



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 2024

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)

Концепт на покажувач

■ Директно адресирање

- ☐ документот хуз е во 3-тата фасцикла на 3-тата полица
- ☐ предавањата по предметот abc се во предавална 123

■ Индиректно адресирање

- ☐ во првата фасцикла на првата полица има документ во кој е запишана локацијата на документот хуз
- ☐ на огласната табла во холот има распоред на кој е напишано во која предавална се предавањата по предметот abc

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;
```

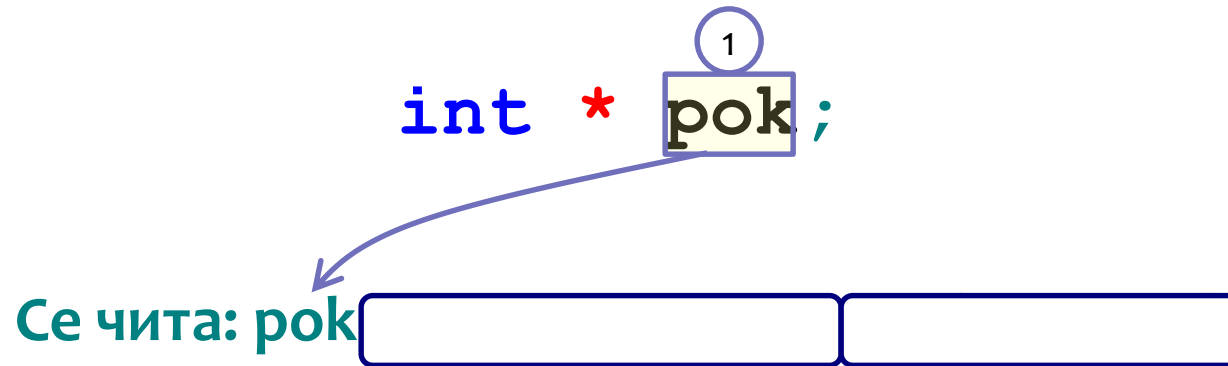
