



Универзитет „Св. Кирил и Методиј“ во Скопје
ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ И
КОМПЈУТЕРСКО ИНЖЕНЕРСТВО

ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ И КОМПЈУТЕРСКО ИНЖЕНЕРСТВО

Покажувачи

Структурно програмирање

ФИНКИ 2014

Концепт на адресирање

- Покажувачите се посебен тип на променливи кои секогаш претставуваат адреса на (друга) мемориска локација т.е. покажуваат кон мемориската локација на дадената адреса
- Покажувачот содржи позитивна целобројна вредност без предзнак, што се интерпретира како мемориска адреса (на која се чува друга променлива)
- Променливите содржат вредности за податокот (директно референцирање)
- Покажуважите содржат адреси на променливи (индиректно референцирање)

За што се користат...

- Зошто се користат покажувачите?
 - Сложените податочни структури полесно се манипулираат со помош на покажувачи
 - Покажувачите овозможуваат ефикасен начин за пристап до големите мемориски множества
 - Со покажувачите е овозможена работа со динамички алоцирана меморија

Покажувачи

- Формат

*tip *pokIme;*

- Секој покажувач има тип. Типот се однесува на типот на променливата кон која тој покажува.
- При декларацијата за секој покажувач мора да се декларира неговиот податочен тип
- Вредноста на секој покажувач е позитивен цел број без предзнак (мемориска адреса), но, како се интерпретира вредноста која се наоѓа на оваа мемориска локација зависи од типот на покажувачот.

Декларација на покажувачи

■ Пример:

```
int *pok; /* se deklarira celobroen pokazuvac */  
double *myPtr1, *myPtr2; /*pokazuvac kon realna  
                           promenliva */  
char *x;
```

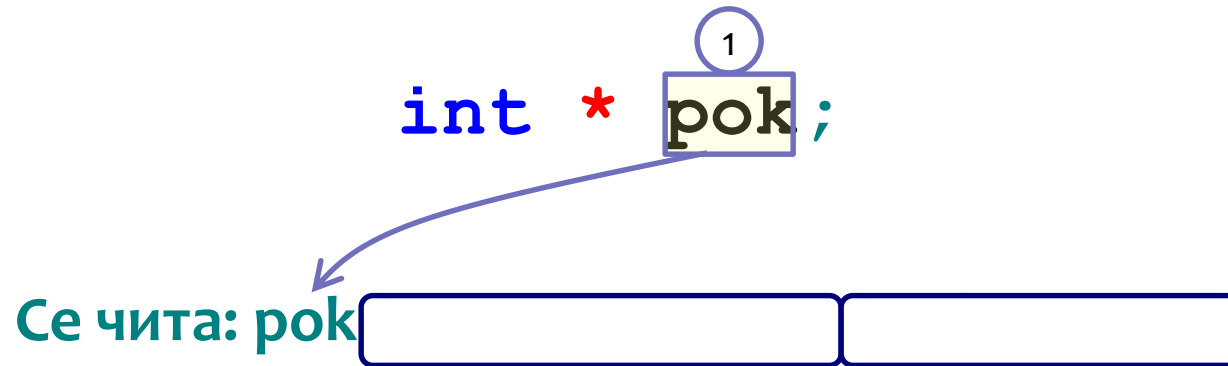
- така разликуваме покажувач кон цел број, покажувач кон реален број, итн.
- може да се декларираат покажувачи од кој и да е податочен тип

Читање на декларација на покажувач

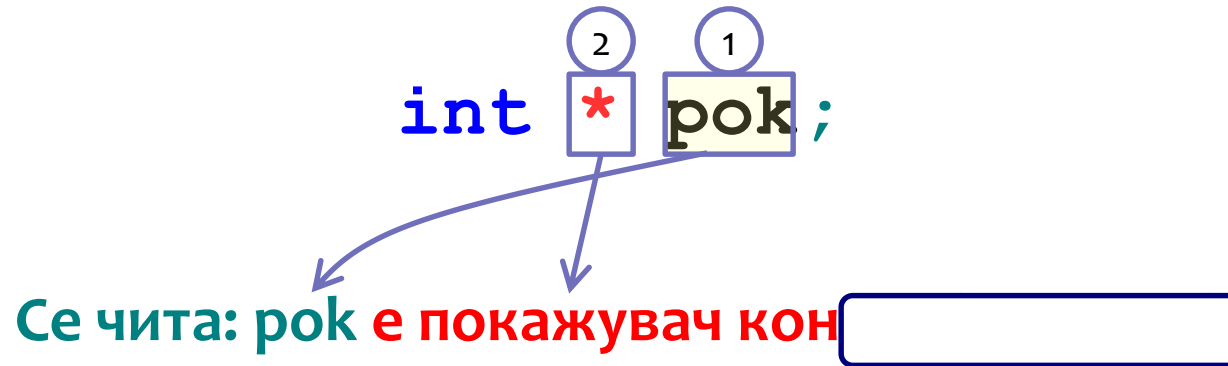
```
int * pok;
```

Се чита:

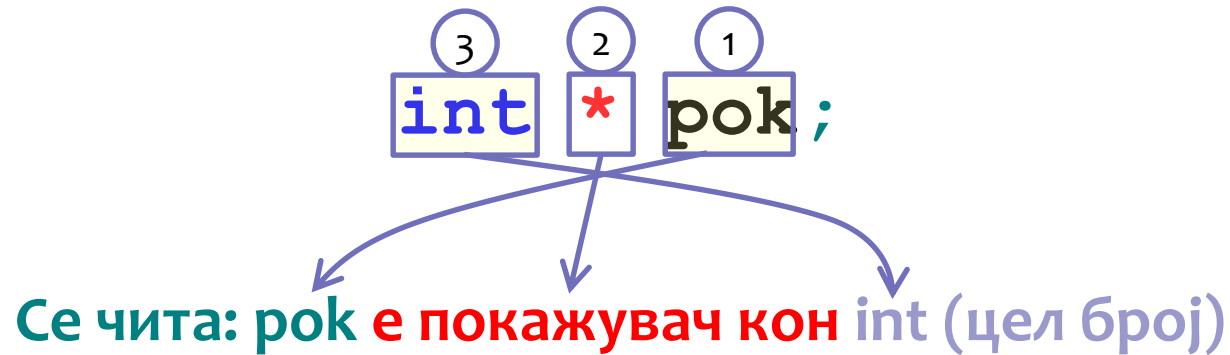
Читање на декларација на покажувач



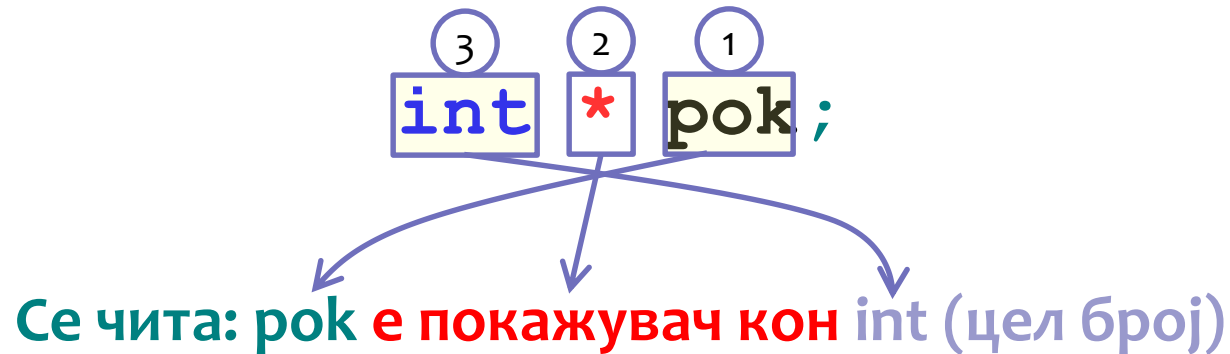
Читање на декларација на покажувач



Читање на декларација на покажувач

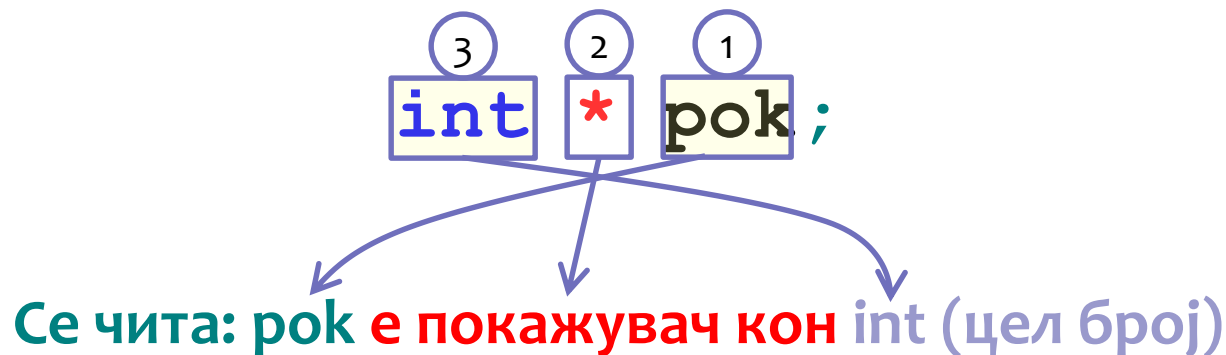


Читање на декларација на покажувач



Типот на променливата **pok** е **int ***
(покажувач кон цел број)

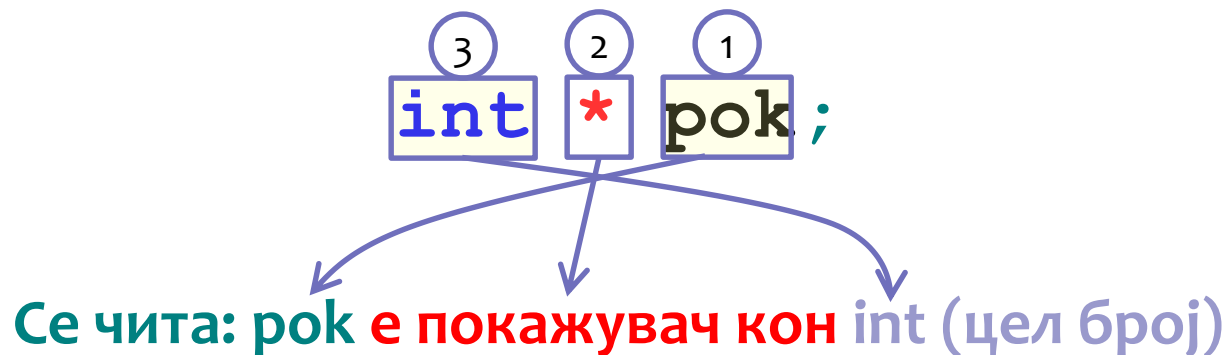
Читање на декларација на покажувач



Типот на променливата `pok` е `int *`
(покажувач кон цел број)

`double *myPtr1;`

Читање на декларација на покажувач



Типот на променливата **pok** е **int ***
(покажувач кон цел број)

double *myPtr1;

Се чита: **myPtr1** е покажувач кон **double** (реален број)

Концепт на мемориски локации, променливи и мемориски адреси

- Операторот **&** (ampersand) е префикс оператор кој ја враќа мемориската адреса на која е сместена променливата
- Оператор ***** (свезда – asterisk) или оператор за дереференцирање - префикс оператор кој ја враќа содржината на мемориската локација чија адреса се наоѓа во променливата покажувач (pok)

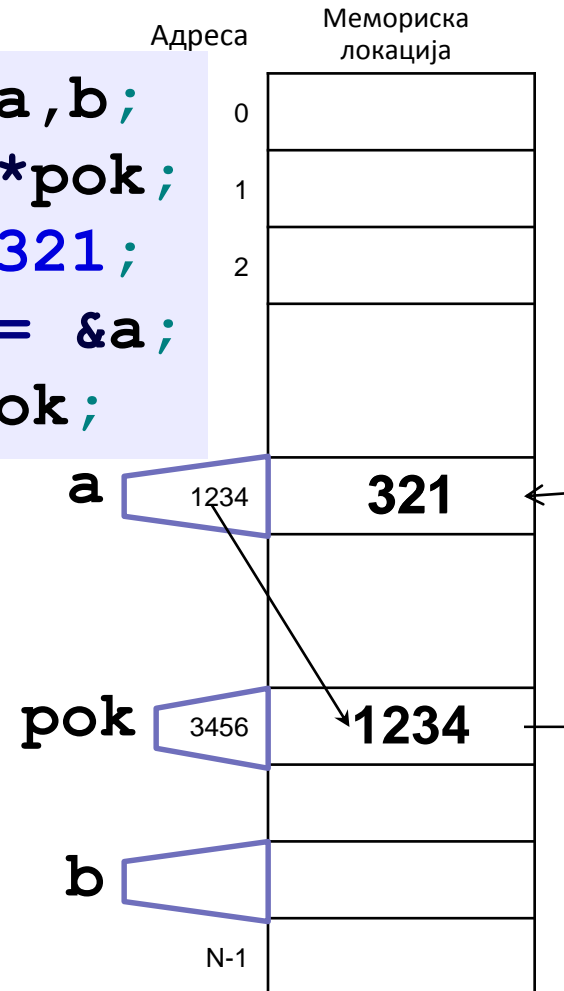
```
int a,b;
int *pok;
a = 321;
pok = &a;
b=*pok;
```

	Адреса	Мемориска локација
	0	
	1	
	2	
a	1234	321
pok	3456	
b		
	N-1	

Концепт на мемориски локации, променливи и мемориски адреси

- Операторот **&** (ampersand) е префикс оператор кој ја враќа мемориската адреса на која е сместена променливата
- Оператор ***** (свезда – asterisk) или оператор за дереференцирање - префикс оператор кој ја враќа содржината на мемориската локација чија адреса се наоѓа во променливата покажувач (pok)

