



PROGRAMIRANJE I

P-08: Pokazivači

prof. dr **Dražen Brđanin**
2023/24



P-08: Pokazivači

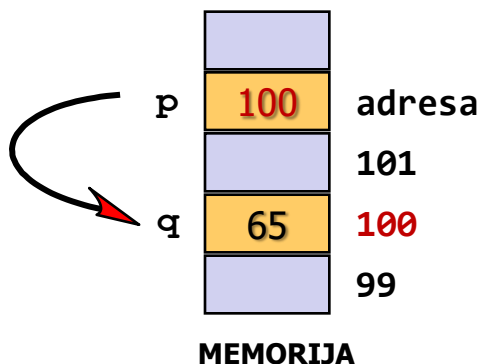
■ Sadržaj predavanja

- definicija pokazivača
- referenciranje i dereferenciranje
- pokazivačka aritmetika
- konstantni pokazivači
- višestruki pokazivači
- pokazivači i argumenti funkcije
- pokazivači i strukture
- pokazivači i nizovi
- dinamička alokacija memorije

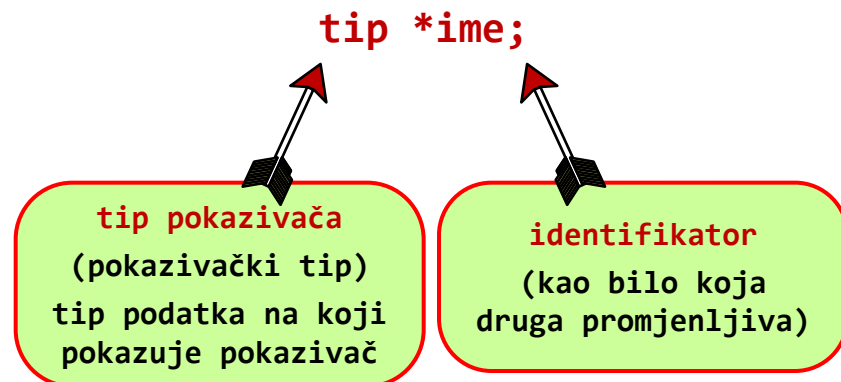
Definicija pokazivača

Pokazivači (Pointeri)

- Pokazivači obezbjeđuju indirektan pristup memoriji (**putem adrese**)
- Pokazivač je promjenljiva koja sadrži **memorijsku adresu nekog objekta** (npr. promjenljiva/konstanta prostog/složenog/korisnički definisanog tipa ili npr. funkcija)
- Ako promjenljiva *p* sadrži adresu objekta *q*, kažemo da *p* pokazuje na *q*



Definicija pokazivača:



Alternativni oblik definicije:

tip* ime;

Primjeri:

```
int *pi;  
char *pc, *tekst;
```

```
int *p, i; // p je pointer, i nije pointer  
int* p, i; // p je pointer, i nije pointer
```

Definicija pokazivača

VELIČINA POKAZIVAČA ZAVISI OD ARHITEKTURE SISTEMA I ISTA JE ZA SVE TIPOVE U SISTEMU!

➤ **Veličina pokazivača (broj bajtova koje pokazivač zauzima u memoriji) zavisi od konkretnog sistema i ista je za sve pokazivače, bez obzira na tip pokazivača (sve adrese su iste veličine):**

- u 16-bitnom sistemu, sve adrese su 16-bitne, tj. 2 bajta
- u 32-bitnom sistemu, sve adrese su 32-bitne, tj. 4 bajta
- u 64-bitnom sistemu, sve adrese su 64-bitne, tj. 8 bajtova

Primjer:

```
#include <stdio.h>
int main()
{
    char *pc;
    double *pd;
    printf("pc: %d\n", sizeof(pc));
    printf("pd: %d\n", sizeof(pd));
    return 0;
}
```

```
pc: 4
pd: 4
```

Primjer:

```
#include <stdio.h>
int main()
{
    printf("char* : %d\n", sizeof(char*));
    printf("int* : %d\n", sizeof(int*));
    printf("void* : %d\n", sizeof(void*));
    return 0;
}
```

```
char* : 4
int* : 4
void* : 4
```

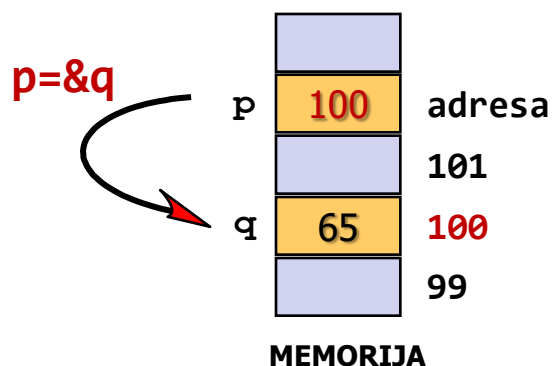
void*

“univerzalni” pokazivački tip – pokazivač ne pokazuje na neki konkretan objekat (jer ne postoje podaci tipa void), ali može da sadrži adresu bilo kakvog objekta

Referenciranje i dereferenciranje

Adresni operator (&)

- alternativni naziv: **operator referenciranja**
- adresni operator daje adresu nekog objekta u memoriji
 - ako je q neka promjenljiva tada $\&q$ daje adresu te promjenljive
 - ako želimo da pokazivač p pokazuje na promjenljivu q , tada pišemo $p = \&q$

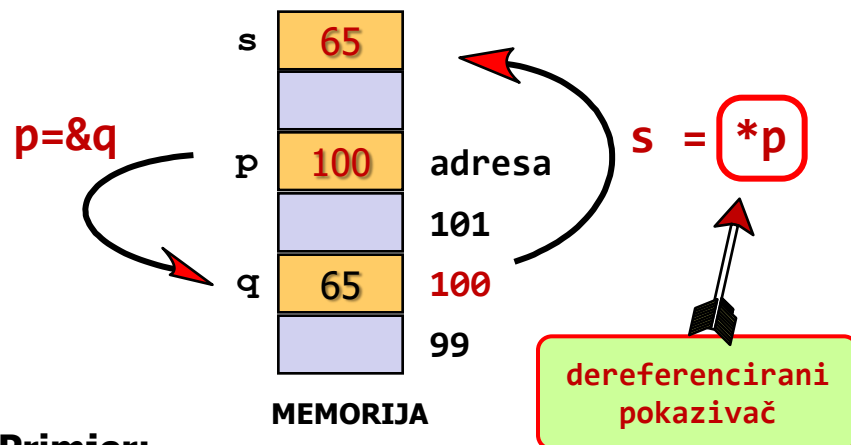


Primjer:

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

Operator indirekcije (*)

