



Показивачи и НИЗОВИ



Показивач на void

- Показивач на неки тип може узети само адресу тог истог типа, у супротном компајлер јавља упозорење или грешку
- Али то не важи за показивач на void

```
int var;  
float* fptr;  
void* vptr;  
  
fptr = &var; /* compiler warning */  
vptr = &var; /* OK */
```

- Ова могућност је некада потребна - пре свега као механизам превезилажења ограничења које намеће статичка типизираност

```
pthread_create(..., func1, (void*)&args1);  
pthread_create(..., func2, (void*)&args2);  
  
struct params1  
{  
    int handle;  
    int value;  
} args1 = {0x45689216, 57};  
  
struct params2  
{  
    short id;  
    char* ident;  
} args2 = {17, "hd0"};
```

```
void* func1(void* param)  
{  
    struct params1* args;  
    args = (struct params1*)param;  
    printf("%x %d", args->handle, args->value);  
}  
  
void* func2(void* params)  
{  
    struct params2* args;  
    args = (struct params2*)param;  
    printf("%d %s", args->id, args->ident);  
}
```



Показивачи и `const` квалификатор 1/2

- Показивач се може мењати али не и оно на шта он показује
- Кључна реч **`const`** мора бити лево од `*`

```
const type* ptr_variable;  
type const* ptr_variable;
```

```
int var1 = 3;  
int var2 = 5;  
const int* ptr = &var1;  
ptr = var2; /* OK */  
*ptr = 7;   /* error */
```

- Експлицитна конверзија мора бити коришћена када вредност оваквог показивача желимо доделити обичном, неконстантном, показивачу

```
int* ptr;  
int const* cptr;  
  
ptr = cptr; /* compiler warning or error */  
ptr = (int*)cptr; /* OK */
```



Показивачи и `const` квалификатор 2/2



- Могуће је мењати оно на шта показивач показује, али не и сам показивач
- Кључна реч **`const`** мора бити са десне стране `*`

```
type* const ptr_variable;
```

```
int var1 = 3;  
int var2 = 5;  
int* const ptr = &var1;  
ptr = var2; /* error */  
*ptr = 7;   /* OK */
```

- А могућ је и дупло константни показивач

```
const type* const ptr_variable;  
type const* const ptr_variable;
```

```
int var1 = 3;  
int var2 = 5;  
int const* const ptr = &var1;  
ptr = var2; /* error */  
*ptr = 7;   /* error */
```



sizeof оператор

- Унарни оператор који срачунава величину типа и изражава је у бајтима (Подсетити се шта је бајт)
- Ради над променљивом или над типом
 - Када ради над променљивом враћа величину те променљиве, то јест величину типа ког је та променљива
 - Када ради над типом враћа његову величину
- По дефиницији sizeof(char) је 1

```
int* ptr;
size_t s;

s = sizeof(*ptr);
printf("%d\n", s);

s = sizeof(int);
printf("%d\n", s);

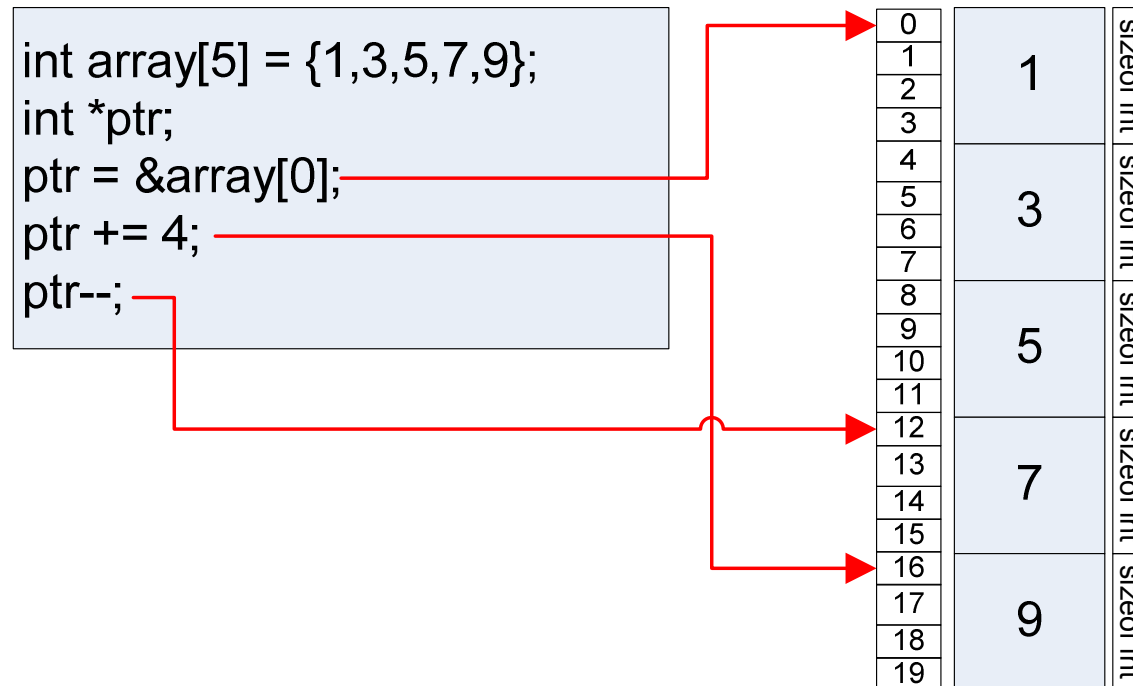
s = sizeof(ptr);
printf("%d\n", s);
```