- Може да се декларираат покажувачи од кој и да е податочен тип, вклучувајќи и покажувач кон покажувач

Читање на декларација на покажувач

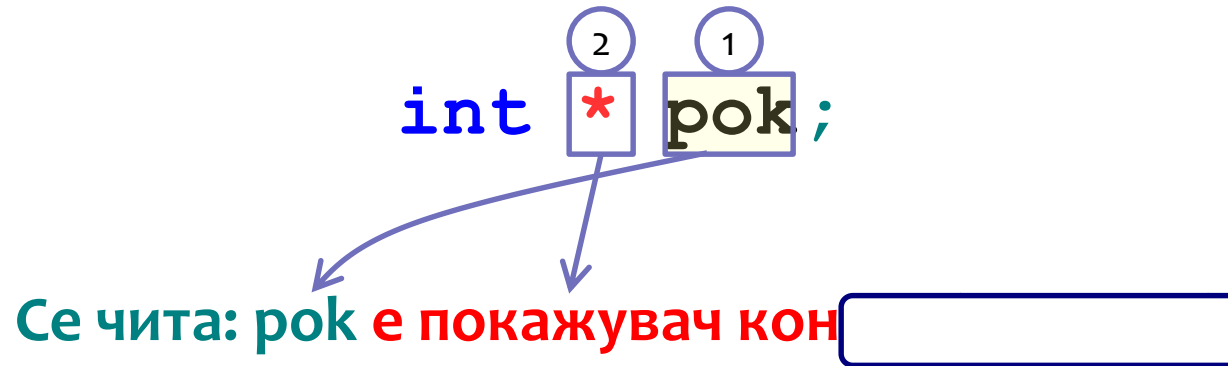
```
int * pok;
```

Се чита:

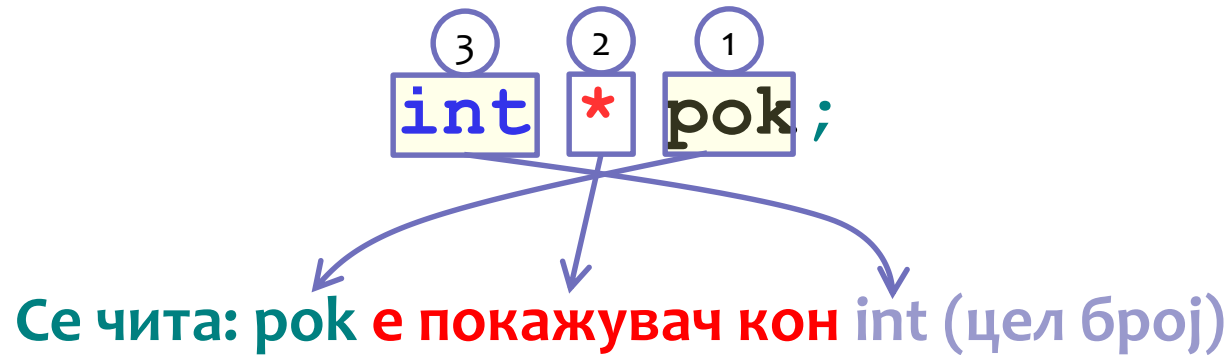
Читање на декларација на покажувач



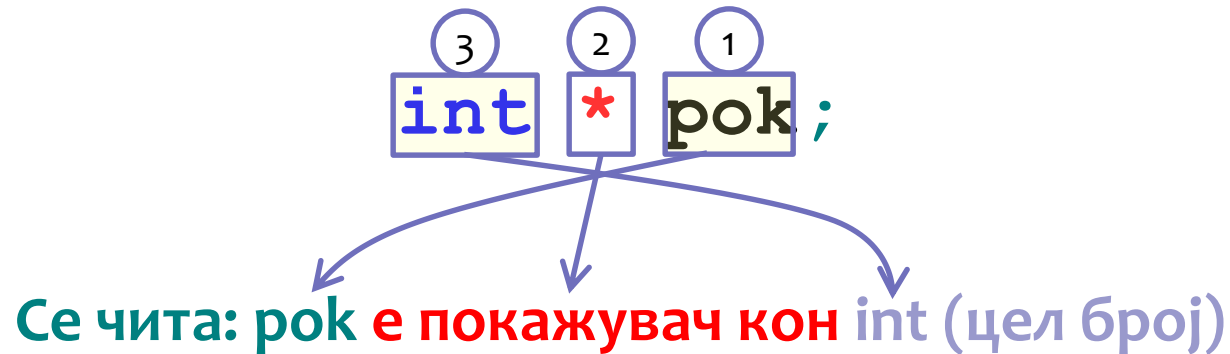
Читање на декларација на покажувач



Читање на декларација на покажувач

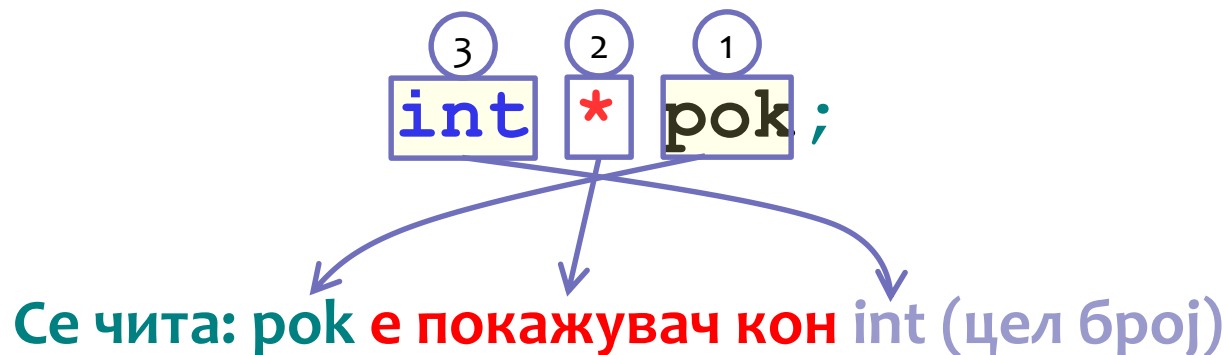


Читање на декларација на покажувач



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;

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)

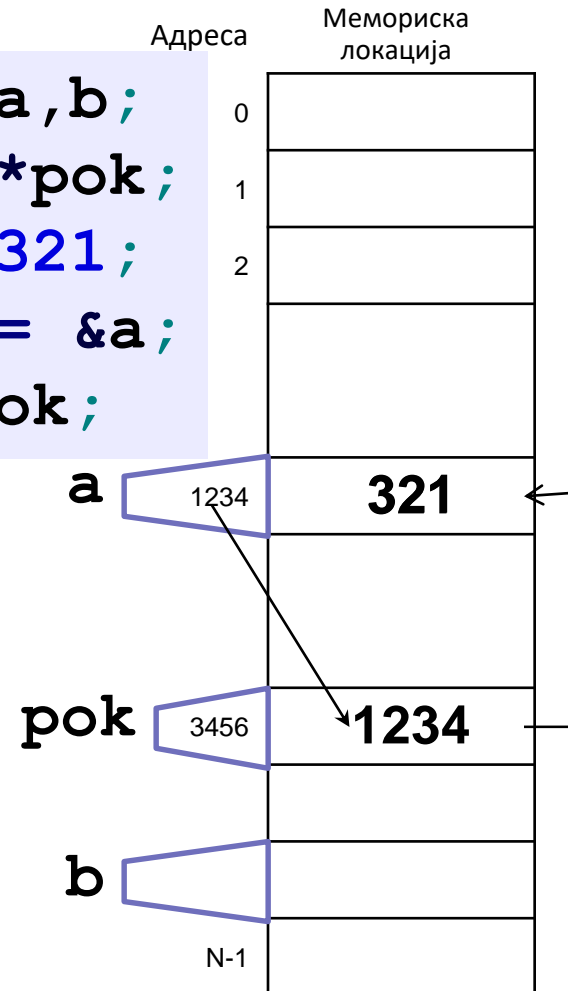
```
int a,b;
int *pok;
a = 321;