- alternativni naziv: **operator dereferenciranja**
- omogućava indirektan pristup promjenljivoj koristeći pokazivač na tu promjenljivu (alternativno se kaže da omogućava dereferenciranje pokazivača)
 - ako je s neka promjenljiva, a p pokazivač na isti tip, tada se može pisati $s = *p$ (s dobija vrijednost koja se nalazi na lokaciji koju pokazuje p)



Primjer:

```
int q=65, s, *p;  
p=&q; s=*p;  // isto kao: s=q
```

Referenciranje i dereferenciranje

Adresni operator i operator indirekcije su **inverzni operatori**

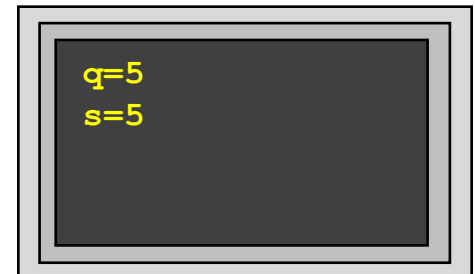
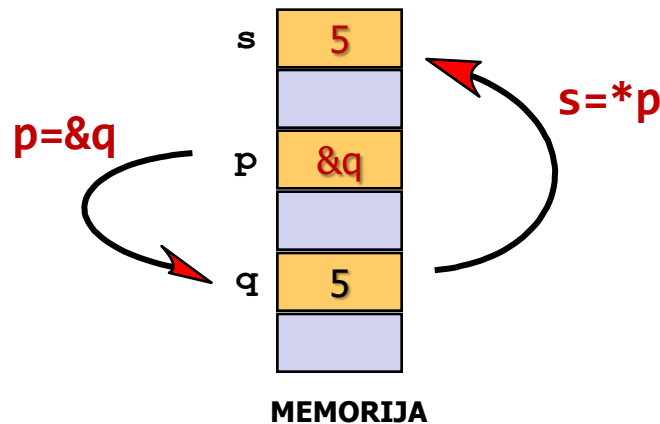
$s = \&q \Leftrightarrow s = q$

ili

```
int q,s, *p;
p = &q;
s = *p;
} s = *p = *&q = q
```

Primjer:

```
#include <stdio.h>
int main()
{
    int q=5, s, *p;
    p = &q;
    s = *p;
    printf("q=%d\n", q);
    printf("s=%d\n", s);
    return 0;
}
```



Referenciranje i dereferenciranje

- Iako je adresa cjelobrojna, **pokazivačima nije dozvoljeno dodjeljivati proizvoljne cjelobrojne vrijednosti**
- **Dozvoljeno je pokazivaču dodijeliti vrijednost 0 (konstanta NULL)**
- **Null-pokazivač** (pokazivač koji ima vrijednost nula) **nije moguće dereferencirati**
- **Pokušaj dereferenciranja null-pokazivača dovodi do greške u izvršavanju programa** (najčešće "segmentation fault") **i rezultuje prekidom izvršavanja programa**
- **Zbog toga bi trebalo prije svakog dereferenciranja nepoznatog pokazivača provjeriti da li je u pitanju null-pokazivač**

Primjer:

```
int *p=0;
```



```
int *p=NULL;
```

Ako pokazivaču dodijelimo vrijednost 0 (NULL), to ne znači da pokazivač pokazuje na adresu 0, nego da pokazivač ne pokazuje na neki konkretan objekat u memoriji

Primjer:

```
void f(int *p)
{
    if (p != NULL)
        // dereferenciranje
    else
        // greska
}
```



```
void f(int *p)
{
    if (p)
        // dereferenciranje
    else
        // greska
}
```

Referenciranje i dereferenciranje

- Iako su sve adrese (svi pokazivači) iste veličine, veoma je bitno kojeg je tipa pokazivač, **jer se pokazivači različitih tipova različito dereferenciraju**
- Dereferenciranjem pokazivača pristupa se adresi na koju pokazuje pokazivač, a sadržaj na datoj adresi se interpretira kao podatak tipa T

Ako je deklarisan pokazivač **TIP *p;**
tada važi **sizeof(*p) = sizeof(TIP)**

Primjer:

```
#include <stdio.h>
int main()
{
    char *pc;
    int *pi;
    double *pd;
    printf("pc:%d    *pc:%d\n", sizeof(pc), sizeof(*pc));
    printf("pi:%d    *pi:%d\n", sizeof(pi), sizeof(*pi));
    printf("pd:%d    *pd:%d\n", sizeof(pd), sizeof(*pd));
    return 0;
}
```

```
pc:4    *pc:1
pi:4    *pi:4
pd:4    *pd:8
```

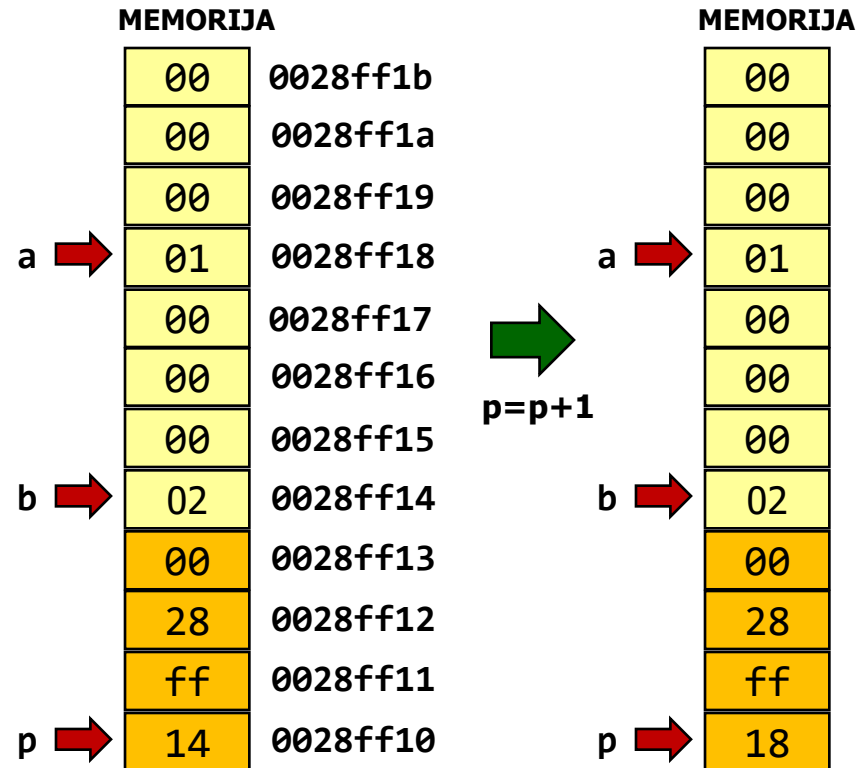

Pokazivačka aritmetika

- Pokazivačka aritmetika uključuje operacije koje se izvode nad pokazivačima
- Pokazivači sadrže adrese pa pokazivačka aritmetika uključuje operacije nad adresama, a rezultati zavise od pokazivačkih tipova
- **Zbog toga se pokazivačka (adresna) aritmetika razlikuje od cjelobrojne aritmetike**

