            return 0;
    return *s - *t;
}
```

≡

```
int strcmp(char *s, char *t){
    for ( ; *s == *t; s++, t++)
        if (!*s)
            return 0;
    return *s - *t;
}
```

# Многомерни масиви<sub>1</sub>

С поддържа многомерни масиви. Няма ограничение в броя на измерения, които може да има един масив. Най-често използваният многомерен масив е двумерният.

Един двумерен масив може да бъде разгледан като едномерен масив, на който всеки елемент е масив

```
int arr[10][100]; /*двумерен масив с 10 реда и 100 колони*/  
  
arr[0][5]; //ред 0, колона 5  
arr[i][j];
```

# Многомерни масиви<sub>2</sub>

Когато двумерен масив се предава като аргумент на функция, декларацията на масива задължително трябва да съдържа броя на колоните. Това е така за да може еднозначно да се определи от къде (в паметта) започва даден ред

Броят на редовете не е задължителен.

```
void f (int arr[10][100]) {  
    ...  
}  
void f(int arr[][100]){  
    ...  
}
```

# Пример

```
static char daytab[2][13] = {  
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},  
    {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}  
};
```

```
// колко дена са изминали от началото на годината до даден ден  
int day_of_year(int year, int month, int day) {  
    int i, leap;  
    leap = ((year%4 == 0 && year%100 != 0) || year%400 == 0);  
    for (i = 1; i < month; i++)  
        day += daytab[leap][i];  
    return day;  
}
```