pok = &a;
b=*pok;
```

	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)

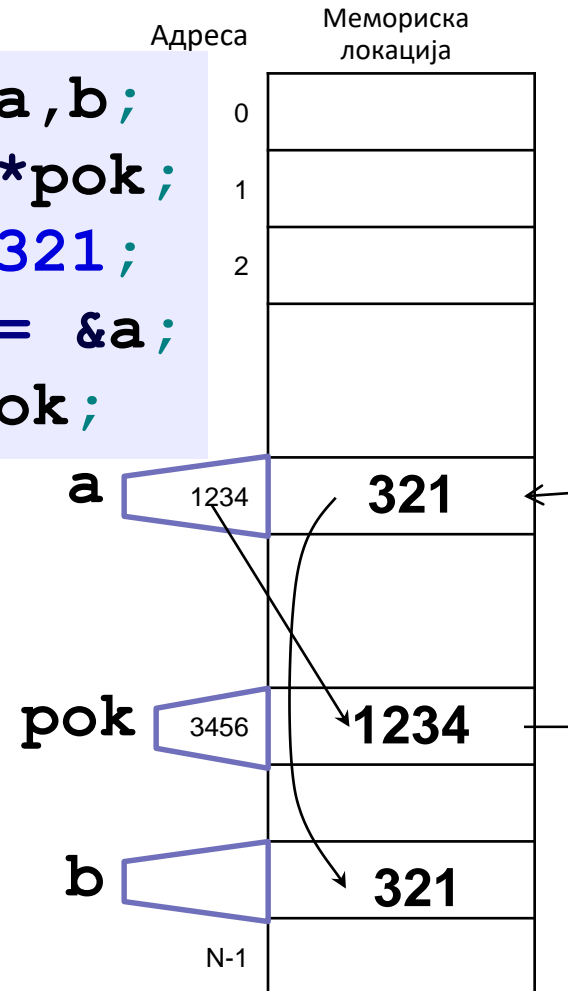
```
int a,b;
int *pok;
a = 321;
pok = &a;
b=*pok;
```



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 некоја променлива и `yptr=&y`, односно `yptr` покажува на `y`

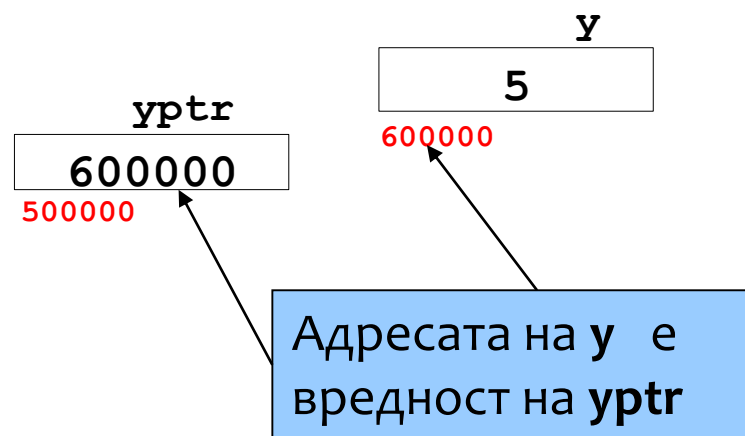
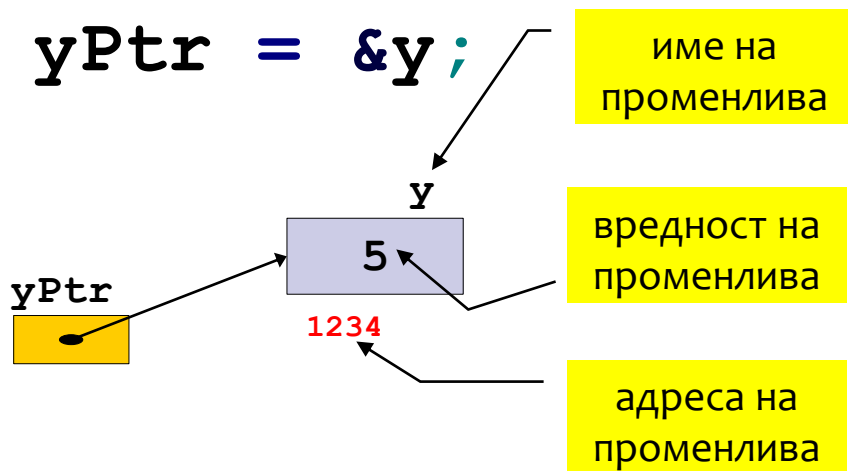
Иницијализација на покажувачи

- `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`, или `nullptr` по C++11) – 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;
cout << "i = " << i << endl;
cout << "*ptr = " << *ptr << endl;
cout << "ptr = " << ptr << endl;
```