Primjer:

```
#include <stdio.h>
int main()
{
    int a=1, b=2;
    printf("b:%p\n", &b);
    printf("a:%p\n", &a);

    int *p=&b;
    printf("p:%p *p:%d\n", p, *p);
    p=p+1;
    printf("p:%p *p:%d\n", p, *p);
    return 0;
}
```



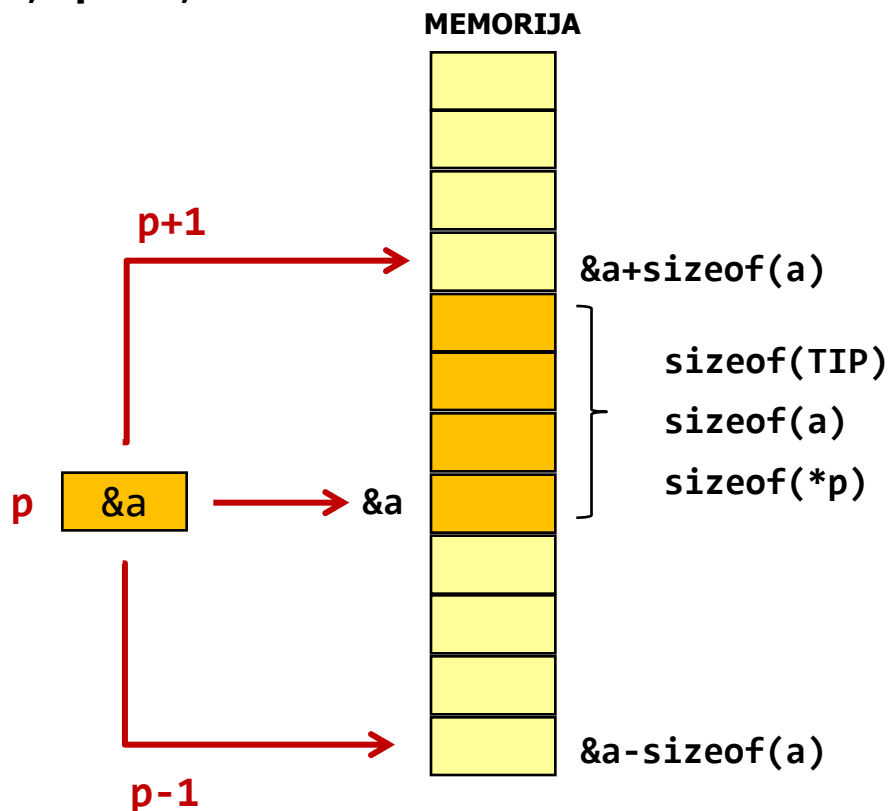
```
b:0028ff14
a:0028ff18
p:0028ff14 *p:2
p:0028ff18 *p:1
```

Inkrementovanje pokazivača rezultuje pokazivanjem sljedećeg podatka istog tipa u memoriji

Pokazivačka aritmetika

Osnovna pravila pokazivačke aritmetike

TIP `a`, `*p=&a`;



Uvećavanje pokazivača

$$p+k == p+k*\text{sizeof}(*p)$$

Umanjivanje pokazivača

$$p-k == p-k*\text{sizeof}(*p)$$

Razlika dva pokazivača

$$p1-p2 == (p1-p2)/\text{sizeof}(*p1)$$

Pokazivačka aritmetika

Uvećavanje pokazivača

Primjer:

```
char c, *p=&c;
printf("p:      %p\n", p);
printf("++p:    %p\n", ++p);
printf("p+1:     %p\n", p+1);
printf("p+10:    %p\n", p+10);
printf("p:      %p\n", p);
printf("p++:     %p\n", p++);
printf("p:      %p\n", p);
```

Primjer:

```
int i, *p=&i;
printf("p:      %p\n", p);
printf("++p:    %p\n", ++p);
printf("p+1:     %p\n", p+1);
printf("p+10:    %p\n", p+10);
printf("p:      %p\n", p);
printf("p++:     %p\n", p++);
printf("p:      %p\n", p);
```

Dozvoljeno:

- inkrementovanje pokazivača
- uvećavanje pokazivača za konstantu

`p+k == p+k*sizeof(*p)`

```
p:      0028ff1b
++p:    0028ff1c
p+1:    0028ff1d
p+10:   0028ff26
p:      0028ff1c
p++:    0028ff1c
p:      0028ff1d
```

```
p:      0028ff18
++p:    0028ff1c
p+1:    0028ff20
p+10:   0028ff44
p:      0028ff1c
p++:    0028ff1c
p:      0028ff20
```

Nije dozvoljeno:

- sabiranje dva pokazivača

Pokazivačka aritmetika

Umanjivanje pokazivača

Primjer:

```
char c, *p=&c, *q=&c;
printf("p:      %p\n", p);
printf("--p:   %p\n", --p);
printf("p-2:    %p\n", p-2);
printf("p--:    %p\n", p--);
printf("p:      %p\n", p);
printf("q:      %p\n", q);
printf("q-p:    %d\n", q-p);
```

```
p:      0028ff17
--p:    0028ff16
p-2:    0028ff14
p--:    0028ff16
p:      0028ff15
q:      0028ff17
q-p:    2
```

Primjer:

```
int i, *p=&i, *q=&i;
printf("p:      %p\n", p);
printf("--p:   %p\n", --p);
printf("p-2:    %p\n", p-2);
printf("p--:    %p\n", p--);
printf("p:      %p\n", p);
printf("q:      %p\n", q);
printf("q-p:    %d\n", q-p);
```

```
p:      0028ff14
--p:    0028ff10
p-2:    0028ff08
p--:    0028ff10
p:      0028ff0c
q:      0028ff14
q-p:    2
```

Dozvoljeno:

- dekrementovanje pokazivača
- umanjivanje pokazivača za konstantu

$p-k == p-k*\text{sizeof}(*p)$

- oduzimanje dva pokazivača istog tipa (broj podataka datog tipa između dvije adrese)