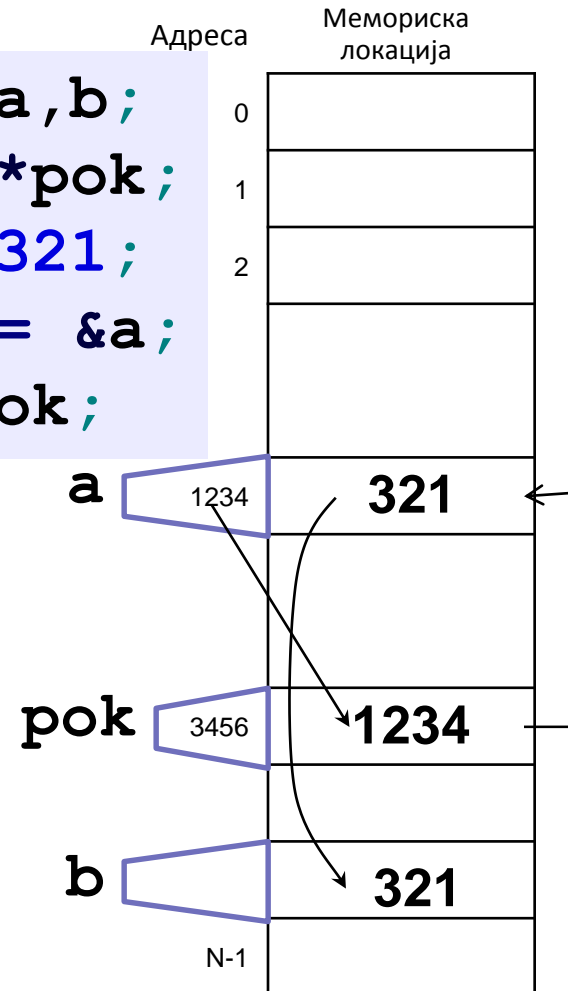
```
int a,b;
int *pok;
a = 321;
pok = &a;
b=*pok;
```



Концепт на мемориски локации, променливи и мемориски адреси

- Операторот **&** (ampersand) е префикс оператор кој ја враќа мемориската адреса на која е сместена променливата
- Оператор ***** (свезда – asterisk) или оператор за дереференцирање - префикс оператор кој ја враќа содржината на мемориската локација чија адреса се наоѓа во променливата покажувач (pok)

```
int a,b;
int *pok;
a = 321;
pok = &a;
b=*pok;
```



Оператори * и &

- Во делот на инструкциите (операторот за дереференцирање) * се чита како „содржината на ...“ или „мемориската локација кон која покажува ...“

`*pok = *pok + 1;`

- Операторот & се чита како „адресата на ...“

`pok = &a;`

- * се & инверзни и се поништуваат

`*&yptr -> *(&yptr) -> *(address of yptr) -> yptr`

`&*yptr -> &(*yptr) -> &(y) -> yptr`

Каде `y` е променлива од типот `int` и `yptr=&y`

Се сеќавате на
`scanf("%d", &i);`

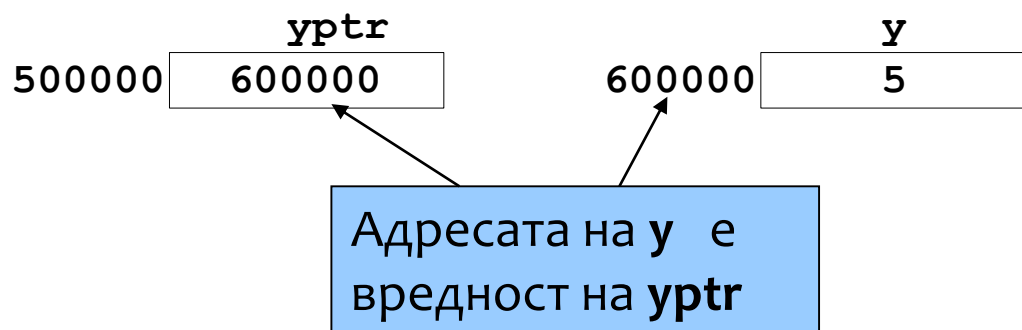
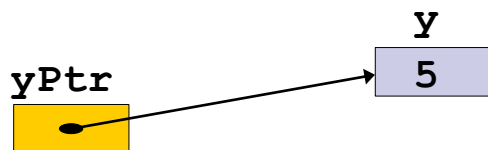
Иницијализација на покажувачи

- `int i = 5;` декларирање и иницијализација на целобројна променлива
- `int *ptr = &i;` декларирање и иницијализација на покажувачот `ptr` да покажува кон `i` (мемориската локација на која е сместена променливата `i`)
- На покажувач од тип `type` може да се додели адреса на променлива од истиот тип `type` или специјалната вредност `NULL` (или `0`) – се користи да означи невалидна вредност на покажувачот



Вредност на покажувач

```
int y = 5;
int *yPtr;
yPtr = &y;
```

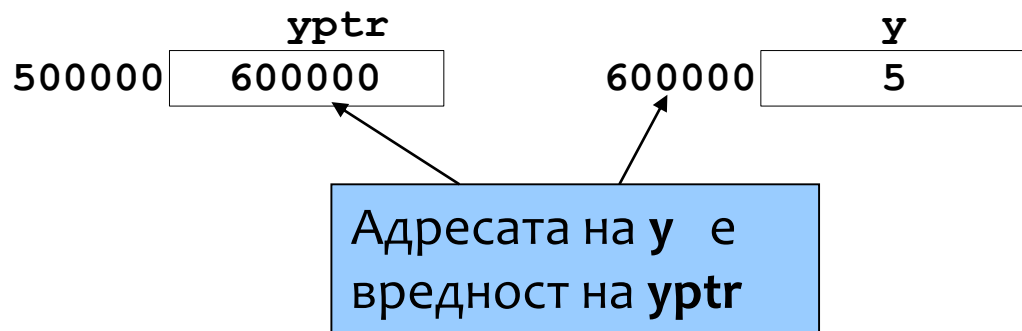
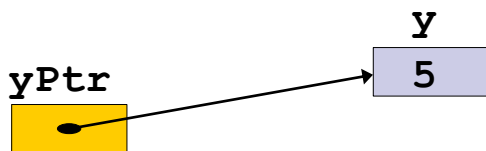


Што ќе биде отпечатено?

```
int i=5, *ptr=&i;
printf("i = %d\n", i);
printf("*ptr = %d\n", *ptr);
printf("ptr = %p\n", ptr);
```

Вредност на покажувач

```
int y = 5;
int *yPtr;
yPtr = &y;
```



Што ќе биде отпечатено?

```
int i=5, *ptr=&i;
printf("i = %d\n", i);
printf("*ptr = %d\n", *ptr);
printf("ptr = %p\n", ptr);
```

Излез:

i = 5

*ptr = 5

ptr = effff5e0

вредност на ptr =
адреса на i во
меморијата

Операции со покажувачи

- И покажувачите се променливи како и сите други. Со нив може да се изведат само одредени операции:
- Инкрементирање/декрементирање на покажувачи (++ или --)
 - Вредноста на покажувачот се зголемува односно намалува за големината на мемориската локација на која покажува покажувачот → покажувачот ќе покажува не следната/претходната локација
- Додавање на целобројна вредност на покажувач (+ или += , - или -=)
- Одземање на покажувачи
 - го враќа бројот на елементи меѓу двете адреси

Изрази со покажувачи

Пример:

- $p = p+1; \quad p++;$ - двата изрази извршуваат иста операција, и овозможуваат p да покажува кон следниот мемориски елемент по елементот на кој почетно покажувал покажувачот p .
- $q = p+i;$ - q покажува кон податочниот елемент што се наоѓа i позиции по елементот на кој покажува p .
- $n = q - p;$ - n е број на елементи меѓу p и q , и претставува целобројна вредност.

Аритметика со покажувачи

- На покажувачите може да се додели само адреса (вредност на друг покажувач од истиот тип). Покажувачи од ист тип може да се употребат во наредби за доделување на вредност
 - ако не се од ист тип потребно е користење на **cast** оператор
 - исклучок: покажувач од типот **void (void *)**
 - генерички покажувач, покажува кон кој и да е тип (покажувач кон што било)
 - не е неопходен **cast** оператор за да се конвертира вредноста на покажувачот во **void** покажувач

Споредување на покажувачи

■ Споредба на покажувачи ($<$, $==$, $>$, $!=$)

□ пример

$q == p$ и $q < p$ се релативни изрази,

$q == p$ - дали q и p покажуваат кон иста мемориска адреса,

$q < p$ - дали елементот кон кој покажува q претходи на елементот на кој покажува p .

Доделување на вредност на покажувач

- На нив може да се додели само адреса (вредност на друг покажувач од истиот тип)

```
int x, *p, *q;  
float *f;  
void *v;
```

- КУСА ПРОВЕРКА

- Што значи **ptr + 1**?
- Што означува **ptr - 1**?
- Што означуваат **ptr*2** и **ptr/2**?

```
p=123;
```

```
p=&x;
```

```
q=p;
```

```
q=p+5;
```

```
v=p;
```

```
q=v;
```

```
f=q;
```


Доделување на вредност на покажувач

- На нив може да се додели само адреса (вредност на друг покажувач од истиот тип)

```
int x, *p, *q;
float *f;
void *v;
```

- КУСА ПРОВЕРКА

- Што значи **ptr + 1**?
- Што означува **ptr - 1**?
- Што означуваат **ptr*2** и **ptr/2**?

```
p=123; ❌
```

```
p=&x;
```

```
q=p;
```

```
q=p+5;
```

```
v=p;
```

```
q=v;
```

```
f=q;
```

Доделување на вредност на покажувач

- На нив може да се додели само адреса (вредност на друг покажувач од истиот тип)

```
int x, *p, *q;
float *f;
void *v;
```

- КУСА ПРОВЕРКА

- Што значи **ptr + 1**?
- Што означува **ptr - 1**?
- Што означуваат **ptr*2** и **ptr/2**?

p=123; 

p=&x; 

q=p;

q=p+5;

v=p;

q=v;

f=q;

Доделување на вредност на покажувач

- На нив може да се додели само адреса (вредност на друг покажувач од истиот тип)

```
int x, *p, *q;  
float *f;  
void *v;
```

- КУСА ПРОВЕРКА
- Што значи **ptr + 1**?
- Што означува **ptr - 1**?
- Што означуваат **ptr*2** и **ptr/2**?

```
p=123; 
```

```
p=&x; 
```

```
q=p; 
```

```
q=p+5;
```

```
v=p;
```

```
q=v;
```

```
f=q;
```

Доделување на вредност на покажувач

- На нив може да се додели само адреса (вредност на друг покажувач од истиот тип)

```
int x, *p, *q;  
float *f;  
void *v;
```

- КУСА ПРОВЕРКА
- Што значи **ptr + 1**?
- Што означува **ptr - 1**?
- Што означуваат **ptr*2** и **ptr/2**?

p=123; 

p=&x; 

q=p; 

q=p+5; 

v=p;

q=v;

f=q;

Доделување на вредност на покажувач

- На нив може да се додели само адреса (вредност на друг покажувач од истиот тип)

```
int x, *p, *q;
float *f;
void *v;
```

- КУСА ПРОВЕРКА
- Што значи **ptr + 1**?
- Што означува **ptr - 1**?
- Што означуваат **ptr*2** и **ptr/2**?

p=123; 

p=&x; 

q=p; 

q=p+5; 

v=p; 

q=v;

f=q;

Доделување на вредност на покажувач

- На нив може да се додели само адреса (вредност на друг покажувач од истиот тип)

```
int x, *p, *q;
float *f;
void *v;
```

■ КУСА ПРОВЕРКА

- Што значи **ptr + 1**?
- Што означува **ptr - 1**?
- Што означуваат **ptr*2** и **ptr/2**?

p=123; 

p=&x; 

q=p; 

q=p+5; 

v=p; 

q=v; 

f=q;

Доделување на вредност на покажувач

- На нив може да се додели само адреса (вредност на друг покажувач од истиот тип)

```
int x, *p, *q;  
float *f;  
void *v;
```

- КУСА ПРОВЕРКА

- Што значи **ptr + 1**?
- Што означува **ptr - 1**?
- Што означуваат **ptr*2** и **ptr/2**?

p=123; 

p=&x; 

q=p; 

q=p+5; 

v=p; 

q=v; 

f=q; 

Доделување на вредност на покажувач

```
int x=56, y;  
int *p, *q;  
float f, *r;
```

```
void *v;
```

```
p=123;
```

```
p=&x;
```

```
q=p;
```

```
q=p+5;
```

```
v=p;
```

```
q=v;
```

```
q=(int *)v;
```

```
*p=3.21;
```

```
*p=321;
```

```
r=&f;
```

```
*r=7.8;
```

```
*r=*q;
```

```
*v=456;
```


Доделување на вредност на покажувач

```
int x=56, y;  
int *p, *q;  
float f, *r;
```

```
void *v;
```

```
p=123; 
```

```
p=&x;
```

```
q=p;
```

```
q=p+5;
```

```
v=p;
```

```
q=v;
```

```
q=(int *)v;
```

```
*p=3.21;
```

```
*p=321;
```

```
r=&f;
```

```
*r=7.8;
```

```
*r=*q;
```

```
*v=456;
```

Доделување на вредност на покажувач

```
int x=56, y;  
int *p, *q;  
float f, *r;
```

```
void *v;
```

```
p=123; 
```

```
p=&x; 
```

```
q=p;
```

```
q=p+5;
```

```
v=p;
```

```
q=v;
```

```
q=(int *)v;
```

```
*p=3.21;
```

```
*p=321;
```

```
r=&f;
```

```
*r=7.8;
```

```
*r=*q;
```

```
*v=456;
```