Излез:

$i = 5$

$*ptr = 5$

$ptr = 000000162B11F6F4$

вредност на ptr =
адреса на i во
меморијата

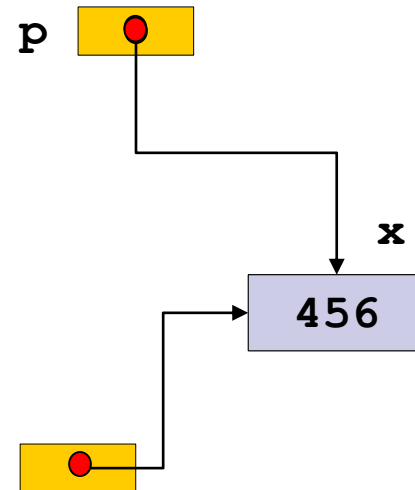
Работа со покажувачи

- Во даден момент покажувачот покажува кон единствена точно одредена мемориска локација, но во текот на програмата истиот покажувач може да биде променет и да биде наведен да покажува кон друга локација
- Кон иста променлива (мемориска локација) може да покажуваат повеќе покажувачи (во ист момент)

Пример (анимиран)

```
int main() {
    int x = 123;
    int *p = &x, *q;
```

```
    ➔ q = p;
    ➔ cout << "x = " << x << endl;
    ➔ cout << "*p = " << *p << endl;
    ➔ *q = 456;
    ➔ cout << "x = " << x << endl;
    return 0;
}
```



```
x = 123
*p = 123
x = 456
```

Пример

```
int main()
```

```
{
```

```
    int x = 123;
```

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

```
    q = p;
```

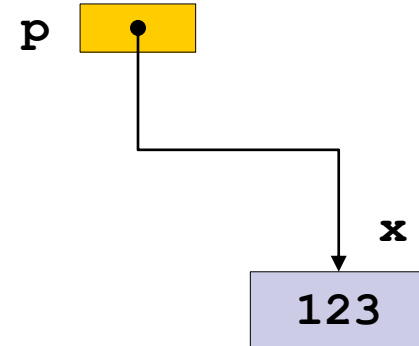
```
    cout << "x = " << x << endl;
```

```
    cout << "*p = " << *p << endl;
```

```
    *q = 456;
```

```
    cout << "x = " << x << endl;
```

```
}
```



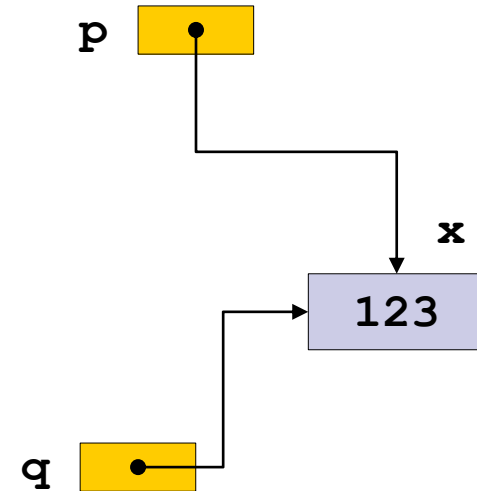
q

Пример

```
int main()
{
    int x = 123;
    int* p = &x, * q;
```

➔

```
    q = p;
    cout << "x = " << x << endl;
    cout << "*p = " << *p << endl;
    *q = 456;
    cout << "x = " << x << endl;
}
```