$p1-p2 == (p1-p2)/\text{sizeof}(*p1)$

Pokazivačka aritmetika

Poređenje pokazivača

- Dozvoljeno je porediti pokazivače, tj. primjenjivati operatore poređenja

<, <=, >, >=, ==, !=

Primjer:

```
char c, *p=&c, *q=&c;
printf("p==q:   %d\n", p==q);
printf("*p==*q: %d\n", *p==*q);
q++;
printf("p<q:    %d\n", p<q);
printf("p>q:    %d\n", p>q);
```

```
p==q:   1
*p==*q: 1
p<q:    1
p>q:    0
```

Složeni izrazi

- Posebno treba voditi računa o složenim izrazima u kojima figurišu operatori inkrementovanja i dekrementovanja u kombinaciji sa operatorom dereferenciranja
- **Postfiksni operatori imaju viši prioritet u odnosu na prefiksne unarne operatore (postfiksni ++ ima prioritet u odnosu na *)**

Npr:

```
*p++; // vrijednost izraza je *p, tj.
      // pristupa se lokaciji *p,
      // a zatim se inkrementuje p
```

```
*(p++); // prvo se inkrementuje p,
        // a zatim se pristupa lokaciji
        // na koju pokazuje uvecani p
```

```
(*p)++; // inkrementuje se dereferencirani
        // pokazivac, tj. inkrementuje se
        // podatak na lokaciji na koju
        // pokazuje p
```



Konstantni pokazivači

pokazivač na konstantu

pokazivač na konstantu tipa T

const T* p;

moguće je mijenjati
pokazivač, ali ne i ono na šta
on pokazuje

Primjer:

```
const int k1=100, k2=200;
```

```
const int *p;
```

```
p=&k1;
```

```
*p=1; // ne moze se promijeniti  
      // ono na sta pokazuje p
```

```
p=&k2; // moze se promijeniti p
```

konstantan pokazivač

konstantan pokazivač na
promjenljivu tipa T

T* const p ;

moguće je mijenjati ono
na šta pokazuje pokazivač,
ali ne i sam pokazivač

Primjer:

```
int k1=100, k2=200;
```

```
int* const p = &k1;
```

```
*p=1; // isto kao k1=1
```

```
p=&k2; // ne moze se  
      // promijeniti p
```

konstantan pokazivač na konstantu

konstantan pokazivač na
konstantu tipa T

const T* const p ;

nije moguće mijenjati ni
pokazivač ni ono na šta on
pokazuje

Primjer:

```
const int k1=100;
```

```
int k2;
```

```
const int* const p = &k1;
```

```
*p=1; // ne moze se mijenjati  
      // ono na sta pokazuje p
```

```
p=&k2; // ne moze se  
      // promijeniti p
```

Višestruki pokazivači

Pokazivač na pokazivač

- **Pokazivač na pokazivač** je pokazivač koji sadrži adresu nekog pokazivača i omogućava indirektan pristup tom pokazivaču

Definicija pokazivača na pokazivač:

tip **pp;

Primjeri:

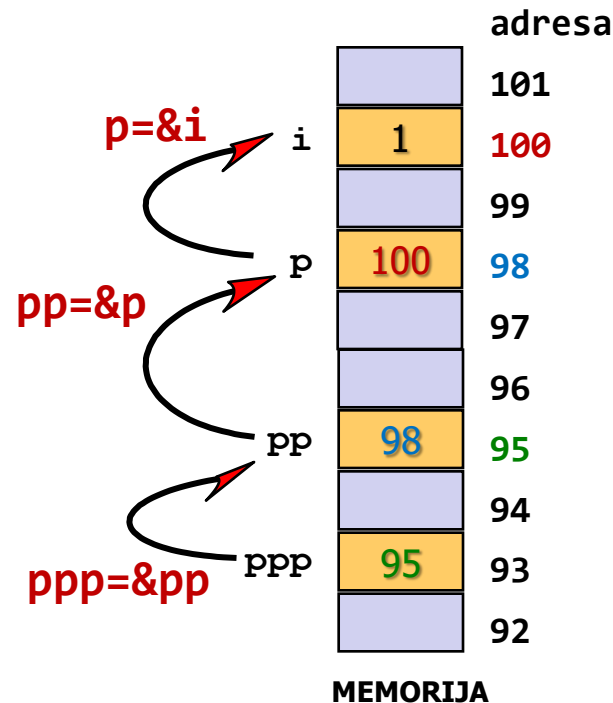
```
int i;
```

```
int *p=&i;    // p je pokazivac na int i pokazuje i
```

```
int **pp;     // pp je pokazivac na pokazivac na int  
pp = &p;      // pp pokazuje na p
```

```
int ***ppp=&pp; // ppp je trostruki pokazivac  
                // i pokazuje na pp
```

Višestrukost pokazivača
(broj nivoa indirekcije)
nije ograničena



Višestruki pokazivači

Pokazivač na pokazivač

Primjer (veličina pokazivača):

```
#include <stdio.h>
int main()
{
    char c;
    char *p1 = &c;
    char **p2 = &p1;
    char ***p3 = &p2;
    char ****p4 = &p3;
    printf("p1: %d\n", sizeof(p1));
    printf("p2: %d\n", sizeof(p2));
    printf("p3: %d\n", sizeof(p3));
    printf("p4: %d\n", sizeof(p4));
    return 0;
}
```

```
p1: 4
p2: 4
p3: 4
p4: 4
```

Primjer (razmještaj u memoriji):

```
#include <stdio.h>
int main()
{
    char c;
    char *p1 = &c;
    char **p2 = &p1;
    char ***p3 = &p2;
    printf("&c: %p\n", &c);
    printf("&p1: %p  p1:%p\n", &p1, p1);
    printf("&p2: %p  p2:%p\n", &p2, p2);
    printf("&p3: %p  p3:%p\n", &p3, p3);
    return 0;
}
```

```
&c: 0028ff1f
&p1: 0028ff18  p1:0028ff1f
&p2: 0028ff14  p2:0028ff18
&p3: 0028ff10  p3:0028ff14
```