Доделување на вредност на покажувач

```
int x=56, y;  
int *p, *q;  
float f, *r;
```

```
void *v;
```

```
p=123; 
```

```
p=&x; 
```

```
q=p; 
```

```
q=p+5;
```

```
v=p;
```

```
q=v;
```

```
q=(int *)v;
```

```
*p=3.21;
```

```
*p=321;
```

```
r=&f;
```

```
*r=7.8;
```

```
*r=*q;
```

```
*v=456;
```

Доделување на вредност на покажувач

```
int x=56, y;  
int *p, *q;  
float f, *r;
```

```
void *v;
```

```
p=123; 
```

```
p=&x; 
```

```
q=p; 
```

```
q=p+5; 
```

```
v=p;
```

```
q=v;
```

```
q=(int *)v;
```

```
*p=3.21;
```

```
*p=321;
```

```
r=&f;
```

```
*r=7.8;
```

```
*r=*q;
```

```
*v=456;
```

Доделување на вредност на покажувач

```
int x=56, y;  
int *p, *q;  
float f, *r;
```

```
void *v;
```

```
p=123; 
```

```
p=&x; 
```

```
q=p; 
```

```
q=p+5; 
```

```
v=p; 
```

```
q=v;
```

```
q=(int *)v;
```

```
*p=3.21;
```

```
*p=321;
```

```
r=&f;
```

```
*r=7.8;
```

```
*r=*q;
```

```
*v=456;
```

Доделување на вредност на покажувач

```
int x=56, y;  
int *p, *q;  
float f, *r;
```

```
void *v;
```

```
p=123; 
```

```
p=&x; 
```

```
q=p; 
```

```
q=p+5; 
```

```
v=p; 
```

```
q=v; 
```

```
q=(int *)v;
```

```
*p=3.21;
```

```
*p=321;
```

```
r=&f;
```

```
*r=7.8;
```

```
*r=*q;
```

```
*v=456;
```

Доделување на вредност на покажувач

```
int x=56, y;  
int *p, *q;  
float f, *r;
```

```
void *v;
```

```
p=123; 
```

```
p=&x; 
```

```
q=p; 
```

```
q=p+5; 
```

```
v=p; 
```

```
q=v; 
```

```
q=(int *)v; 
```

```
*p=3.21;
```

```
*p=321;
```

```
r=&f;
```

```
*r=7.8;
```

```
*r=*q;
```

```
*v=456;
```

Доделување на вредност на покажувач

```
int x=56, y;  
int *p, *q;  
float f, *r;
```

```
void *v;
```


```
p=123; 
```

```
p=&x; 
```

```
q=p; 
```

```
q=p+5; 
```

```
v=p; 
```

```
q=v; 
```

```
q=(int *)v; 
```

```
*p=3.21; 
```

```
*p=321;
```

```
r=&f;
```

```
*r=7.8;
```

```
*r=*q;
```

```
*v=456;
```


Доделување на вредност на покажувач

```
int x=56, y;  
int *p, *q;  
float f, *r;
```

```
void *v;
```

```
p=123; 
```

```
p=&x; 
```

```
q=p; 
```

```
q=p+5; 
```

```
v=p; 
```

```
q=v; 
```

```
q=(int *)v; 
```

```
*p=3.21; 
```

```
*p=321; 
```

```
r=&f;
```

```
*r=7.8;
```

```
*r=*q;
```

```
*v=456;
```

Доделување на вредност на покажувач

```
int x=56, y;  
int *p, *q;  
float f, *r;
```

```
void *v;
```

```
p=123; 
```

```
p=&x; 
```

```
q=p; 
```

```
q=p+5; 
```

```
v=p; 
```

```
q=v; 
```

```
q=(int *)v; 
```

```
*p=3.21; 
```

```
*p=321; 
```

```
r=&f; 
```

```
*r=7.8;
```

```
*r=*q;
```

```
*v=456;
```

Доделување на вредност на покажувач

```
int x=56, y;  
int *p, *q;  
float f, *r;
```

```
void *v;
```

```
p=123; 
```

```
p=&x; 
```

```
q=p; 
```

```
q=p+5; 
```


```
v=p; 
```

```
q=v; 
```

```
q=(int *)v; 
```

```
*p=3.21; 
```

```
*p=321; 
```

```
r=&f; 
```

```
*r=7.8; 
```

```
*r=*q;
```

```
*v=456;
```

Доделување на вредност на покажувач

```
int x=56, y;  
int *p, *q;  
float f, *r;
```

```
void *v;
```

```
p=123; 
```

```
p=&x; 
```

```
q=p; 
```

```
q=p+5; 
```

```
v=p; 
```

```
q=v; 
```

```
q=(int *)v; 
```

```
*p=3.21; 
```

```
*p=321; 
```

```
r=&f; 
```

```
*r=7.8; 
```

```
*r=*q; 
```

```
*v=456;
```

Доделување на вредност на покажувач

```
int x=56, y;  
int *p, *q;  
float f, *r;
```

```
void *v;
```

```
p=123; 
```

```
p=&x; 
```

```
q=p; 
```

```
q=p+5; 
```

```
v=p; 
```

```
q=v; 
```

```
q=(int *)v; 
```

```
*p=3.21; 
```

```
*p=321; 
```

```
r=&f; 
```

```
*r=7.8; 
```

```
*r=*q; 
```

```
*v=456; 
```

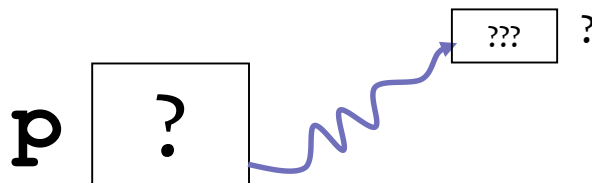
Дереференцирање на неиницијализиран покажувач

- Покажувачот мора да има вредност пред да го дереференцирате (за да го следите покажувачот)



`int * p;` неиницијализиран покажувач (покажува ... кој-знае-каде)

`*p = 7;` ја менува содржината на таа мемориска локација



Обид да се смести вредност во непозната мемориска локација ќе резултира во *run-time грешка*, или полошо, во *логичка грешка*

Адресата во `p` може да биде која било мемориска локација

```
int * p;
p=&nekoja_promenлива;
*p = 7;
```

Менување на содржина со употреба на покажувач

```
int x, *p;
```

```
p = &x;
```

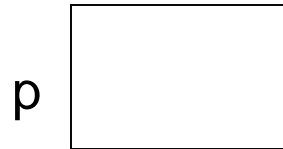
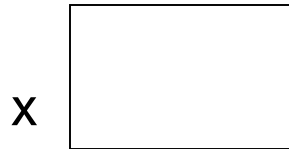
```
*p = 4;
```

Менување на содржина со употреба на покажувач

```
int x, *p;
```

```
p = &x;
```

```
*p = 4;
```



Менување на содржина со употреба на покажувач

```
int x, *p;
```

```
p = &x;
```

```
*p = 4;
```

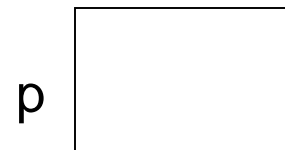
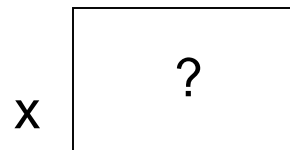
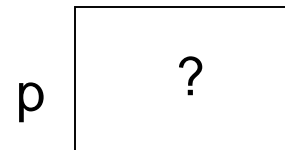
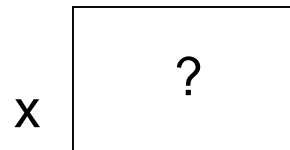


Менување на содржина со употреба на покажувач

```
int x, *p;
```

```
p = &x;
```

```
*p = 4;
```

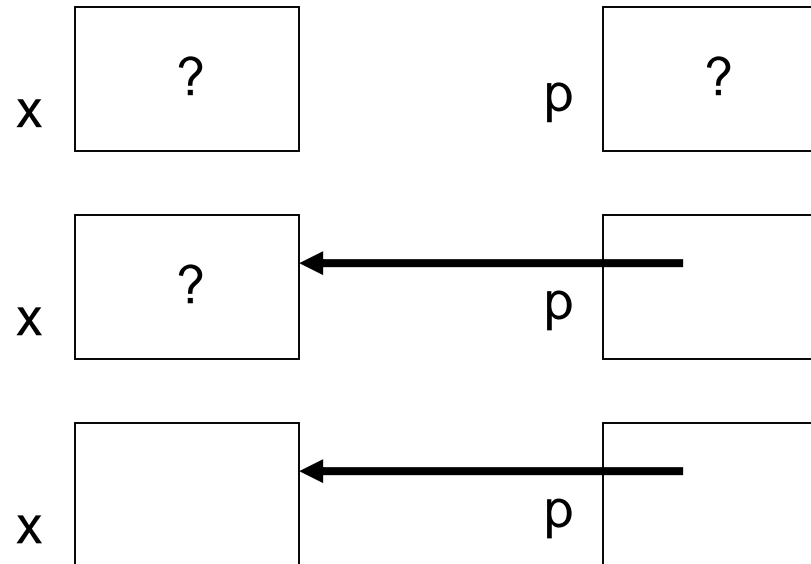


Менување на содржина со употреба на покажувач

```
int x, *p;
```

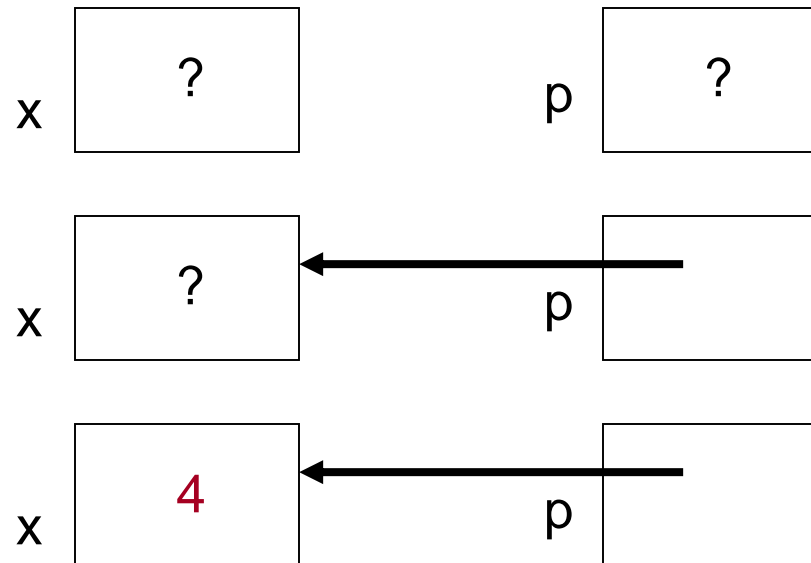
```
p = &x;
```

```
*p = 4;
```



Менување на содржина со употреба на покажувач

```
int x, *p;
p = &x;
*p = 4;
```



со изразот

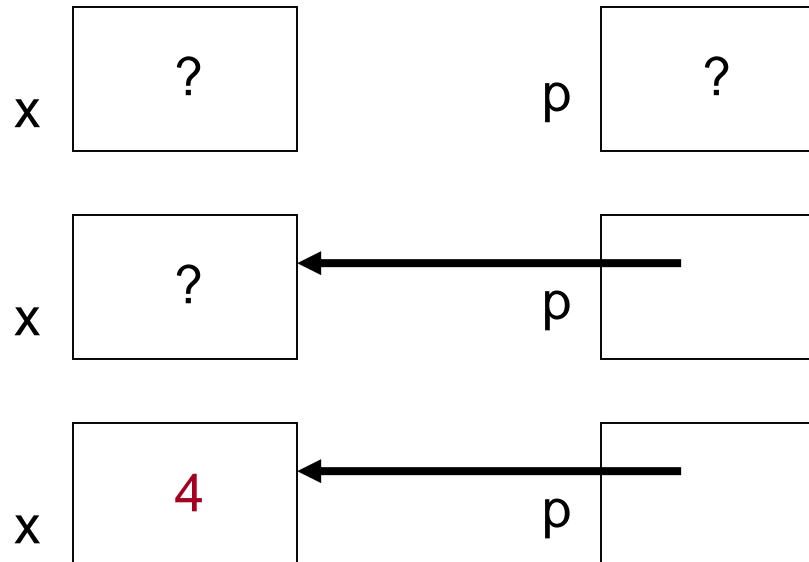
`*p = 4;`

е изменета
вредноста на **`x`**

Менување на содржина со употреба на покажувач

```
int x, *p;
p = &x;
*p = 4;
```

int x, *p;



со изразот

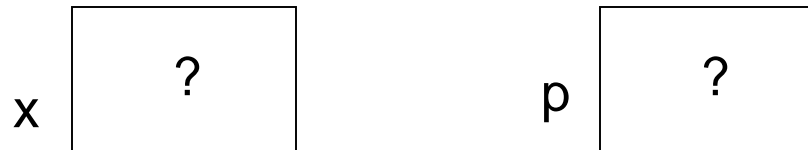
***p = 4;**

е изменета
вредноста на **x**

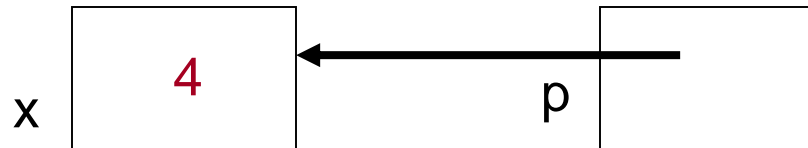
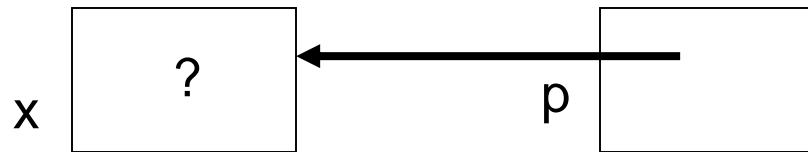
Менување на содржина со употреба на покажувач

```
int x, *p;  
p = &x;  
*p = 4;
```

int x, *p;



p = &x;