Показивачи и операције над њима 1/4



- Сабирање и одузимање целог броја:
 - `data_type* ptr;`
`ptr ± n <=> ptr ± n * sizeof(data_type)`
 - Исто важи и за унарне операторе ++/--





Показивачи и операције над њима 2/4



- Одузимање два показивача - само ако су истог типа

```
#include <stddef.h>
int array[5] = {1,3,5,7,9};
int* ptr1;
int* ptr2;
ptrdiff_t diff;

ptr1 = &array[1];
ptr2 = &array[4];
diff = ptr2 - ptr1;
```

diff = 3

- И има смисла само ако показују на адресе унутар истог парчета меморије

```
int array1[5] = {1,3,5,7,9};
int array2[5] = {2,4,6,8,10};
int* ptr1;
int* ptr2;
ptrdiff_t diff;

ptr1 = &array1[1];
ptr2 = &array2[4];
diff = ptr2 - ptr1;
```

Undefined result

Сабирање два показивача
НЕ МОЖЕ



Показивачи и операције над њима 3/4



- Поређење показивача:
 - Могуће поредити само показиваче на објекте

```
int array[5] = {9, 7, 5, 3, 1};  
int* ptr1;  
int* ptr2;  
  
ptr1 = &array[1];  
ptr2 = &array[4];  
if(ptr1 < ptr2)  
    printf("Expected\n");  
else  
    printf("Unexpected\n");
```

Output: Expected

- Исто има смисла само ако показују на адресе унутар истог парчета меморије



Показивачи и операције над њима 4/4



- Оператор []

def: $A[B] \Leftrightarrow *(A + B)$

A + B мора бити показивачког типа јер оператор * ради само са показивачем

Дакле, или A мора бити показивач, а B целобројног типа, или обрнуто!

```
data_type* A; int B;
```

```
A[B] <=> *(A + B) <=> *(A + B * sizeof(data_type))
```

```
data_type* A; int B;
```

```
B[A] <=> *(B + A) <=> *(A + B * sizeof(data_type))
```

```
float* p;  
float x;
```

```
/* neka je p 1000, tj. p pokazuje na adresu 1000 */
```

```
x = *p; // x je float vrednost sa adrese 1000
```

```
x = p[0]; // x je float vrednost sa adrese 1000
```

```
x = p[4]; // x je float vrednost sa adrese 1000 + 4*sizeof(float)
```

```
x = 4[p]; // x je float vrednost sa adrese 1000 + 4*sizeof(float)
```

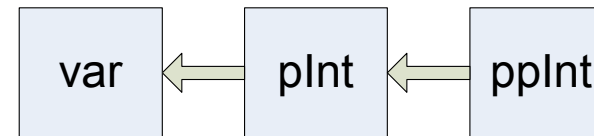


Показивач на показивач 1/2



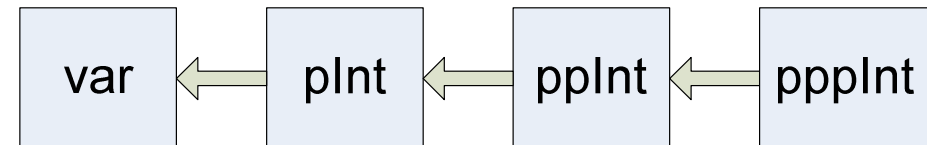
- Показивач може показивати на било који тип, па тако и на тип показивача

```
int var;  
int* pInt = &var;  
int** ppInt = &pInt;
```



- И то тако може у недоглед

```
int var;  
int* pInt = &var;  
int** ppInt = &pInt;  
int*** pppInt = &ppInt;
```





Показивач на показивач 2/2



- Када нам то треба?
- 1. Прослеђивање показивача по референци

```
int g_var;  
  
void bar(int** p)  
{  
    /* change pointer value */  
    *p = &g_var;  
  
    /* change value of variable to  
    which pointer points to */  
    **p = 39;  
}
```

```
void foo()  
{  
    int var;  
    int* ptr= &var;  
    bar(&ptr);  
    ...  
}
```

- 2. Вишедимензионални низови...