Višestruki pokazivači

Dereferenciranje višestrukih pokazivača

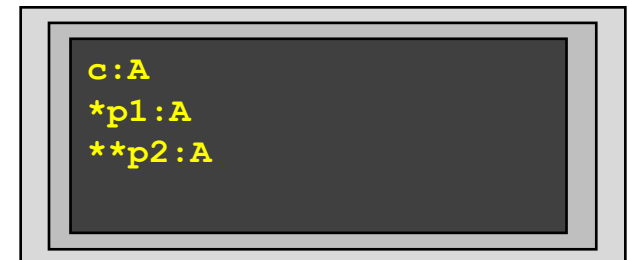
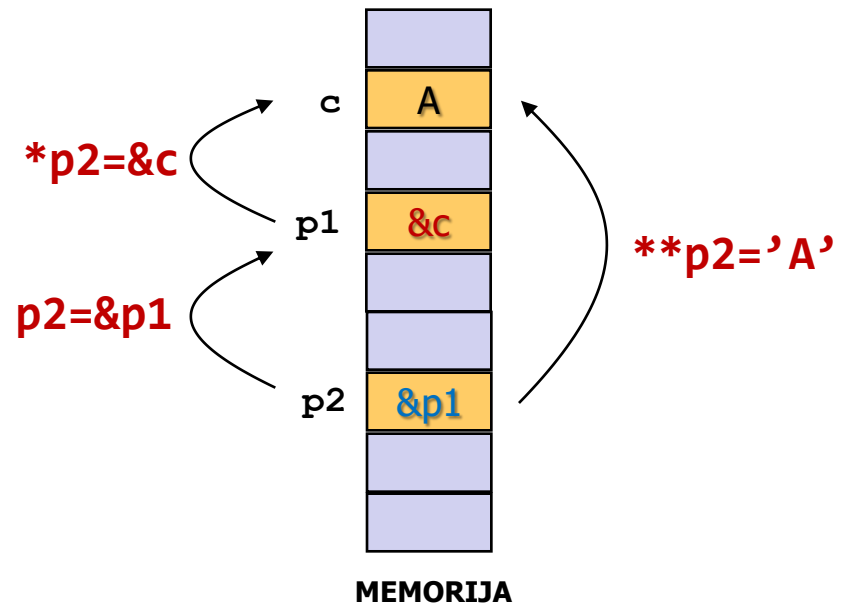
Primjer:

```
#include <stdio.h>
int main()
{
    char c;
    char *p1;
    char **p2;

    p2 = &p1;
    *p2 = &c;    // *p2=p1 => p1=&c
    **p2 = 'A';  // *p2=p1 => **p2=*p1=c => c='A'

    printf("c:%c\n", c);
    printf("*p1:%c\n", *p1);
    printf("**p2:%c\n", **p2);

    return 0;
}
```





Pokazivači i argumenti funkcije

➤ Argumenti se prilikom poziva funkcije prenose PO VRIJEDNOSTI:

1. prilikom poziva funkcije šalje se vrijednost argumenta
2. funkcija prihvata proslijeđene vrijednosti u formalne argumente – formira se kopija stvarnih argumenata na steku i funkcija koristi te kopije, a ne originale
3. prilikom izlaska iz funkcije formalni argumenti automatski nestaju

U funkciji se kreira slika (kopija) stvarnih argumenata i koriste se te kopije, a ne stvarne promjenljive iz glavnog programa, zato nakon izlaska iz funkcije promjenljive u glavnom programu ostaju nepromijenjene!!!

➤ Prenos argumenata REFERISANJEM:

Da bi se omogućilo da funkcija mijenja vrijednost stvarnog argumenta:

1. Prilikom poziva funkcije treba slati adresu!
2. Formalni argument treba definisati kao pokazivač na tip stvarnog argumenta!
3. U tijelu funkcije treba koristiti operator indirekcije!

REFERISANJE = PRENOS ADRESE (REFERENCE) stvarnog argumenta

Referisanje omogućava da funkcija mijenja vrijednost stvarnog argumenta!

Pokazivači i argumenti funkcije

Primjer prenosa argumenata referisanjem:

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

```
void suma ( int niz[], int n, int *s )
```

```
{
```

```
    *s = 0;
```

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

```
        *s += niz[i];
```

```
}
```

```
int main()
```

```
{
```

```
    int niz[]={1,2,3,4};
```

```
    int x;
```

```
    suma( niz, 4, &x );
```

```
    printf("Suma: %d", x);
```

```
    return 0;
```

```
}
```

Korištenje
operatora
indirekcije

Formalni argument
je pokazivač

šalje se adresa

Suma: 10



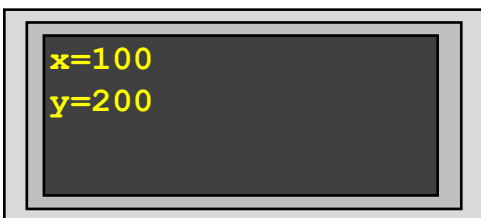
Pokazivači i argumenti funkcije

Primjer prenosa argumenata po vrijednosti:

```
#include <stdio.h>

void zamjena( int x, int y )
{
    int pom = x;
    x = y;
    y = pom;
}

int main()
{
    int x=100, y=200;
    zamjena( x, y );
    printf("x=%d\n", x);
    printf("y=%d\n", y);
    return 0;
}
```



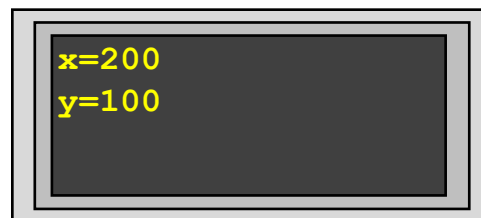
```
x=100
y=200
```

Primjer prenosa argumenata po adresi:

```
#include <stdio.h>

void zamjena( int *px, int *py )
{
    int pom = *px;
    *px = *py;
    *py = pom;
}

int main()
{
    int x=100, y=200;
    zamjena( &x, &y );
    printf("x=%d\n", x);
    printf("y=%d\n", y);
    return 0;
}
```



```
x=200
y=100
```

Pokazivači i argumenti funkcije

Prenos pokazivača u funkciju

- Prilikom prenosa pokazivača u funkciju, treba voditi računa o tome da li se prenos vrši po vrijednosti ili po adresi

Prenos pokazivača po vrijednosti (original se ne mijenja)

Primjer:

```
#include <stdio.h>
void f( int *p )
{
    p=NULL;
}
int main()
{
    int x, *px=&x;
    f(px);
    printf("px:%p", px);
    return 0;
}
```

px:0028ff18

Prenos pokazivača po adresi (original se mijenja)

Primjer:

```
#include <stdio.h>
void f( int **p )
{
    *p=NULL;
}
int main()
{
    int x, *px=&x;
    f(&px);
    printf("px:%p", px);
    return 0;
}
```

px:00000000

Pokazivači i strukture


Pokazivač na strukturu

Neka imamo strukturu

```
struct datum { int dd, mm, gg; };
```

Tada možemo da definišemo pokazivač na strukturu:

```
struct datum *pd;
```



Pokazivač na strukturu datum
(još uvijek ne pokazuje
nikakav konkretan podatak)

Npr:

```
struct datum d, *pd=&d;
```

Npr:

```
struct datum d, *pd;  
pd=&d;
```

Pristup strukturi pomoću pokazivača (indirektan pristup elementima strukture)

Direktan pristup elementima strukture

`promjenljiva.elementStrukture`

Indirektan pristup elementima strukture

`(*pokazivac).elementStrukture`

ili

`pokazivac->elementStrukture`

Npr:

```
struct datum d, *pd=&d;
```

```
(*pd).dd=1;  
(*pd).mm=1;  
(*pd).gg=2022;
```