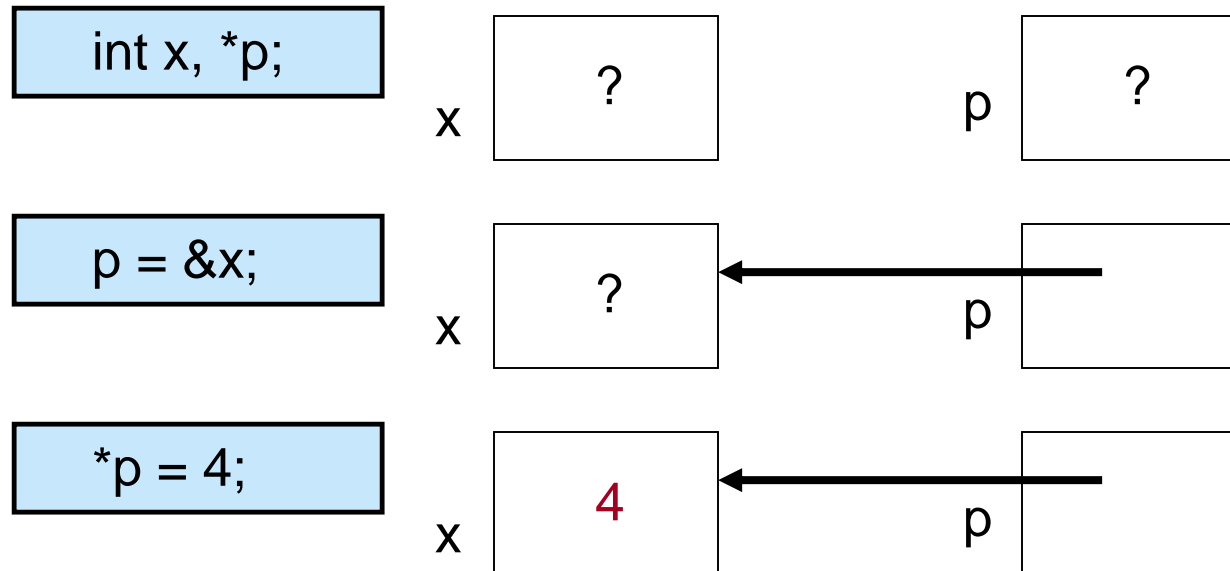
со изразот

`*p = 4;`

е изменета
вредноста на **x**

Менување на содржина со употреба на покажувач

```
int x, *p;
p = &x;
*p = 4;
```



со изразот

`*p = 4;`

е изменета
вредноста на **x**

Пример за работа со покажувачи

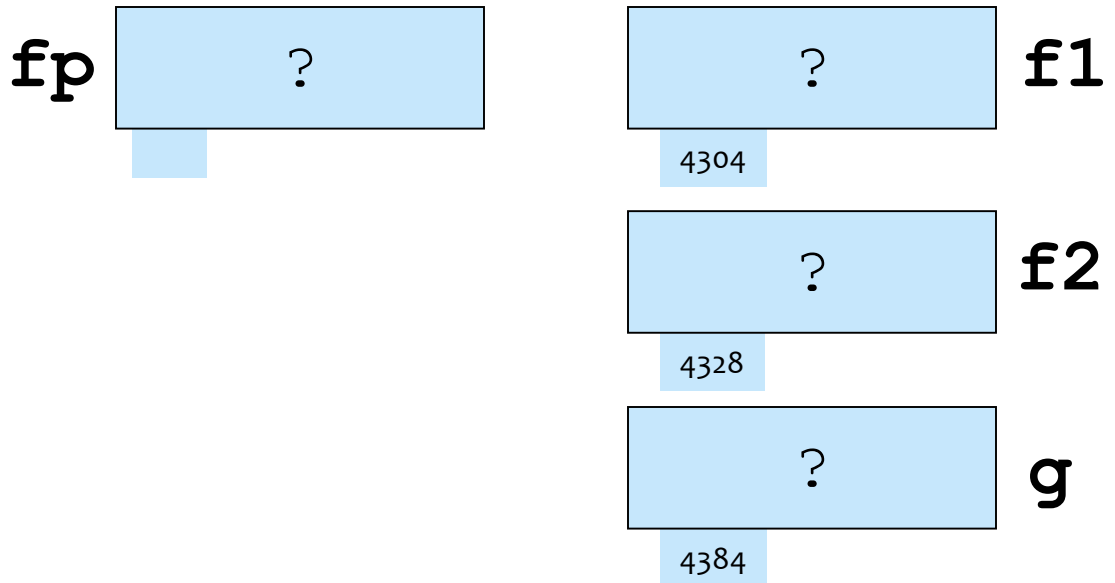
```
float f1, f2, g;  
float *fp;
```

```
fp = &f1;  
*fp=1.2;
```

```
fp = &f2;  
*fp=2.3;
```

```
g = *fp;  
*fp = 3.4;
```

```
printf("%.1f %.1f %.1f\n", f1, f2, g);
```



Пример за работа со покажувачи

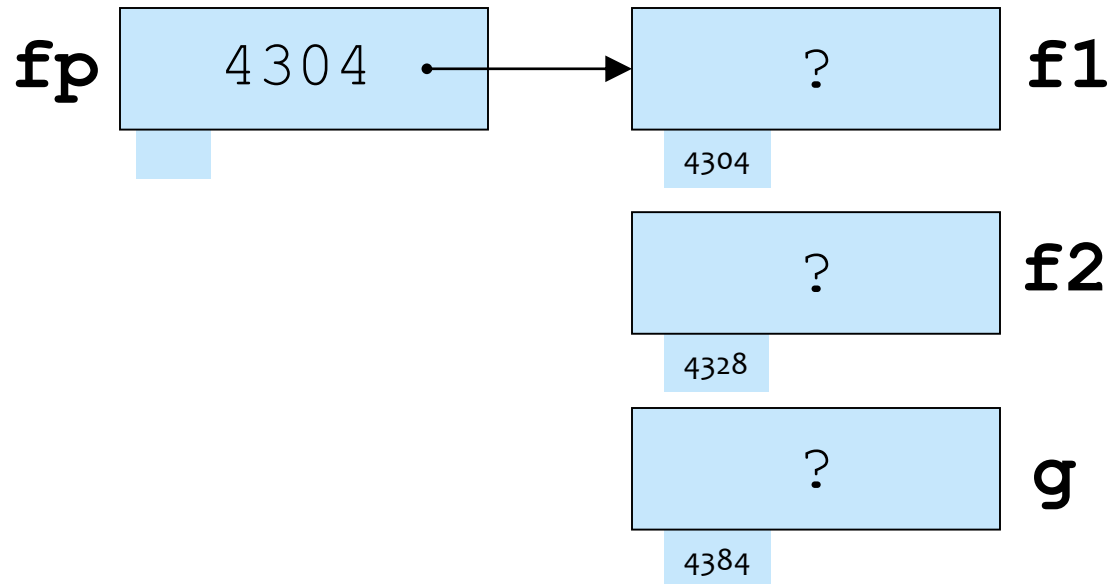
```
float f1, f2, g;  
float *fp;
```

```
fp = &f1;  
*fp=1.2;
```

```
fp = &f2;  
*fp=2.3;
```

```
g = *fp;  
*fp = 3.4;
```

```
printf("%.1f %.1f %.1f\n", f1, f2, g);
```



Пример за работа со покажувачи

```
float f1, f2, g;  
float *fp;
```

```
fp = &f1;
```

```
*fp=1.2;
```

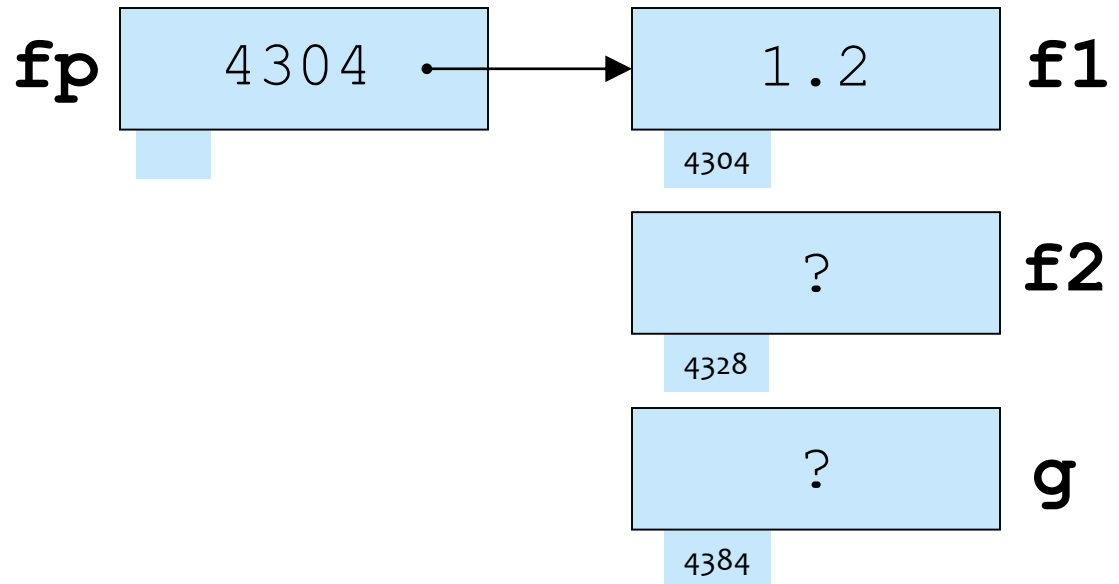
```
fp = &f2;
```

```
*fp=2.3;
```

```
g = *fp;
```

```
*fp = 3.4;
```

```
printf("%.1f %.1f %.1f\n", f1, f2, g);
```



Пример за работа со покажувачи

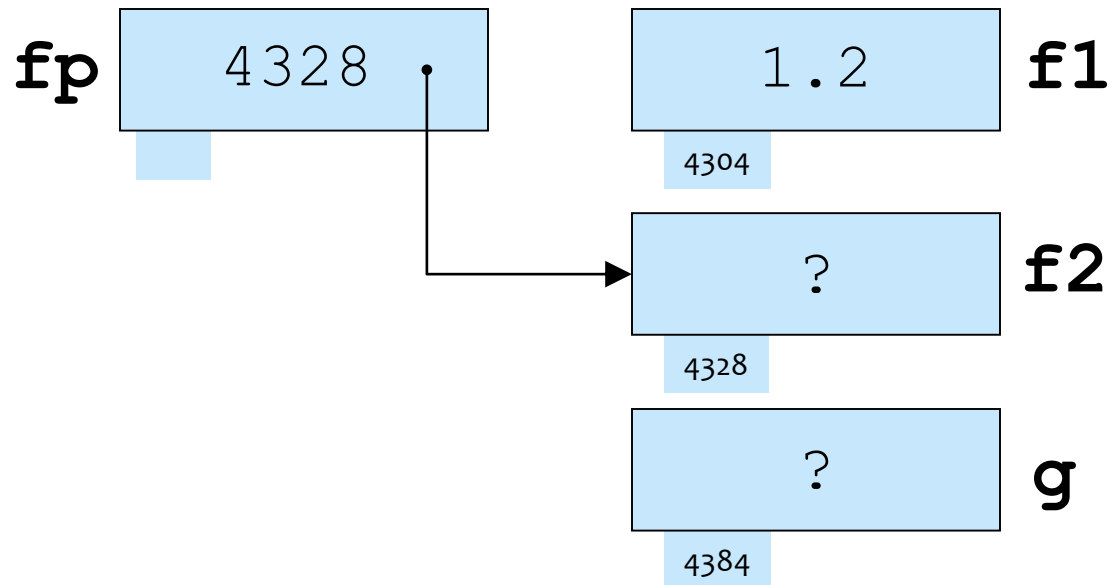
```
float f1, f2, g;  
float *fp;
```

```
fp = &f1;  
*fp=1.2;
```

```
fp = &f2;  
*fp=2.3;
```

```
g = *fp;  
*fp = 3.4;
```

```
printf("%.1f %.1f %.1f\n", f1, f2, g);
```



Пример за работа со покажувачи

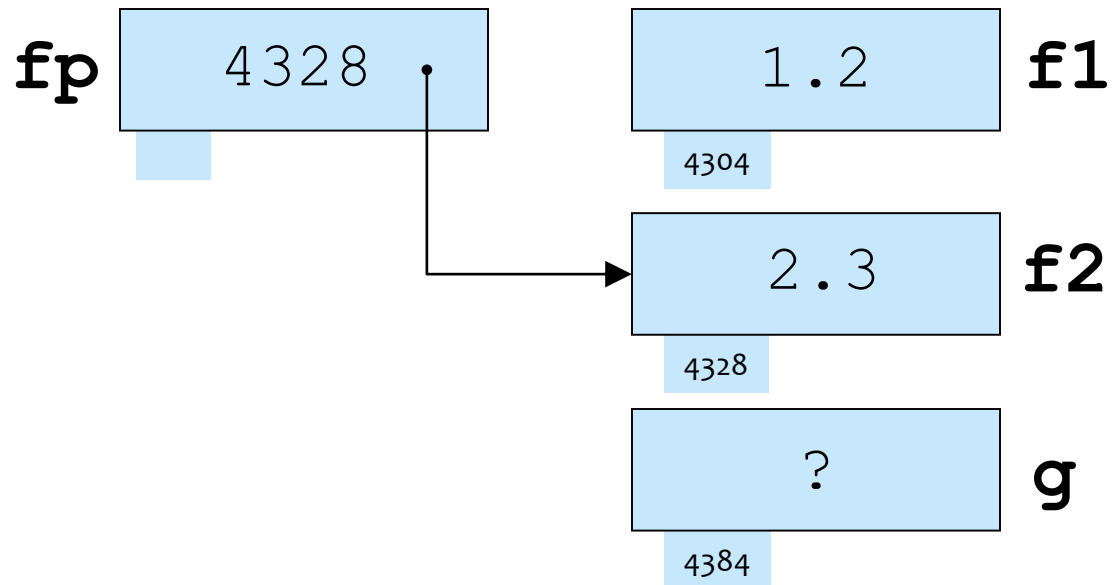
```
float f1, f2, g;  
float *fp;
```

```
fp = &f1;  
*fp=1.2;
```

```
fp = &f2;  
*fp=2.3;
```

```
g = *fp;  
*fp = 3.4;
```

```
printf("%.1f %.1f %.1f\n", f1, f2, g);
```