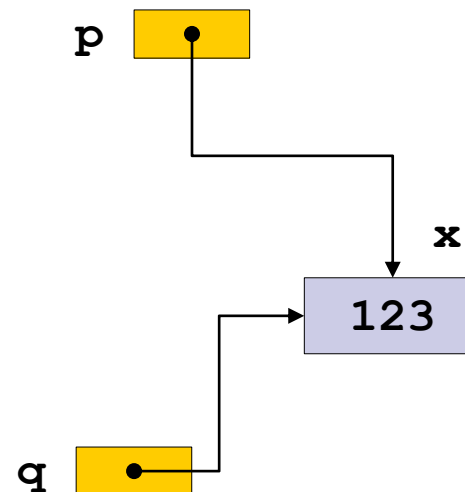


Пример

```
int main()  
{  
    int x = 123;  
    int* p = &x, * q;
```

```
    q = p;
```

```
    cout << "x = " << x << endl;  
    cout << "*p = " << *p << endl;  
    *q = 456;  
    cout << "x = " << x << endl;  
}
```

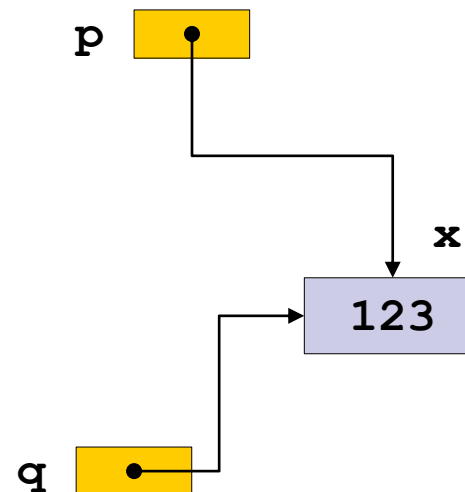


x = 123

Пример

```
int main()
{
    int x = 123;
    int* p = &x, * q;

    q = p;
    cout << "x = " << x << endl;
    ➔ cout << "*p = " << *p << endl;
    *q = 456;
    cout << "x = " << x << endl;
}
```

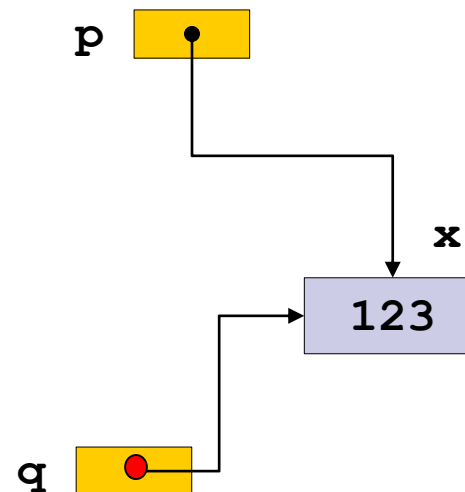


x = 123
*p = 123

Пример

```
int main()
{
    int x = 123;
    int* p = &x, * q;

    q = p;
    cout << "x = " << x << endl;
    cout << "*p = " << *p << endl;
    ➔ *q = 456;
    cout << "x = " << x << endl;
}
```

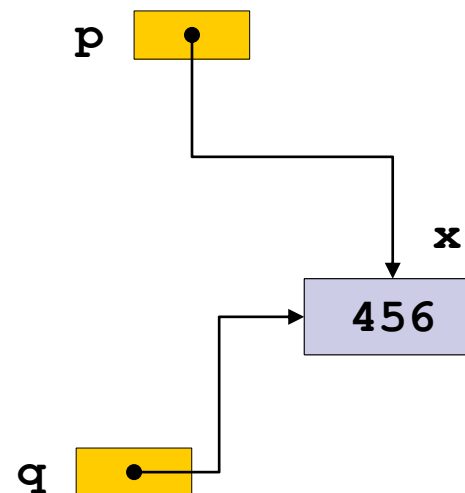


x = 123
*p = 123

Пример

```
int main()
{
    int x = 123;
    int* p = &x, * q;

    q = p;
    cout << "x = " << x << endl;
    cout << "*p = " << *p << endl;
    *q = 456;
    cout << "x = " << x << endl;
}
```