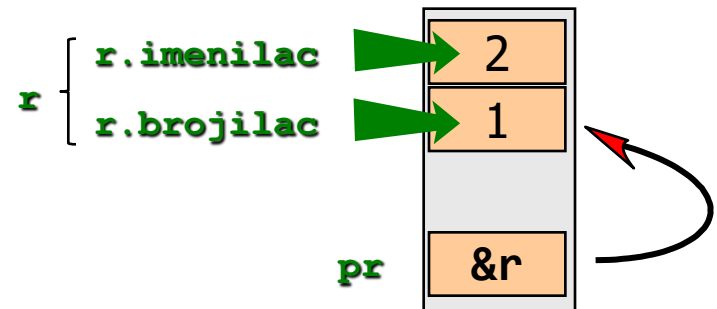
```
pd->dd=1;  
pd->mm=1;  
pd->gg=2022;
```

Pokazivači i strukture

Pristup strukturi pomoću pokazivača (indirektan pristup elementima strukture)

Primjer:

```
#include <stdio.h>
int main()
{
    struct razlomak{ int brojilac, imenilac; };
    struct razlomak r, *pr = &r;
    pr->brojilac = 1;
    pr->imenilac = 2;
    printf("Direktno: %d/%d\n", r.brojilac, r.imenilac);
    printf("Indirektno: %d/%d\n", pr->brojilac, pr->imenilac);
    return 0;
}
```



```
Direktno: 1/2
Indirektno: 1/2
```

Pokazivači i strukture

Pokazivač kao element strukture

Pokazivač može da bude element strukture

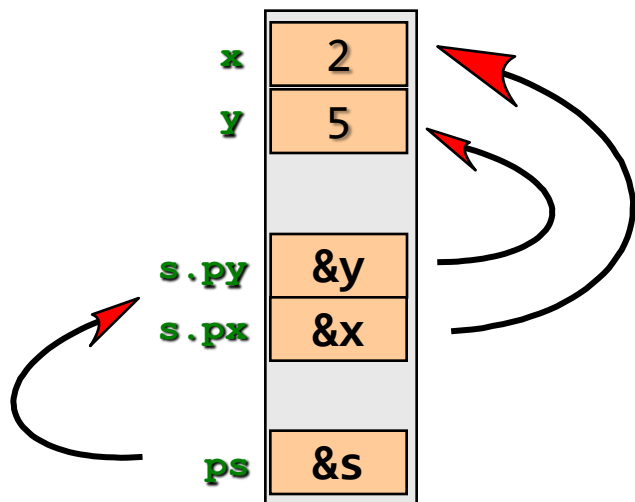
```
struct primjer { T *p; } s, *ps=&s;
```

Direktno dereferenciranje pokazivača u strukturi

***s.p**

Indirektno dereferenciranje pokazivača u strukturi

***ps->p**



Primjer:

```
#include <stdio.h>
int main()
{
    int x, y;
    struct strSaPok { int *px, *py; };
    struct strSaPok s, *ps = &s;

    // direktan pristup pokazivacu u strukturi
    s.px = &x; *s.px = 2;

    // indirektan pristup pokazivacu u strukturi
    ps->py = &y; *ps->py = 5;

    printf("x=%d\n", x);
    printf("y=%d\n", y);

    return 0;
}
```



Pokazivači i strukture

Povezane strukture

Kombinovanjem **pokazivača na strukturu** i **strukture sa pokazivačem** možemo da dobijemo **strukturu sa pokazivačem na istu strukturu** tj. imamo **povezane strukture** (eng. *Self-referencing structure*)

```
struct cvor {
```

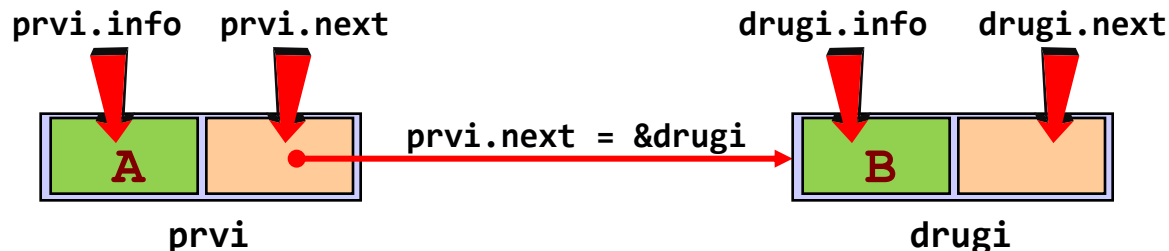
```
    tip info;
```

```
    struct cvor *next;
```

```
} prvi, drugi;
```

informaciono polje
u strukturi

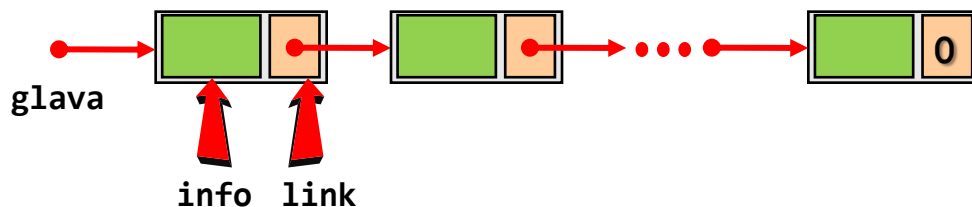
pokazivač na istu strukturu
(omogućava povezivanje struktura i
pravljenje liste - dinamičkog niza)