Пример за работа со покажувачи

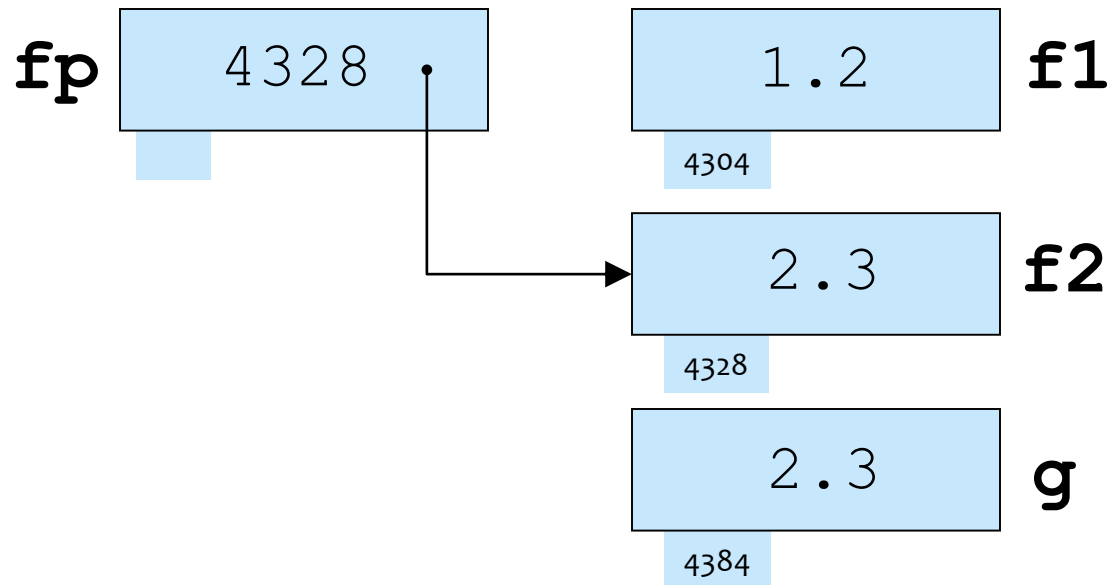
```
float f1, f2, g;  
float *fp;
```

```
fp = &f1;  
*fp=1.2;
```

```
fp = &f2;  
*fp=2.3;
```

```
g = *fp;  
*fp = 3.4;
```

```
printf("%.1f %.1f %.1f\n", f1, f2, g);
```



Пример за работа со покажувачи

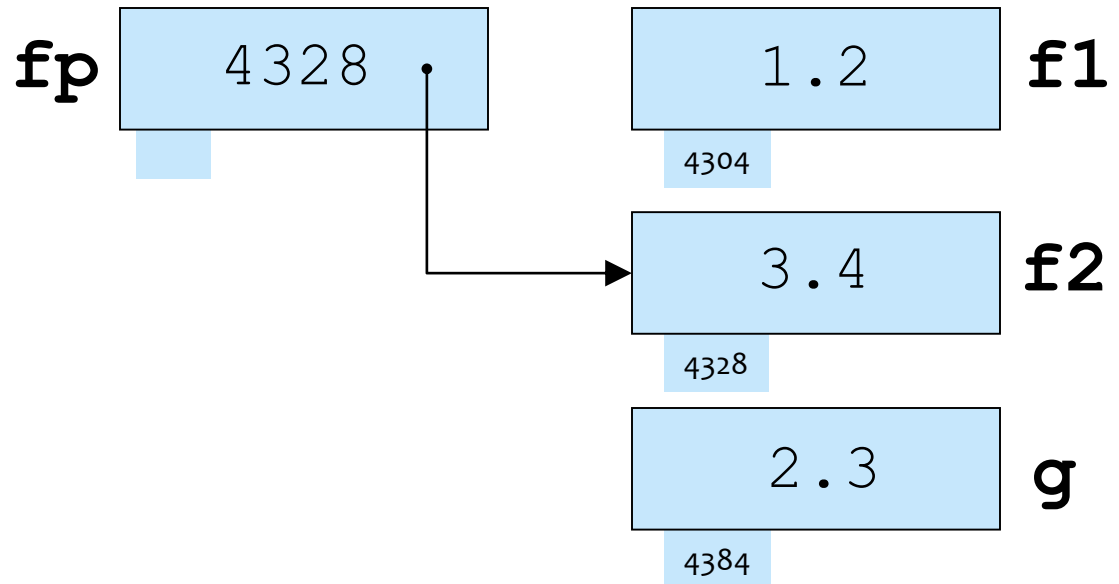
```
float f1, f2, g;  
float *fp;
```

```
fp = &f1;  
*fp=1.2;
```

```
fp = &f2;  
*fp=2.3;
```

```
g = *fp;  
*fp = 3.4;
```

```
printf("%.1f %.1f %.1f\n", f1, f2, g);
```



Пример за работа со покажувачи

```
float f1, f2, g;
float *fp;
```

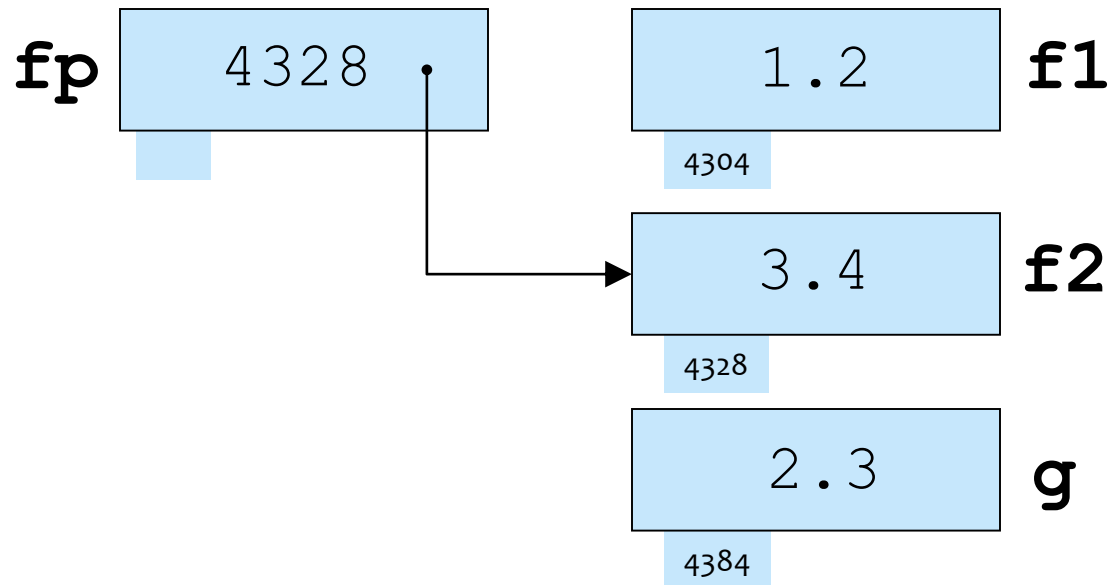
```
fp = &f1;
*fp=1.2;
```

```
fp = &f2;
*fp=2.3;
```

```
g = *fp;
*fp = 3.4;
```

```
printf("%.1f %.1f %.1f\n", f1, f2, g);
```

1.2 3.4 2.3



Употреба на покажувачи

- За пренесување на аргументи на функции
- За пристап до елементи од полиња
 - Пренос на полиња како аргументи на функции
- Пристап до динамички алоцирана меморија
- Освен покажувачи на променливи може да се дефинираат и покажувачи на функции

Константни покажувачи

```
int a=1, b=2, x;  
int const * cip = &a; /* покажувач кон константна меморија */  
x=*cip;  
cip=&b; /* cip може да се пренасочи да покажува кон друга локација */  
*cip=a; /* не може - cip секогаш покажува на константна меморија */  
/* error: assignment of read-only location '*cip' */  
  
int * const icp = &a; /* константен покажувач кон меморија */  
x=*icp;*icp=b; /* може да се промени содржината на мемориската локација на  
која покажува icp */  
icp=&b; /* не може да се пренасочи на друга локација */  
/* error: assignment of read-only variable 'cip' */  
  
int const * const icp = &a; /* константен покажувач кон константна меморија*/
```

Полиња и покажувачи

- Името на полето е всушност константен покажувач кој покажува кон првиот елемент од полето

- Ако важи `int a[10]`, тогаш

- ☐ `a` е `int * const`

- ☐ `a` значи `&a[0]`

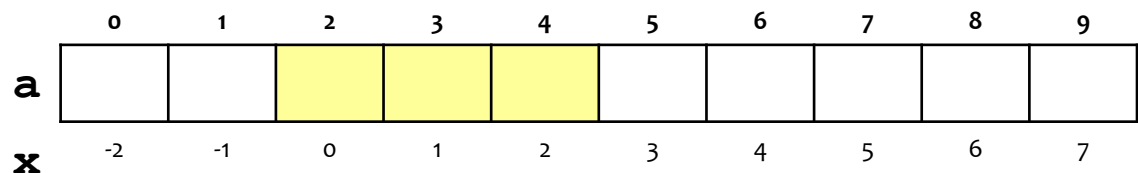
- Операторот `[]` може да се употреби и со покажувачи

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

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

```
for (int i=0; i<3; i++)
```

```
    x[i]++;
```



Полиња и покажувачи

- Следните два изрази се еквивалентни $a[i]$ и $*(a+i)$ и овозможуваат пристап до елементот од полето на позиција i
- пример, нека се декларирани вектор $a[5]$ и покажувач $aPtr$. Следните наредби се точни:

```
aPtr = a;
```

```
aPtr = &a[0];
```

```
a[n] == *( aPtr + n )
```

```
a[3] == *(a + 3)
```

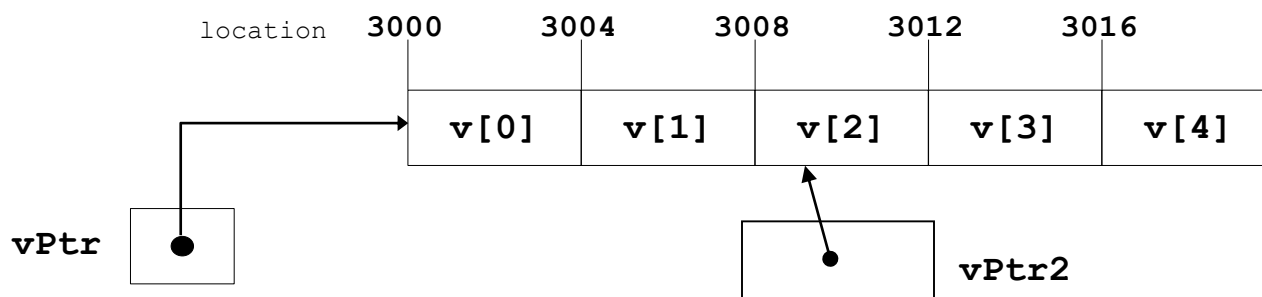
```
a+i == &a[i]
```

```
*(a+i) == a[i] == i[a]
```

- Пристапот до елемент на поле преку неговото име и индекс $a[i]$ преведувачот секогаш интерно го интерпретира како $*(a+i)$, така што на пример, наместо $a[5]$ сосема правилно ќе работи и $5[a]$!!!