Функције са променљивим бројем параметара



```
int printf(const char* format, ...); // декларација  
printf("Student %s sa ocenom %d\n", ime, ocena);
```

- **void va_start(va_list ap, parmN)** – paramN последњи параметар
- **type va_arg(va_list ap, type)**
- **void va_end(va_list ap)**

```
#include <stdarg.h>  
  
void foo(int broj, ...)  
{  
    va_list p1;  
    int i;  
    va_start(p1, broj);  
    for (i = 0; i < broj; ++i)  
    {  
        float x = va_arg(p1, float);  
        printf("%f", x);  
    }  
    va_end(p1);  
}
```

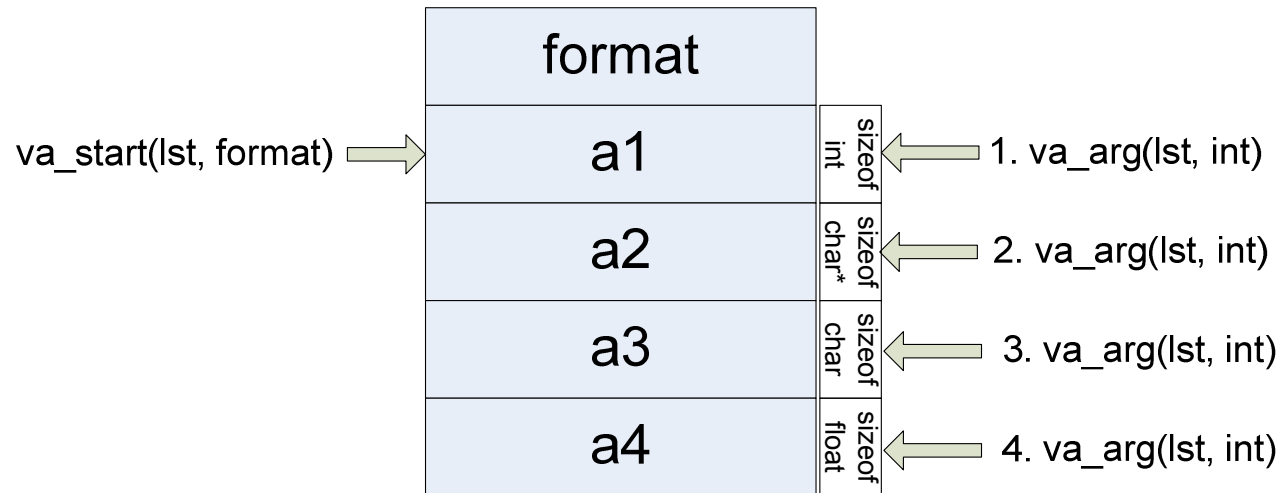


Функције са променљивим бројем параметара



```
int printf(const char* format, ...);
```

```
printf("%d %s %c %f", a1, a2, a3, a4);
```



```
typedef void* va_list;  
#define va_start(ap, paramN) (ap) = &(paramN)  
#define va_end(ap) (ap) = NULL  
#define va_arg(ap, type) (*((type*) (ap))++)
```

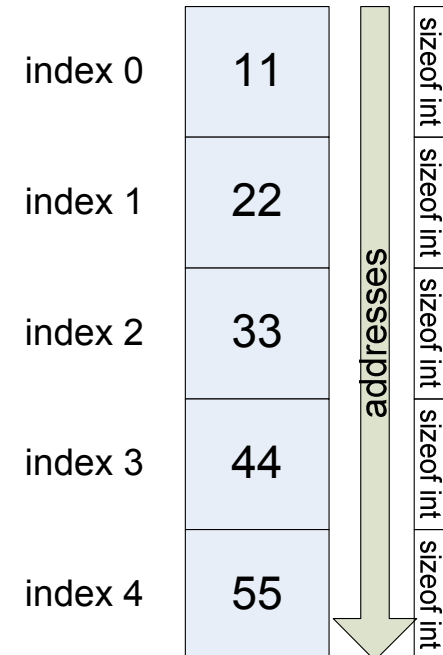


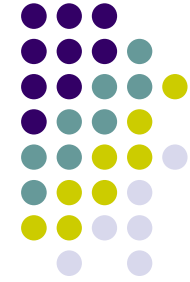
Низови

- Низ у програмском језику Це је тешко одредљива конструкција.
- У суштини представља блок меморије који се тумачи као да се у њему један за другим налазе елементи исте величине
- Кључни аспект низа је његова декларација:
`element_type array_name[dimension] = {declaration_list};`
- По осталим аспектима је јако сличан показивачу који се не може мењати
Рецимо, смисао оператора `[]` је исти

```
int array[5] = {11, 22, 33, 44, 55};  
printf("element with index 3: %d\n", array[3])
```