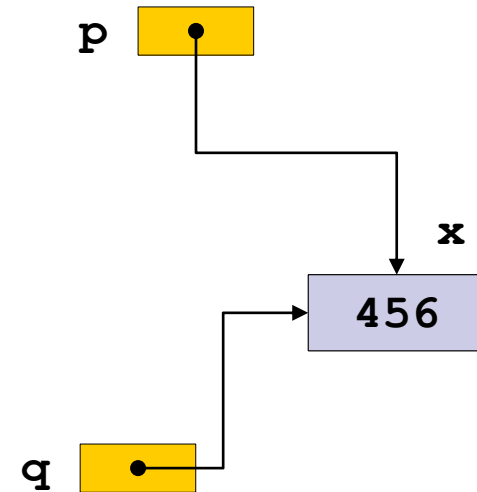


x = 123
*p = 123

Пример

```
int main()
{
    int x = 123;
    int* p = &x, * q;

    q = p;
    cout << "x = " << x << endl;
    cout << "*p = " << *p << endl;
    *q = 456;
    cout << "x = " << x << endl;
}
```



```
x = 123
*p = 123
x = 456
```

Операции со покажувачи

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

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.

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 .

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
 - **внимателно!** – со ова само се наложува покажувачот да покажува кон мемориска локација на која се чува вредност од друг тип – но оваа вредност не го менува типот!!!
 - 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
 - повеќето оператори не може да се користат над **void** покажувач (*****, **++**, **--**, **+=**, **-=**)

Доделување на вредност на покажувач

- Може да им се додели само адреса (вредност на друг покажувач од истиот тип)

КУСА ПРОВЕРКА

- Што значи $\text{ptr} + 1$?
- Што означува $\text{ptr} - 1$?
- Што означуваат $\text{ptr} * 2$ и $\text{ptr} / 2$?

множење и делење со
покажувачи не е дозволено!

Доделување на вредност на покажувач (анимиран)

```
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; 
```

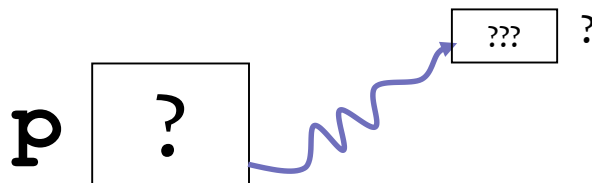
Дереференцирање на неиницијализиран покажувач

- Покажувачот мора да има вредност (да се насочи) 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;
```

Употреба на покажувачи

- За пренесување на аргументи на функции
- За пристап до елементи од низи
 - Пренос на низи 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*/
```

Низи и покажувачи

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

- Ако важи `int a[10]`, тогаш

- `a` е `int * const`

- `a` значи `&a[0]`

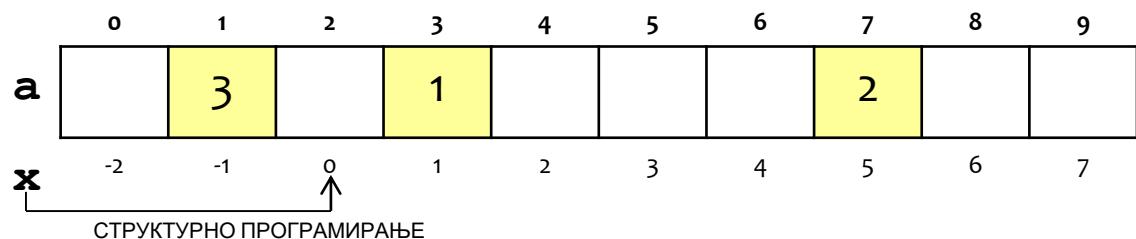
- Операторот `[]` може да се употреби и со покажувачи