Pokazivači i strukture

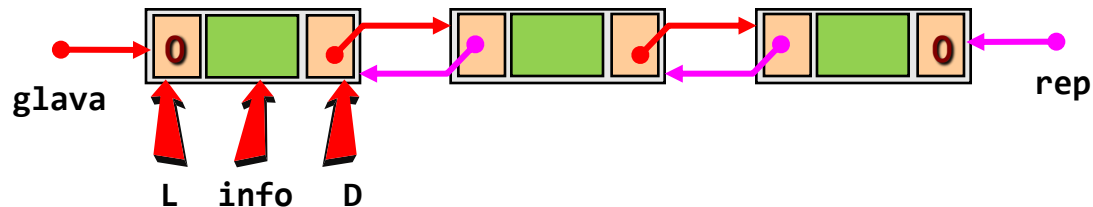
Primjeri povezanih struktura

Linearna jednostruko povezana lista



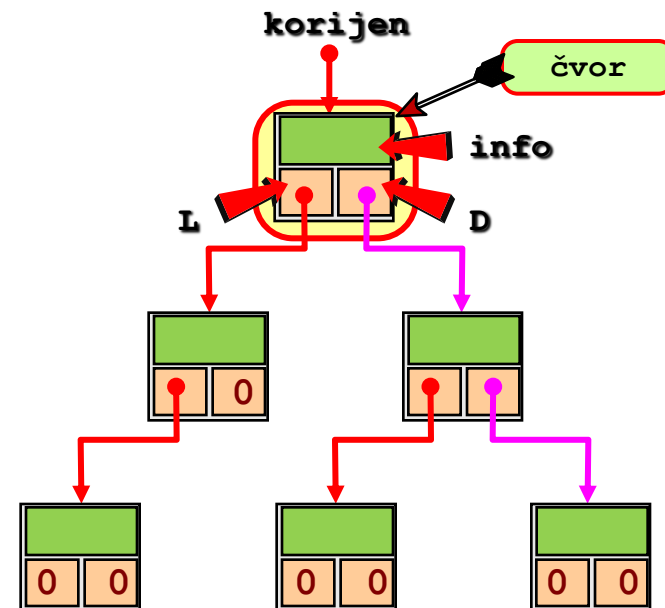
```
struct element {  
    tip info;  
    struct element *link;  
} *glava;
```

Linearna dvostruko povezana lista



Detaljnije u okviru kursa
PROGRAMIRANJE 2

Binarno stablo



Pokazivači i strukture

Prenos strukture u funkciju

Iako se struktura u funkciju može prenijeti i po vrijednosti i po adresi, u **praksi se struktura češće prenosi po adresi** (jer je prostorno i vremenski manje zahtjevno)

Prenos strukture po vrijednosti

Primjer:

```
#include <stdio.h>
struct S { int x, *p; };
void f( struct S str )
{
    str.x=0; str.p=NULL;
}
int main()
{
    int i;
    struct S s={i,&i};
    f(s);
    printf("s.x:%d\n", s.x);
    printf("s.p:%p\n", s.p);
    return 0;
}
```

```
s.x:2
s.p:0028ff1c
```

Prenos strukture po adresi

Primjer:

```
#include <stdio.h>
struct S { int x, *p; };
void f( struct S *ps )
{
    ps->x=0; ps->p=NULL;
}
int main()
{
    int i;
    struct S s={i,&i};
    f(&s);
    printf("s.x:%d\n", s.x);
    printf("s.p:%p\n", s.p);
    return 0;
}
```

```
s.x:0
s.p:00000000
```

Pokazivači i strukture

Prenos strukture u funkciju

Ako se prilikom prenosa strukture po adresi (u cilju smanjenja prostorne i vremenske složenosti) želi onemogućiti da funkcija mijenja original, tada **parametar mora biti deklarisan kao pokazivač na konstantu**

Primjer:

```
struct S { int x, *p; };  
void f( const struct S *ps )  
{  
    ps->x=0;      // greska:  
    ps->p=NULL;   // ps pokazuje na konstantu  
}
```

Primjer:

```
#include <stdio.h>  
struct S { int x, *p; };  
void f( const struct S *ps )  
{  
    printf("ps->x:%d\n", ps->x);  
    printf("ps->p:%p\n", ps->p);  
}  
int main()  
{  
    int i=100;  
    struct S s={i,&i};  
    f(&s);  
    return 0;  
}
```

```
ps->x:100  
ps->p:0028ff1c
```

Parametar je deklarisan kao pokazivač na konstantu pa funkcija samo čita i ispisuje vrijednosti elemenata strukture

Pokazivači i nizovi

Veza između nizova i pokazivača

- Postoji čvrsta veza između nizova i pokazivača
- **Ime niza pokazuje na početni element niza i već je samo po sebi pokazivač** (konstantan pokazivač – ne može da se mijenja tokom izvršavanja programa)

Primjer:

```
#include <stdio.h>
int main()
{
    int a[5]={};
    printf("a:      %p\n", a);
    printf("&a:      %p\n", &a);
    printf("&a[0]: %p\n", &a[0]);
    return 0;
}
```

```
a:      0028ff0c
&a:      0028ff0c
&a[0]: 0028ff0c
```

- Operacije nad nizovima mogu se iskazati i pomoću pokazivača i često se ne pravi razlika između ova dva načina pristupa nizu

```
int a[5]={};
int *p=a; // *p=&a ili *p=&a[0]
```

a[4]	0	← p+4
a[3]	0	← p+3
a[2]	0	← p+2
a[1]	0	← p+1
a[0]	0	← p

adresa i-tog elementa niza

$\&a[i]$ ↔ $p+i$

pristup i-tom elementu niza

$a[i]$ ↔ $p[i]$ ↔ $*(p+i)$

Pokazivači i nizovi

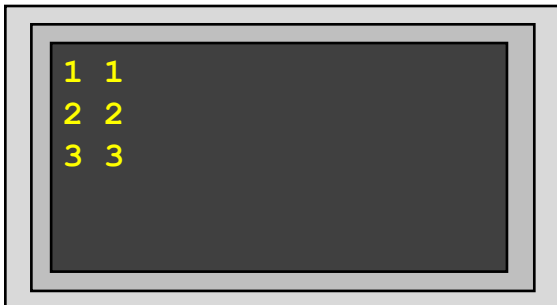
Veza između nizova i pokazivača

Primjer:

```
#include <stdio.h>

void f(int niz[], int n)
{
    for (int i=0; i<3; i++)
        printf("%d %d\n", niz[i], *(niz+i));
}

int main()
{
    int x[] = {1,2,3};
    f(x,3);
    return 0;
}
```

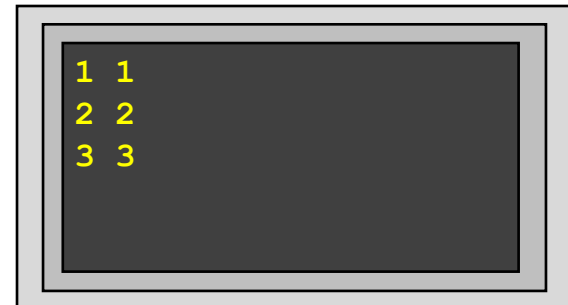


Primjer:

```
#include <stdio.h>

void f(int *p, int n)
{
    for (int i=0; i<3; i++)
        printf("%d %d\n", p[i], *(p+i));
}

int main()
{
    int x[] = {1,2,3};
    f(x,3);
    return 0;
}
```