Полиња и покажувачи

- Над покажувачите може да се примени целобројна аритметика. Ако покажувачот се зголеми за еден, неговата вредност ќе се зголеми за големината на објектот кон кој покажува
- За вектор со 5 целобројни променливи **int v[5]** и покажувачи **int *vPtr, *vPtr2** важи
 - vPtr2 = &v[2];** → **vPtr2** покажува кон елементот **v[2]**
 - vPtr = &v[0];** → **vPtr** покажува кон првиот елемент **v[0]** на локација 3000. (**vPtr = 3000**)
 - vPtr2 - vPtr == 2** → бројот на елементи меѓу двата покажувачи
 - vPtr += 2;** → го поставува **vPtr** на 3008, **vPtr** покажува на **v[2]** (зголемен е за две мемориски локации)



Полиња и покажувачи

- пример: Нека важат следните декларации

```
char a[50], x, y, *pa, *pa1, *pai;
```

```
pa = &a[0]; - адресата на a[0] смести ја во pa
```

```
pa = a; - исто како и претходното
```

```
x = *pa; - вредноста на a[0] смести ја во x
```

```
pa1 = pa+1; - определи ја адресата на a[1]
```

```
pai = pa+i; - определи ја адресата на a[i]
```

```
y = *(pa+i); - вредноста на a[i] во y
```

Илустрација на користење на покажувачи

```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
Напомени:
a[2] == *(a + 2)
ptr == &(a[2])
      == &(* (a + 2))
      ==      a + 2
```

Податочна табела			
Име	Тип	Опис	Вредност
a[0]	float	елемент на вектор (променлива)	
a[1]	float	елемент на вектор (променлива)	
a[2]	float	елемент на вектор (променлива)	
a[3]	float	елемент на вектор (променлива)	
ptr	float *	покажувач кон реална променлива	
*ptr	float	променлива кон која покажува покажувачот	

Илустрација на користење на покажувачи

```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
Напомени:
a[2] == *(a + 2)
ptr == &(a[2])
      == &(* (a + 2))
      ==      a + 2
```

Податочна табела			
Име	Тип	Опис	Вредност
a[0]	float	елемент на вектор (променлива)	?
a[1]	float	елемент на вектор (променлива)	?
a[2]	float	елемент на вектор (променлива)	?
a[3]	float	елемент на вектор (променлива)	?
ptr	float *	покажувач кон реална променлива	
*ptr	float	променлива кон која покажува покажувачот	

Илустрација на користење на покажувачи

```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
Напомени:
a[2] == *(a + 2)
ptr == &(a[2])
      == &(* (a + 2))
      ==      a + 2
```

Податочна табела			
Име	Тип	Опис	Вредност
a[0]	float	елемент на вектор (променлива)	?
a[1]	float	елемент на вектор (променлива)	?
a[2]	float	елемент на вектор (променлива)	?
a[3]	float	елемент на вектор (променлива)	?
ptr	float *	покажувач кон реална променлива	?
*ptr	float	променлива кон која покажува покажувачот	

Илустрација на користење на покажувачи

```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```

Напомени:

```
a[2] == *(a + 2)
ptr == &(a[2])
      == &(* (a + 2))
      == a + 2
```

Податочна табела			
Име	Тип	Опис	Вредност
a[0]	float	елемент на вектор (променлива)	?
a[1]	float	елемент на вектор (променлива)	?
a[2]	float	елемент на вектор (променлива)	?
a[3]	float	елемент на вектор (променлива)	?
ptr	float *	покажувач кон реална променлива	?
*ptr	float	променлива кон која покажува покажувачот	

Илустрација на користење на покажувачи

```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```

Напомени:

```
a[2] == *(a + 2)
ptr == &(a[2])
      == &(* (a + 2))
      == a + 2
```

Податочна табела			
Име	Тип	Опис	Вредност
a[0]	float	елемент на вектор (променлива)	?
a[1]	float	елемент на вектор (променлива)	?
a[2]	float	елемент на вектор (променлива)	3.14
a[3]	float	елемент на вектор (променлива)	?
ptr	float *	покажувач кон реална променлива	?
*ptr	float	променлива кон која покажува покажувачот	3.14

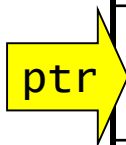
Илустрација на користење на покажувачи

```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```

Напомени:

```
a[2] == *(a + 2)
ptr == &(a[2])
      == &(* (a + 2))
      == a + 2
```

Податочна табела			
Име	Тип	Опис	Вредност
a[0]	float	елемент на вектор (променлива)	?
a[1]	float	елемент на вектор (променлива)	?
a[2]	float	елемент на вектор (променлива)	3.14
a[3]	float	елемент на вектор (променлива)	?
ptr	float *	покажувач кон реална променлива	?
*ptr	float	променлива кон која покажува покажувачот	3.14

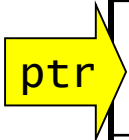


Илустрација на користење на покажувачи

```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```

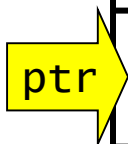
Напомени:

```
a[2] == *(a + 2)
ptr == &(a[2])
      == &(* (a + 2))
      == a + 2
```

Податочна табела			
Име	Тип	Опис	Вредност
a[0]	float	елемент на вектор (променлива)	?
a[1]	float	елемент на вектор (променлива)	?
a[2]	float	елемент на вектор (променлива)	3.14
 a[3]	float	елемент на вектор (променлива)	9.0
ptr	float *	покажувач кон реална променлива	?
*ptr	float	променлива кон која покажува покажувачот	9.0

Илустрација на користење на покажувачи

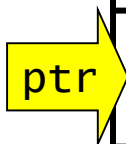
```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
Напомени:
a[2] == *(a + 2)
ptr == &(a[2])
      == &(* (a + 2))
      == a + 2
```



Податочна табела			
Име	Тип	Опис	Вредност
a[0]	float	елемент на вектор (променлива)	?
a[1]	float	елемент на вектор (променлива)	?
a[2]	float	елемент на вектор (променлива)	3.14
a[3]	float	елемент на вектор (променлива)	9.0
ptr	float *	покажувач кон реална променлива	?
*ptr	float	променлива кон која покажува покажувачот	9.0

Илустрација на користење на покажувачи

```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
Напомени:
a[2] == *(a + 2)
ptr == &(a[2])
      == &(* (a + 2))
      == a + 2
```



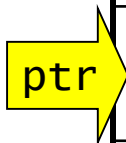
Податочна табела			
Име	Тип	Опис	Вредност
a[0]	float	елемент на вектор (променлива)	6.0
a[1]	float	елемент на вектор (променлива)	?
a[2]	float	елемент на вектор (променлива)	3.14
a[3]	float	елемент на вектор (променлива)	9.0
ptr	float *	покажувач кон реална променлива	?
*ptr	float	променлива кон која покажува покажувачот	6.0

Илустрација на користење на покажувачи

```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```

Напомени:

```
a[2] == *(a + 2)
ptr == &(a[2])
      == &(* (a + 2))
      == a + 2
```



Податочна табела			
Име	Тип	Опис	Вредност
a[0]	float	елемент на вектор (променлива)	6.0
a[1]	float	елемент на вектор (променлива)	?
a[2]	float	елемент на вектор (променлива)	3.14
a[3]	float	елемент на вектор (променлива)	9.0
ptr	float *	покажувач кон реална променлива	?
*ptr	float	променлива кон која покажува покажувачот	6.0

Илустрација на користење на покажувачи

```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```

Напомени:

```
a[2] == *(a + 2)
ptr == &(a[2])
      == &(* (a + 2))
      == a + 2
```

Податочна табела			
Име	Тип	Опис	Вредност
a[0]	float	елемент на вектор (променлива)	6.0
a[1]	float	елемент на вектор (променлива)	?
a[2]	float	елемент на вектор (променлива)	7.0
a[3]	float	елемент на вектор (променлива)	9.0
ptr	float *	покажувач кон реална променлива	?
*ptr	float	променлива кон која покажува покажувачот	7.0

Пренесување на променливи

- Пренесување на променливи се прави со покажувачи
 - се пренесува адресата на аргументот со & операторот
 - Овозможува во функцијата да се измени содржината на аргументот
 - полињата во функциите се пренесуваат како покажувачи

```
void double(int *number) {  
    *number = 2 * (*number);  
}
```

Пренесување на аргументи на функција преку покажувачи

```
void main(void) {
    int x,y;
    x = 5;      x 5   y 6
    y = 6;
    swap(x,y);
    printf("%d %d\n",x,y);
}
```

```
void swap(int a,int b){
    int temp;  a     b  
    temp = a;
    a = b;
    b = temp;
}
```

5 6

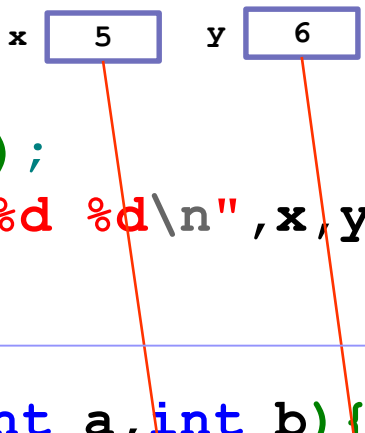
```
void main(void) {
    int x,y;
    x = 5;      x 100 5   y 200 6
    y = 6;
    swap(&x, &y);
    printf("%d %d\n",x,y);
}
```

```
void swap(int *a,int *b){
    int temp;  a     b  
    temp = *a;
    *a = *b;
    *b = temp;
}
```

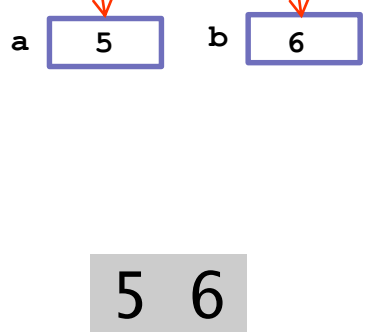
6 5

Пренесување на аргументи на функција преку покажувачи

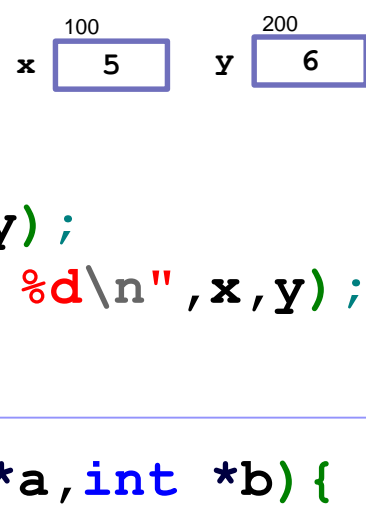
```
void main(void) {
    int x,y;
    x = 5;
    y = 6;
    swap(x,y);
    printf("%d %d\n",x,y);
}
```



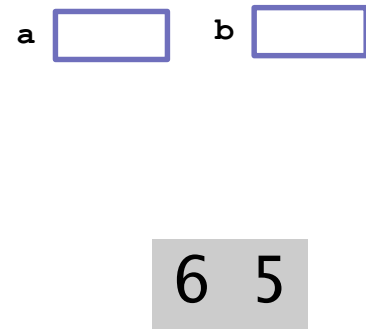
```
void swap(int a,int b){
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```



```
void main(void) {
    int x,y;
    x = 5;
    y = 6;
    swap(&x, &y);
    printf("%d %d\n",x,y);
}
```

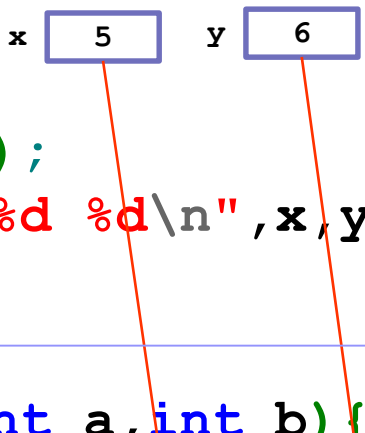


```
void swap(int *a,int *b){
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

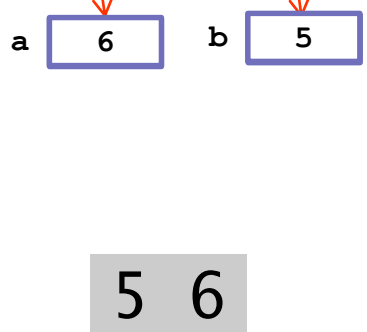


Пренесување на аргументи на функција преку покажувачи

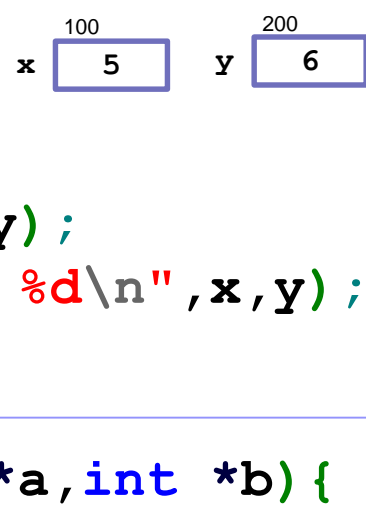
```
void main(void) {
    int x,y;
    x = 5;
    y = 6;
    swap(x,y);
    printf("%d %d\n",x,y);
}
```



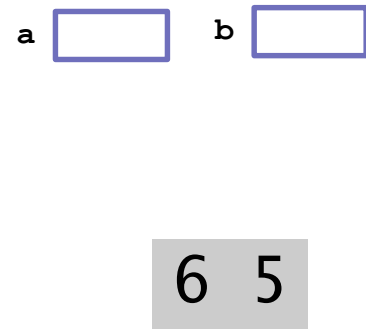
```
void swap(int a,int b){
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```



```
void main(void) {
    int x,y;
    x = 5;
    y = 6;
    swap(&x, &y);
    printf("%d %d\n",x,y);
}
```



```
void swap(int *a,int *b){
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```



Пренесување на аргументи на функција преку покажувачи

```
void main(void) {
    int x,y;
    x = 5;      x 5   y 6
    y = 6;
    swap(x,y);
    printf("%d %d\n",x,y);
}
```

```
void swap(int a,int b){
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```

5 6

```
void main(void) {
    int x,y;
    x = 5;      x 5   y 6
    y = 6;
    swap(&x, &y);
    printf("%d %d\n",x,y);
}
```

```
void swap(int *a,int *b){
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

6 5

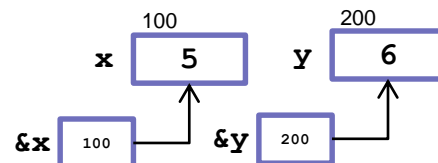
Пренесување на аргументи на функција преку покажувачи

```
void main(void) {
    int x,y;
    x = 5;    x 5    y 6
    y = 6;
    swap(x,y);
    printf("%d %d\n",x,y);
}
```

```
void swap(int a,int b){
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```

5 6

```
void main(void) {
    int x,y;
    x = 5;
    y = 6;
    swap(&x, &y);
    printf("%d %d\n",x,y);
}
```



```
void swap(int *a,int *b){
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