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

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

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

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



Низи и покажувачи

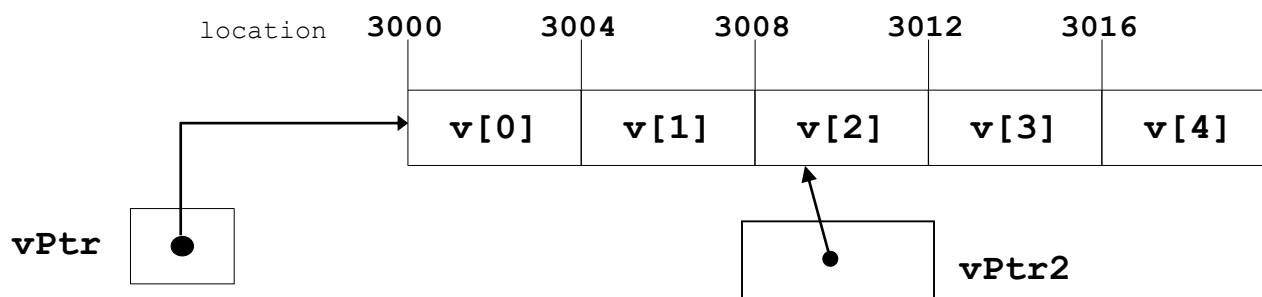
- Следните два изрази се еквивалентни $a[i]$ и $*(a+i)$ и овозможуваат пристап до елементот од низата на позиција i
- пример, нека се декларирани низа $a[5]$ и покажувач $aPtr$. Следните наредби се точни:

```
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 низа преку нејзиното 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 $5[a]$!!!

Низи 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 низа 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)



Низи 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 низа (променлива)	6.0
a[1]	float	елемент на низа (променлива)	?
a[2]	float	елемент на низа (променлива)	3.04
a[3]	float	елемент на низа (променлива)	9.0
ptr	float *	покажувач кон реална променлива	?
*ptr	float	променлива кон која покажува покажувачот	6.04

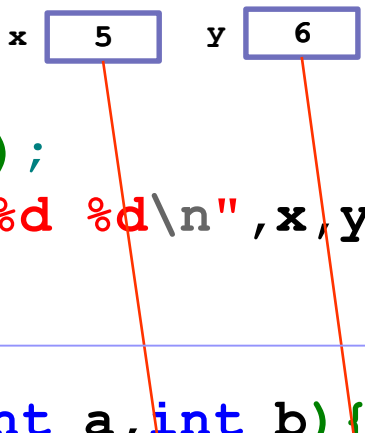
Пренесување на променливи

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

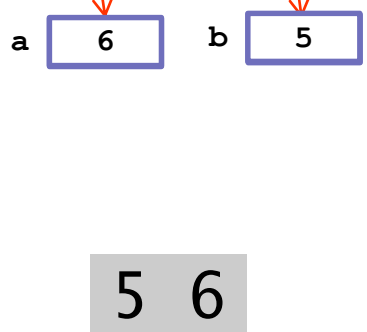
```
void dbl(int *number) {  
    *number = 2 * (*number);  
}
```

Пренесување на аргументи на функција преку покажувачи

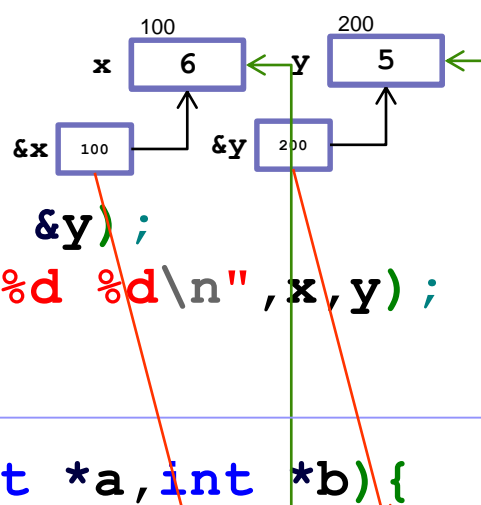
```
void main(void) {
    int x,y;
    x = 5;
    y = 6;
    swap(x,y);
    printf("%d %d\n",x,y);
}
```



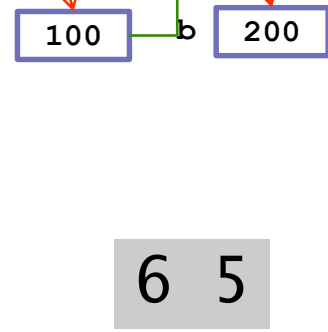
```
void swap(int a,int b){
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```



```
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;
}
```



Референци (&)

- Референците се специјален вид на променливи кои претставуваат алтернативно име (алиас) на веќе постоечка променлива. Откако референцата ќе биде дефинирана да референцира кон одредена променлива, кон таа променлива може да се пристапи и со нејзиното име и со референцата.

Референци наспроти покажувачи

- Референците не може да бидат неиницијализирани (NULL). Се подразбира дека референците секогаш се поврзани кон валидна мемориска локација.
- По креирањето референцата не може да ја промени променливата кон која референцира. Покажувачите може да се пренасочат кон друга локација во секое време.
- Референцата мора да се иницијализира при нејзиното креирање. Покажувачите може да се иницијализираат во кое било време.