Output: 44





Иницијализација низа

- Као и код регуларних променљивих, ако нема експлицитне иницијализације, низови статичке трајности ће бити постављени на нулу (сви њихови елементи), а аутоматске трајности неће.

	local					global				
<code>int array[5];</code>	NDF	NDF	NDF	NDF	NDF	0	0	0	0	0

- Листа елемената у витичастим заградама користи се за иницијализацију
- Листа не сме имати више елемената него што је димензија низа. Ако има мање, преостали елементи се попуњавају са нулама.

<code>int array[5] = {1, 3, 5};</code>	1	3	5	0	0
--	---	---	---	---	---

- Ако постоји иницијализација низа, димензија може бити изостављена
Тада ће димензија аутоматски бити једнака величини иницијализаторске листе

```
int array[] = {1, 2, 3}; <=> int array[3] = {1, 2, 3};
```

- У случају да желимо иницијализовати само неке елементе низа, C99 нуди решење:

```
int array[100] = {[13] = 5, [77] = 6};
```



Однос показивача и низова 1/3



- Као што је речено, низови и показивачи су блиски рођаци.

```
int array[] = {1, 3, 5, 7};  
int* ptr = array;  
if (array[0] == ptr[0]  
    && array[1] == ptr[1]  
    && array[2] == ptr[2]  
    && array[3] == ptr[3])  
{  
    printf("equal");  
}  
else  
{  
    printf("not equal");  
}
```

Output: equal



Однос показивача и низова 2/3



- Али иако су рођаци, нису баш исто
- Када се дефинише низ, заузима се меморија за све његове елементе
- Када се дефинише показивач, заузима се меморија само за њега

```
char array[5];  
char* ptr;
```

array:



ptr



- Низ, то јест име низа, представља адресу
- Показивач је променљива чија вредност је адреса
- Отприлике, у овом смислу низ је као показивачки литерал

```
int array[] = {1, 3, 5, 7, 9};  
int* ptr = array;
```

array:



ptr



Однос показивача и низова 3/3



- Разлика у резултату sizeof операције
 - За низ враћа број меморијских речи које су заузеле за цео низ
 - За показивач - колико меморијских речи треба за адресу

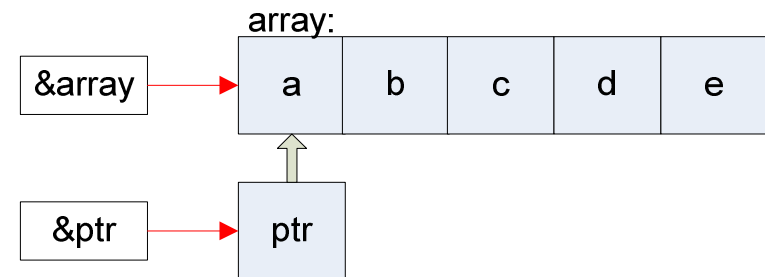
```
char array[15];  
char* ptr;  
  
printf("sizeof array: %d\n", sizeof(array));  
printf("sizeof pointer: %d\n", sizeof(ptr));
```

Output:
sizeof array: 15
sizeof pointer: 4

- Разлика у & оператору
 - За низ враћа адресу првог елемента
 - За показивач - адресу показивачке променљиве

```
char array[] = {'a', 'b', 'c', 'd', 'e'};  
char* ptr = array;
```

```
array[0] == ptr[0]  
&array != &ptr
```





Вишедимензионални НИЗОВИ



- У Цео вишедимензионални низови су у ствари низови низова
- Пример дводимензионалног низа:

```
int matrix[3][4];
```