6 5

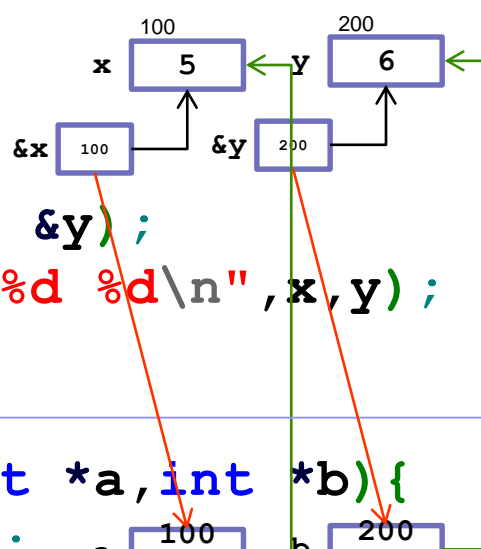
Пренесување на аргументи на функција преку покажувачи

```
void main(void) {
    int x,y;
    x = 5;    x 5    y 6
    y = 6;
    swap(x,y);
    printf("%d %d\n",x,y);
}
```

```
void swap(int a,int b){
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```

5 6

```
void main(void) {
    int x,y;
    x = 5;
    y = 6;
    swap(&x, &y);
    printf("%d %d\n",x,y);
}
```



```
void swap(int *a,int *b){
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

6 5

Пренесување на аргументи на функција преку покажувачи

```
void main(void) {
    int x,y;
    x = 5;    x 5    y 6
    y = 6;
    swap(x,y);
    printf("%d %d\n",x,y);
}
```

```
void swap(int a,int b){
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```

5 6

```
void main(void) {
    int x,y;
    x = 5;    x 5    y 6
    y = 6;
    swap(&x, &y);
    printf("%d %d\n",x,y);
}
```

```
void swap(int *a,int *b){
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

6 5

Пренесување на аргументи на функција преку покажувачи

```
void main(void) {
    int x,y;
    x = 5;    x 5    y 6
    y = 6;
    swap(x,y);
    printf("%d %d\n",x,y);
}
```

```
void swap(int a,int b){
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```

5 6

```
void main(void) {
    int x,y;
    x 6    y 5
    y = 6;
    swap(&x, &y);
    printf("%d %d\n",x,y);
}
```

```
void swap(int *a,int *b){
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

6 5

Пренесување на аргументи на функција преку покажувачи

```
void main(void) {
    int x,y;
    x = 5;      x 5   y 6
    y = 6;
    swap(x,y);
    printf("%d %d\n",x,y);
}
```

```
void swap(int a,int b){
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```

5 6

```
void main(void) {
    int x,y;
    x = 5;      x 6   y 5
    y = 6;
    swap(&x, &y);
    printf("%d %d\n",x,y);
}
```

```
void swap(int *a,int *b){
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

6 5

Пренесување на полиња во функции - дефинирање

- Декларирање на формален параметар поле во заглавје на функција
tipFunkcija Imefunkcija(tipElement ImePole[], int Indeks){...}

- вообичаено бројот на елементи во полето исто така се пренесува како аргумент

- пример:

```
void funkcija(int tpole[ ], int indeks /* broj na
elementi vo vektorot */) {
    tpole[indeks-1] = 0;
}
```

- Функцииски прототипови

```
void modifyArray( int b[], int arraySize );
```

- бидејќи името на формалните параметри не е важно претходната наредба може да гласи

```
void modifyArray( int [], int );
```

Пренесување на полиња во функции

- Повикување на функција со аргумент поле
 - Се наведува само името на полето (без [])
int myArray[24];
myFunction(myArray, 24);
- преносот на променлива поле се реализира со пренесување на адресата на првиот елемент, поради што сите промени на вредностите на полето во функцијата се видливи и по завршувањето на функцијата
- Пренесување на елементи на полето
 - може да се пренесат како вредности
 - во листата на аргументи се наведува името на полето и индексот на елементот (**myArray[3]**)
 - ако елементот на полето е пренесен како вредност сите промени на вредноста на формалниот аргумент во функцијата нема да се рефлектираат на аргументот со кој е повикана функцијата

Пример за пренесување на вектори

Следните функции извршуваат иста работа:

```
int clear1(int array[], int size){
    int i;
    for(i = 0; i<size; i++) array[i] = 0;
    return 0;
}
```

```
int clear2(int *array, int size){
    int *p;
    for(p=&array[0]; p<&array[size];p++) *p=0;
    return 0;
}
```

```
int clear3(int *array, int size){
    int i;
    for(i = 0; i < size; i++) array[i] = 0;
    return 0;
}
```

```
int clear4(int array[], int size) {
    int *p;
    for(p=&array[0]; p<&array[size];p++) *p=0;
    return 0;
}
```

При преносот како аргумент на ф-ја

int a[] и **int *a** се однесуваат потполно исто.

Во главната програма (каде се декларирани) разликата е:

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

☺ **pa[5]=...** е исто со **a[5]=...**

☺ **pa=...** или **pa++** е ОК,

Но **a** е **const** и може само да се чита,

☹ **a = ...** или **a++** не може!

Ако направам **pa++** тогаш ќе стане

☹ **pa[0]==a[1],**

☹ **pa[-1]==a[0] !!!**

Пример за пренесување на вектори

Следните функции извршуваат иста работа:

```
int clear1(int array[], int size){
```

```
    int i;
```

```
    for(i = 0; i < size; i++) array[i] = 0;
```

При преносот како аргумент на ф-ја

```
int clear1(int array[], int size){
    int i;
    for(i = 0; i < size; i++) array[i] = 0;
    return 0;
}
```

```
int i;
```

```
for(i = 0; i < size; i++) array[i] = 0;
```

```
return 0;
```

```
}
```

```
int clear4(int array[], int size) {
```

```
    int *p;
```

```
    for(p=&array[0]; p<&array[size];p++) *p=0;
```

```
    return 0;
```

```
}
```



a = ... или **a++** не може!

Ако направам **pa++** тогаш ќе стане



pa[0]==a[1],



pa[-1]==a[0] !!!

Пример за пренесување на вектори

Следните функции извршуваат иста работа:

```
int clear1(int array[], int size){
    int i;
    for(i = 0; i<size; i++) array[i] = 0;
    return 0;
}
```

При преносот како аргумент на ф-ја

int a[] и **int *a** се однесуваат потполно исто.

Во главната програма (каде се декларирани) разликата

```
int clear2(int *array, int size){
    int *p;
    for(p=&array[0]; p<&array[size];p++)
        *p=0;
    return 0;
}
```

```
int clear4(int array[], int size) {
    int *p;
    for(p=&array[0]; p<&array[size];p++) *p=0;
    return 0;
}
```



pa[-1]==a[0] !!!



Пример за пренесување на вектори

Следните функции извршуваат иста работа:

```
int clear1(int array[], int size){
    int i;
    for(i = 0; i<size; i++) array[i] = 0;
    return 0;
}
```

При преносот како аргумент на ф-ја

int a[] и **int *a** се однесуваат потполно исто.

Во главната програма (каде се декларирани) разликата е:

```
int clear2(int *array, int size){
```

```
int clear3(int *array, int size){
    int i;
    for(i = 0; i < size; i++) array[i] = 0;
    return 0;
}
```

```
int clear4(int array[], int size) {
    int *p;
    for(p=&array[0]; p<&array[size];p++) *p=0;
    return 0;
}
```



pa[-1]==a[0] !!!

Пример за пренесување на вектори

Следните функции извршуваат иста работа:

```
int clear1(int array[], int size){
    int i;
    for(i = 0; i<size; i++) array[i] = 0;
    return 0;
}
```

```
int clear2(int *array, int size){
    int *p;
```

```
int clear4(int array[], int size) {
    int *p;
    for(p=&array[0]; p<&array[size];p++)
        *p=0;
    return 0;
}
```

При преносот како аргумент на ф-ја

int a[] и **int *a** се однесуваат потполно исто.

Во главната програма (каде се декларирани) разликата е:

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

☺

```
pa[5]=... е исто со a[5]=...
```

Пример за пренесување на вектори

Следните функции извршуваат иста работа:

```
int clear1(int array[], int size){
    int i;
    for(i = 0; i<size; i++) array[i] = 0;
    return 0;
}
```

```
int clear2(int *array, int size){
    int *p;
    for(p=&array[0]; p<&array[size];p++) *p=0;
    return 0;
}
```

```
int clear3(int *array, int size){
    int i;
    for(i = 0; i < size; i++) array[i] = 0;
    return 0;
}
```

```
int clear4(int array[], int size) {
    int *p;
    for(p=&array[0]; p<&array[size];p++) *p=0;
    return 0;
}
```

При преносот како аргумент на ф-ја

int a[] и **int *a** се однесуваат потполно исто.

Во главната програма (каде се декларирани) разликата е:

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

☺ **pa[5]=...** е исто со **a[5]=...**

☺ **pa=...** или **pa++** е ОК,

Но **a** е **const** и може само да се чита,

☹ **a = ...** или **a++** не може!

Ако направам **pa++** тогаш ќе стане

☹ **pa[0]==a[1],**

☹ **pa[-1]==a[0] !!!**

Пример: Пренесување на вектори со покажувачи

```
#define N 5
int find_largest(int *a, int n){
    int i, max;
    max = a[0];
    for (i = 1; i < n; i++){
        if (a[i] > max) max = a[i];
    }
    return max;
}
int main(){
    int b[N] = {8,9,1,4,7}, largest;
    largest = find_largest(&b[2], 3);
    return 0;
}
```

```
#define N 5
int find_largest(int *p, int n){
    int i, max;
    max = *p; /* i.e. max = p[0] */
    for (i = 1; i < n; i++){
        if (*(p+i) > max) max = *(p+i);
    }
    return max;
}
int main(){
    int b[N] = {8,9,1,4,7}, largest;
    largest = find_largest(b, N);
    return 0;
}
```

```
#define N 5
int find_largest(int *p, int n) {
    int i, max;
    max = *p;
    for (i = 1; i < n; i++){
        if (*p > max) max = *p;
        p++;
    }
    return max;
}
int main(){
    int b[N] = {8,9,1,4,7}, largest;
    largest = find_largest(&b[0], N);
    return 0;
}
```

Сите три програми решаваат ист проблем, ама излезот на една од нив е различен. Зошто? Функцијата за наоѓање максимум во една од програмите нема да работи коректно во сите случаи. Која и зошто?

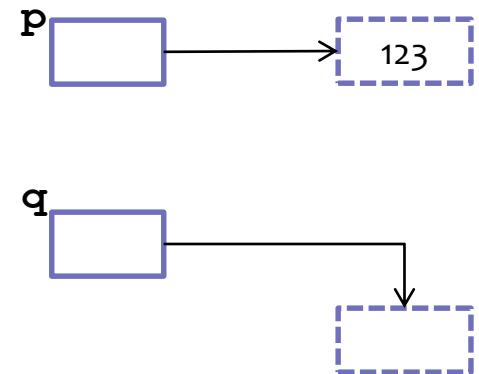
Динамичка алокација на меморија за променлива

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

int main()
{
    int *p, *q; /* deklariranje na pokazuvaci */
    p=malloc(sizeof(int)); /* alociraj memorija za 1 int
i nasoci go p kon nea */
    q=malloc(sizeof(int));
    *p=123;

    free(p); /* oslobodi ja alociranata memorija */

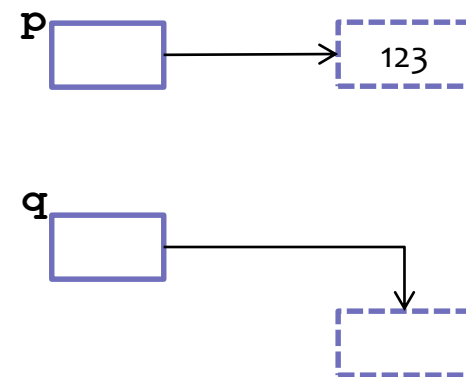
    free(q);
    return 0;
}
```



Динамичка алокација на меморија за променлива

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

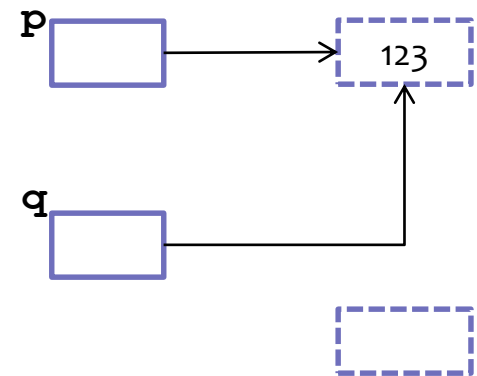
int main()
{
    int *p, *q; /* deklariranje na pokazuvaci */
    p=malloc(sizeof(int)); /* alociraj memorija za 1 int
i nasoci go p kon nea */
    q=malloc(sizeof(int));
    *p=123;
    q=p; /* memory leak! */
    printf("%d\n", *p) ;
    free(p); /* oslobodi ja alociranata memorija */
    *q=789; /* memory overwrite! */
    printf("%d\n", *q) ;
    free(q);
    return 0;
}
```