Референци

```
#include <iostream>
using namespace std;
```

```
int main()
{
    int x = 123;
    int& y = x;
    cout << "x= " << x << " y= " << y << endl;
    x = 23;
    cout << "x= " << x << " y= " << y << endl;
    x++;
    cout << "x= " << x << " y= " << y << endl;
    y++;
    cout << "x= " << x << " y= " << y << endl;

    return 0;
}
```

```
x= 123 y= 123
x= 23  y= 23
x= 24  y= 24
x= 25  y= 25
```

Пренесување на аргументи на функција преку референци

```
int main(void) {
    int x, y;
    x = 5;      x 5   y 6
    y = 6;
    swap(x, y);
    cout <<x<<' ' <<y<< endl;
}
```

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

5 6

```
int main(void) {
    int x, y;
    x = 5;      x 6   y 5
    y = 6;
    swap(x, y);
    cout <<x<<' ' <<y<< endl;
}
```

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

6 5

Пренесување на низи во функции - дефинирање

- Декларирање на формален параметар низа во заглавје на функција **tipFunkcija lmefunkcija(tipElement lmeNiza[], int Indeks){...}**

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

- ☐ пример:

```
void funkcija(int tniza[ ], int indeks /* broj na
elementi vo vektorot */) {
    tniza[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 низи во функции

- Повикување на функција со аргумент низа

- Се наведува само името на низата (без [])

```
int myArray[24] ;
```

```
myFunction(myArray, 24 ) ;
```

- преносот на променлива низа се реализира со пренесување на адресата на првиот елемент, поради што сите промени на вредностите на низата во функцијата се видливи и по завршувањето на функцијата
- Пренесување на елементи на низата
 - може да се пренесат како вредности
 - во листата на аргументи се наведува името на низата и индексот на елементот (**myArray[3]**)
 - ако елементот на низата 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 низи

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 низи

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 низи

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 низи

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 низи

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 низи 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;
}
```

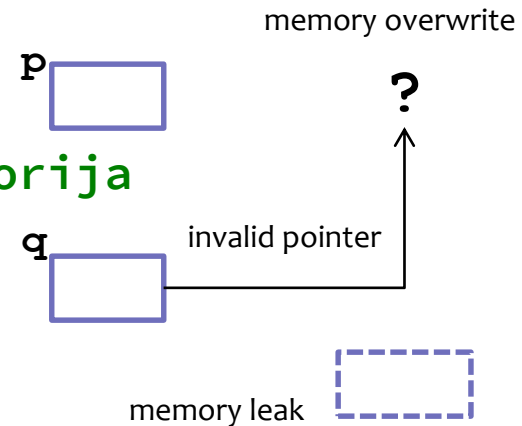
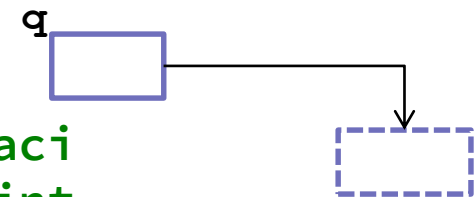
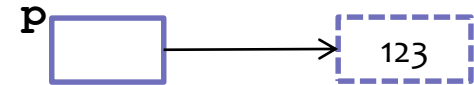
Сите три програми решаваат ист проблем, ама излезот на една од нив е различен. Зошто? Функцијата за наоѓање максимум во една од програмите нема да работи коректно во сите случаи. Која и зошто?

Динамичка алокација на меморија за променлива

```
#include <iostream>
using namespace std;
```

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

```
    *p = 123;
    q = p; // memory leak!
    cout << *p << endl;
    delete p; // oslobodi ja alociranata memorija
    *q = 789; // memory overwrite!
    cout << *q << endl;
    delete q;
}
```



- Ако важи следната декларација `int a[100], *pa, *qa;`
- Кои од следните наредби се погрешни?

`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]; float dummy;
    float (*pfi)(float, float);

    cout << "op1:"; cin >> o1;
    cout << "op2:"; cin >> o2;
    cout << "operation [+ - * / ^] "; cin >> 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;
        default:
            pfi = opf;
    }
    r = (*pfi)(o1, o2);
    cout << "result = " << r << endl;
}
```

```
#include <iostream>
#include <cmath>
using namespace std;
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 <iostream>
using namespace std;

void young(int);
void old(int);
void greeting(void (*)(int), int);
int main(void) {
    int age;
    cout << "Kolku godini imas? ";
    cin >> age;
    if (age > 30) {
        greeting(old, age);
    }
    else {
        greeting(young, age);
    }
}

void greeting(void (*fp)(int), int k) { fp(k); }
void young(int n) { cout << "So samo " << n << " godini ti si sekako mlad.\n"; }
void old(int m) { cout << "So " << m << " godini, Vie ste sigurno star.\n"; }
}
```

B

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?