 - то је низ који има 3 елемента
 - а елементи низа су типа низ од 4 интеџера

Еквивалентна дефиниција:

```
typedef int niz4[4];  
niz4 matrix[3];
```

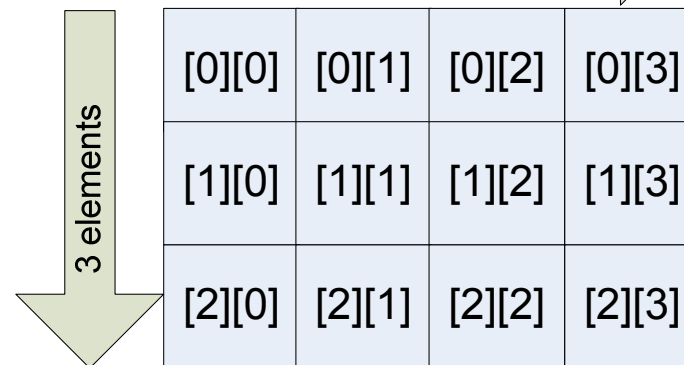
```
int a;  
niz4 y; <=> int y[4];
```

```
y[a]  
*(y + a*sizeof(int))
```

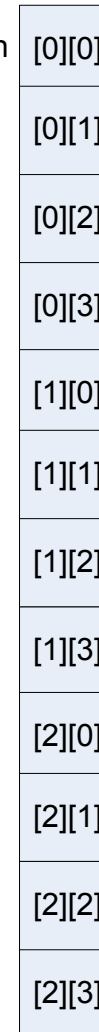
```
int b;  
niz4 x[3]; <=> int x[3][4];
```

```
x[a][b]  
(x[a])[b]  
*(x+a*sizeof(niz4))[b]  
*(x+a*sizeof(niz4)) <=> niz4 T  
T[b]  
*(T+b*sizeof(int))
```

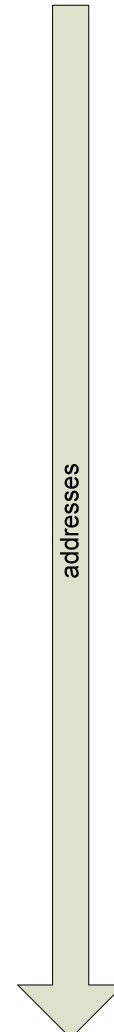
logical
organization



physical
organization
in memory



addresses





Прослеђивање низова функцији



- Не може по вредности
- Увек се прослеђује адреса
- Последица је да су наведене декларације практично исте:

```
int func(int arr[10]);  
int func(int arr[]);  
int func(int* arr);
```

- Број елемената наведен у угластим заградама се игнорише
- За вишедимензионалне низове, димензије морају постојати у свим осим у првим угластим заградама

```
int func (int arr[][7]);  
int func (int* arr[7]);
```

```
element_type name[] [depth1_] ... [depth_n]
```

- Зашто?

```
typedef int niz4[4];  
niz4 x[3]; <=> int x[3][4];  
x[a][b]  
*(T + b*sizeof(int)) // T <=> *(x + a*sizeof(niz4))  
*(*(x + a*sizeof(niz4)) + b*sizeof(int))
```

Број 4 у декларацији је важан за одређивање sizeof(niz4) - број 3 није важан.



Показивачи на функције

1/3



- Ево како се то декларише

```
return_type (*name) (param_type, param_type);
```

- Слично низовима и имена функција се свODE на показивач

```
char* (*fptr) (char* to, const char* from);  
  
fptr = strcpy; /* OK */  
fptr = &strcpy; /* OK */
```

- Позивање функције на коју показивач показује

```
char src[128];  
char dst[128];  
  
fptr(dst, src); /* OK */  
(*fptr) (dst, src); /* OK */
```



Показивачи на функције

2/3



- А могу се декларисати и низови показивача на функције

```
return_type (*name[]) (param_type, param_type);
```

```
char* (*fptr[3])(int x) = {func1, func2, func3};  
  
char* pc = fptr[1](7);  
char c = *pc;  
c = *fptr[1](7); //?
```

```
typedef char* (*fptr_t)(int x);  
fptr_t fptr[] = {func1, func2, func3};
```

- Корисно за табеле скокова, или изведбе аутомата са коначним бројем стања

```
int (*fptr[])(int x) = {state1, state2, state3, state4}  
  
int new_state(int current_state, int input)  
{  
    return fptr[current_state](input);  
}
```



Показивачи на функције

3/3



```
int new_state(int current_state, int input)
{
    switch (current_state)
    {
        case 0:
            return state1(input);
            break;
        case 1:
            return state2(input);
            break;
        case 2:
            return state3(input);
            break;
        case 3:
            return state4(input);
            break;
    }
}
```

```
int (*fptr[])(int x) = {state1, state2, state3, state4}

int new_state(int current_state, int input)
{
    return fptr[current_state](input);
}
```