Динамичка алокација на меморија за променлива

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

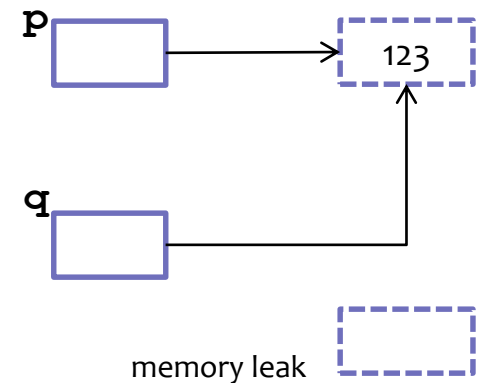
int main()
{
    int *p, *q; /* deklariranje na pokazuvaci */
    p=malloc(sizeof(int)); /* alociraj memorija za 1 int
i nasoci go p kon nea */
    q=malloc(sizeof(int));
    *p=123;
    q=p; /* memory leak! */
    printf("%d\n", *p) ;
    free(p); /* oslobodi ja alociranata memorija */
    *q=789; /* memory overwrite! */
    printf("%d\n", *q) ;
    free(q);
    return 0;
}
```



Динамичка алокација на меморија за променлива

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

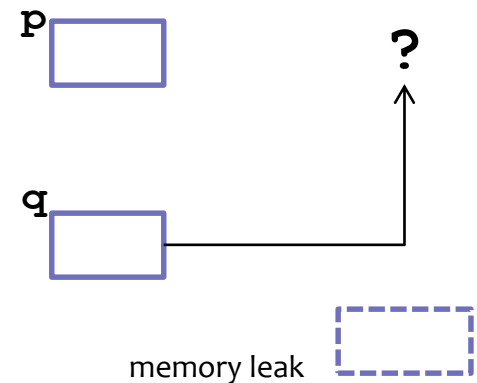
int main()
{
    int *p, *q; /* deklariranje na pokazuvaci */
    p=malloc(sizeof(int)); /* alociraj memorija za 1 int
i nasoci go p kon nea */
    q=malloc(sizeof(int));
    *p=123;
    q=p; /* memory leak! */
    printf("%d\n", *p) ;
    free(p); /* oslobodi ja alociranata memorija */
    *q=789; /* memory overwrite! */
    printf("%d\n", *q) ;
    free(q);
    return 0;
}
```



Динамичка алокација на меморија за променлива

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

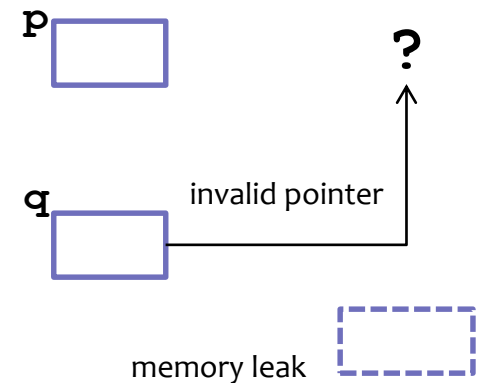
int main()
{
    int *p, *q; /* deklariranje na pokazuvaci */
    p=malloc(sizeof(int)); /* alociraj memorija za 1 int
i nasoci go p kon nea */
    q=malloc(sizeof(int));
    *p=123;
    q=p; /* memory leak! */
    printf("%d\n", *p) ;
    free(p); /* oslobodi ja alociranata memorija */
    *q=789; /* memory overwrite! */
    printf("%d\n", *q) ;
    free(q);
    return 0;
}
```



Динамичка алокација на меморија за променлива

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

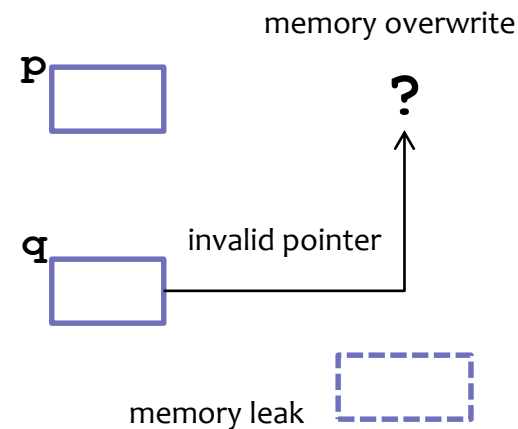
int main()
{
    int *p, *q; /* deklariranje na pokazuvaci */
    p=malloc(sizeof(int)); /* alociraj memorija za 1 int
i nasoci go p kon nea */
    q=malloc(sizeof(int));
    *p=123;
    q=p; /* memory leak! */
    printf("%d\n", *p) ;
    free(p); /* oslobodi ja alociranata memorija */
    *q=789; /* memory overwrite! */
    printf("%d\n", *q) ;
    free(q);
    return 0;
}
```



Динамичка алокација на меморија за променлива

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

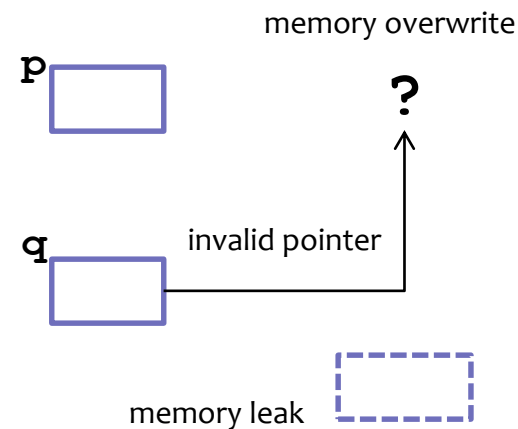
int main()
{
    int *p, *q; /* deklariranje na pokazuvaci */
    p=malloc(sizeof(int)); /* alociraj memorija za 1 int
i nasoci go p kon nea */
    q=malloc(sizeof(int));
    *p=123;
    q=p; /* memory leak! */
    printf("%d\n", *p) ;
    free(p); /* oslobodi ja alociranata memorija */
    *q=789; /* memory overwrite! */
    printf("%d\n", *q) ;
    free(q);
    return 0;
}
```



Динамичка алокација на меморија за променлива

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

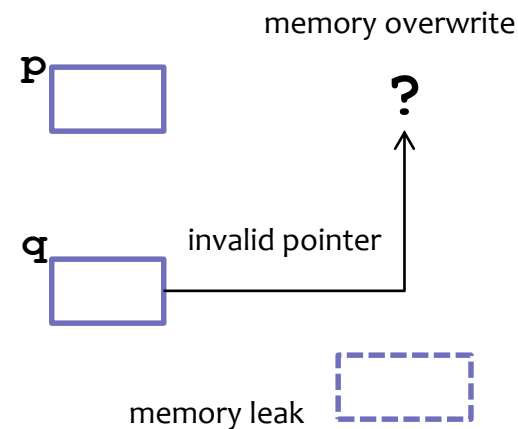
int main()
{
    int *p, *q; /* deklariranje na pokazuvaci */
    p=malloc(sizeof(int)); /* alociraj memorija za 1 int
i nasoci go p kon nea */
    q=malloc(sizeof(int));
    *p=123;
    q=p; /* memory leak! */
    printf("%d\n", *p) ;
    free(p); /* oslobodi ja alociranata memorija */
    *q=789; /* memory overwrite! */
    printf("%d\n", *q) ;
    free(q);
    return 0;
}
```



Динамичка алокација на меморија за променлива

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

int main()
{
    int *p, *q; /* deklariranje na pokazuvaci */
    p=malloc(sizeof(int)); /* alociraj memorija za 1 int
i nasoci go p kon nea */
    q=malloc(sizeof(int));
    *p=123;
    q=p; /* memory leak! */
    printf("%d\n", *p) ;
    free(p); /* oslobodi ja alociranata memorija */
    *q=789; /* memory overwrite! */
    printf("%d\n", *q) ;
    free(q);
    return 0;
}
```



Динамичка алокација на поле

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

Vnesete ja dolzinata na poletoto: 1000000000
Nema tolku memorija!

```
int main()
{
    int i, n, *x, *p;
    printf("Vnesete ja dolzinata na poletoto: ");
    scanf("%d",&n);
    x=malloc(sizeof(int)*n); /* memorija za n int promenlivi */
    if(!x) { printf("Nema tolku memorija!\n"); return -1; }
    for(p=x, i=0; i<n; )
        *p++=i++;
    for(i=0; i<n; i++, n--)
        x[i]=i+x[n-i-1];
    free(x); /* za da se oslobodi memorijata na free treba da se prosledi
pokazuvacot koj bil dobien od malloc pri alokacijata */
    return 0;
}
```

- Ако важи следната декларација `int a[100], *pa, *qa;`
- Што е погрешно во следните наредби?

<code>pa = a;</code>	<code>qa = pa;</code>
<code>a = pa;</code>	<code>*qa = *pa;</code>
<code>pa++;</code>	<code>pa = &a;</code>
<code>a++;</code>	<code>pa = *qa;</code>

- Ако важи следната декларација `int a[100], *pa, *qa;`
- Што е погрешно во следните наредби?

<code>pa = a;</code>	✓	<code>qa = pa;</code>
<code>a = pa;</code>		<code>*qa = *pa;</code>
<code>pa++;</code>		<code>pa = &a;</code>
<code>a++;</code>		<code>pa = *qa;</code>

- Ако важи следната декларација `int a[100], *pa, *qa;`
- Што е погрешно во следните наредби?

<code>pa = a;</code>	✓	<code>qa = pa;</code>
<code>a = pa;</code>	✗	<code>*qa = *pa;</code>
<code>pa++;</code>		<code>pa = &a;</code>
<code>a++;</code>		<code>pa = *qa;</code>

- Ако важи следната декларација `int a[100], *pa, *qa;`
- Што е погрешно во следните наредби?

<code>pa = a;</code>	✓	<code>qa = pa;</code>
<code>a = pa;</code>	✗	<code>*qa = *pa;</code>
<code>pa++;</code>	✓	<code>pa = &a;</code>
<code>a++;</code>		<code>pa = *qa;</code>

- Ако важи следната декларација `int a[100], *pa, *qa;`
- Што е погрешно во следните наредби?

<code>pa = a;</code>	✓	<code>qa = pa;</code>
<code>a = pa;</code>	✗	<code>*qa = *pa;</code>
<code>pa++;</code>	✓	<code>pa = &a;</code>
<code>a++;</code>	✗	<code>pa = *qa;</code>

- Ако важи следната декларација `int a[100], *pa, *qa;`
- Што е погрешно во следните наредби?

`pa = a;` ✓

`a = pa;` ✗

`pa++;` ✓

`a++;` ✗

`qa = pa;` ✓

`*qa = *pa;`

`pa = &a;`

`pa = *qa;`

- Ако важи следната декларација `int a[100], *pa, *qa;`
- Што е погрешно во следните наредби?

`pa = a;` ✓

`a = pa;` ✗

`pa++;` ✓

`a++;` ✗

`qa = pa;` ✓

`*qa = *pa;` ✓

`pa = &a;`

`pa = *qa;`

- Ако важи следната декларација `int a[100], *pa, *qa;`
- Што е погрешно во следните наредби?

`pa = a;` ✓

`a = pa;` ✗

`pa++;` ✓

`a++;` ✗

`qa = pa;` ✓

`*qa = *pa;` ✓

`pa = &a;` ✗

`pa = *qa;`

- Ако важи следната декларација `int a[100], *pa, *qa;`
- Што е погрешно во следните наредби?

`pa = a;` ✓

`a = pa;` ✗

`pa++;` ✓

`a++;` ✗

`qa = pa;` ✓

`*qa = *pa;` ✓

`pa = &a;` ✗

`pa = *qa;` ✗

Факултативно:

Покажувачи на функции

■ Покажувач на функција

- Покажува кон меморијата во која е сместен кодот на одредена функција
- Со негово дереференцирање се повикува функцијата кон која тој покажува

■ Декларирање и работа со покажувач на функција

```
int f1(...);  
int f2(...);  
int (*pf)(); /* deklaracija na pokazuvac na funkcija koja vrackja int */  
pf=&f1;  
(*pf)(...); /* call f1(...) */  
pf=&f2;  
(*pf)(...); /* call f2(...) */
```

■ Правете разлика

- **type (*pf) (...)** – pf е покажувач кон функција која враќа type
- **type *f (...)** – f е функција која враќа покажувач кон type

Покажувачи на функции

```
int main()
{
    float o1, o2, r;
    char c[2]; int dummy;
    float (*pfi)(float, float);

    printf("op1:"); scanf("%f", &o1);
    printf("op2:"); scanf("%f", &o2);
    printf("operation [+ - * / ^] ");
    scanf("%1s", c);
    switch(*c)
    {
        case '+': pfi=op1; break;
        case '-': if(o1>o2) pfi=op2; else pfi=opf; break;
        case '*': pfi=op3; break;
        case '/': if(o2!=0 && o1>o2) pfi=op4; else pfi=opf; break;
        case '^': if(o1==0 && o2==0 || o1<0 && modf(o2,&dummy)!=0)
        pfi=opf; else pfi=op5; break;
    }
    r=(*pfi)(o1,o2);
    printf("result = %f\n",r);
    return 0;
}
```

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

float opf(float x, float y) { return x; }
float op1(float x, float y) { return x + y; }
float op2(float x, float y) { return x - y; }
float op3(float x, float y) { return x * y; }
float op4(float x, float y) { return x / y; }
float op5(float x, float y) { return pow(x,y); }
```

Програмата чита два броја и потоа операнд и ја извршува операцијата, со дополнителни услови за валидноста на операцијата (ако операцијата „не е валидна“ се враќа првиот операнд како резултат).

`modf` – го разложува бројот на цел и децимален дел. Децималниот дел се враќа како резултат од функцијата, а целиот се сместува во вториот аргумент.

Функција со аргумент покажувач на друга функција

```
#include <stdio.h>
void young(int);
void old(int);
void greeting(void (*)(int), int);
int main(void) {
    int age;
    printf("Kolku godini imas? ");
    scanf("%d", &age);
    if (age > 30) {
        greeting(old, age);
    }
    else {
        greeting(young, age);
    }
    return 0;
}
void greeting(void (*fp)(int), int k) { fp(k); }
void young(int n) { printf("So samo %d godini ti si sekako mlad.\n", n); }
void old(int m) { printf("So %d godini, Vie ste sigurno star.\n", m); }
```

Bi Pointer Fun with Binky



by Nick Parlante

This is document 104 in the Stanford CS
Education Library — please see
cslibrary.stanford.edu
for this video, its associated documents,
and other free educational materials.

Copyright © 1999 Nick Parlante. See copyright
panel for redistribution terms.
Carpe Post Meridiem!



Прашања?