Pokazivači i nizovi

Niz pokazivača

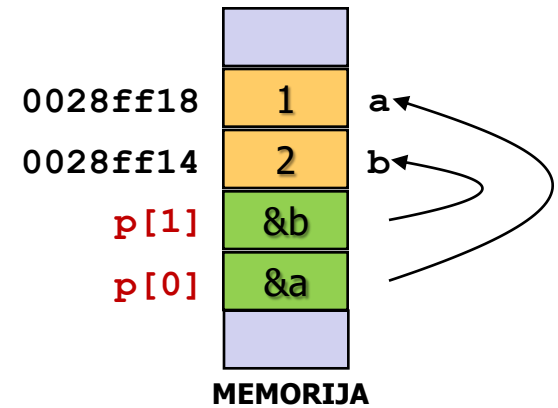
➤ Definicija niza pokazivača na TIP

TIP *niz[DIM];		TIP* niz[DIM];
TIP *niz[DIM]={adrese};	↔	TIP* niz[DIM]={adrese};
TIP *niz[]={adrese};		TIP* niz[]={adrese};

Primjer:

```
#include <stdio.h>
int main()
{
    int a=1, b=2;
    int *p[2]={&a, &b};
    for (int i=0; i<2; i++)
        printf("p[%d]:%p    *p[%d]:%d\n", i, p[i], i, *p[i]);
    return 0;
}
```

```
p[0]:0028ff18    *p[0]:1
p[1]:0028ff14    *p[1]:2
```



Dereferenciranje pokazivača



Dinamička alokacija memorije

Statička alokacija memorije

- **Statička alokacija memorije** podrazumijeva da **programer u fazi programiranja mora da procijeni potrebe za memorijom (najgori slučaj)** i **alocira niz odgovarajuće veličine**.
- U većini realnih aplikacija, **nije moguće unaprijed precizno procijeniti potrebe za memorijom** (npr. u fazi programiranja unaprijed procjenjujemo najgori slučaj, a **korisnik tek u fazi izvršavanja programa određuje koliko elemenata niza će stvarno biti korišteno**).
- U nekim slučajevima, **nije moguće procijeniti gornje ograničenje**
 - ako je **ograničenje premalo**, program **nije u stanju da obrađuje sve ulazne podatke**
 - ako je **ograničenje preveliko**, program **zauzima više memorije nego što mu je stvarno potrebno**

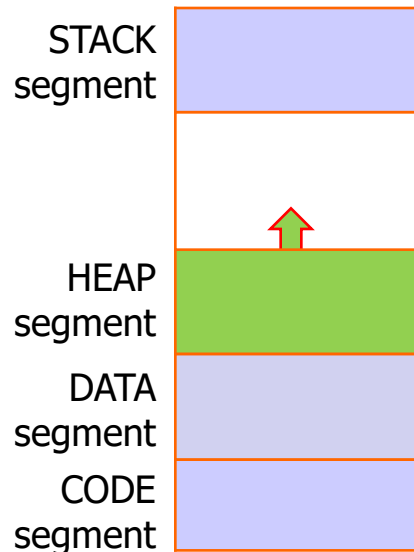
Dinamička alokacija memorije

- Rješenje problema vezanih za statičku alokaciju jeste dinamička alokacija memorije
- **Dinamička alokacija memorije podrazumijeva da program u fazi izvršavanja**
 - **dinamički alocira (zauzima) potreban memorijski prostor** (program od operativnog sistema zahtijeva potrebnu količinu memorije) i
 - **deallocira (oslobađa) zauzetu memoriju kad više ne postoji potreba** (program može i dužan je da je oslobodi i tako je vrati operativnom sistemu na upravljanje)
- Dinamičkom memorijom upravlja operativni sistem, koji vodi računa o raspoloživim i zauzetim blokovima memorije

Dinamička alokacija memorije

Upravljanje dinamičkom zonom memorije

- Prostor za dinamički alociranu memoriju nalazi se u **hîp (heap)** segmentu
- Alokacija i dealokacija vrši se funkcijama iz **standardne biblioteke** i pozivima runtime biblioteke.



Standardne funkcije za rad sa dinamičkom memorijom

Funkcije za rad sa dinamičkom memorijom deklarirane su u zaglavlju **<stdlib.h>**

void *malloc(size_t n)

alocira memorijski blok veličine n bajtova

void *calloc(size_t n, size_t size)

alocira memorijski blok za n elemenata veličine size bajtova (ukupno n*size bajtova)

void free(void *p)

oslobađa memorijski blok na koji pokazuje p

void *realloc(void *p, size_t size)

mijenja veličinu memorijskog bloka na koji pokazuje pokazivač p (size je nova veličina)

Dinamička alokacija memorije

Alokacija

```
void *malloc(size_t n)
```

- Omogućava alokaciju memorijskog bloka veličine **n bajtova**
- Alocirani blok ima "slučajni" sadržaj – svi bajtovi zadržavaju binarni sadržaj koji je prethodno bio na tim lokacijama

```
void *calloc(size_t n, size_t size)
```

- Omogućava alokaciju memorijskog bloka za **n podataka veličine size bajtova (ukupno n*size bajtova)**
- Svi bajtovi u alociranom blok se brišu – svi bajtovi imaju vrijednost 0
- **Za obje funkcije važi:**
 - u slučaju uspješne alokacije, **funkcija vraća adresu alociranog memorijskog bloka**
 - u slučaju neuspješne alokacije, **funkcija vraća 0**

Rezultat treba kastovati u odgovarajući pokazivački tip i prihvatiti pomoću odgovarajućeg pokazivača

```
TIP *p;  
p = (TIP*) malloc(size);
```

```
if (p)  
{  
    // uspjesna alokacija  
    // indirektan pristup bloku  
    // pomocu pokazivaca (*p)  
}  
else  
    // neuspjesna alokacija
```

Obavezno provjeriti uspješnost alokacije, da bismo izbjegli dereferenciranje null-pointera

Dinamička alokacija memorije

Dealokacija

```
void free(void *p)
```

- Omogućava dealokaciju memorijskog bloka u dinamičkoj zoni memorije na koji pokazuje pokazivač p
- Izvršavanjem funkcije free samo se dati memorijski blok proglašava slobodnim, a sadržaj memorijskog bloka se ne briše niti se mijenja vrijednost pokazivača
- Svaki dinamički alocirani blok **TREBA OSLOBODITI** kad prestane potreba za postojanjem tog bloka

Pošto se vrijednost pokazivača ne mijenja, poželjno mu je postaviti vrijednost na NULL, kako ne bi slučajno došlo do ponovnog pristupa oslobođenom bloku (koji se možda već koristi za nešto drugo)

```
TIP *p;  
p = (TIP*) malloc(size);  
  
if (p)  
{  
    // uspjesna alokacija  
    // indirektan pristup bloku  
    // pomocu pokazivaca (*p)  
  
    free (p);  
    p = NULL;  
}  
else  
    // neuspjesna alokacija
```

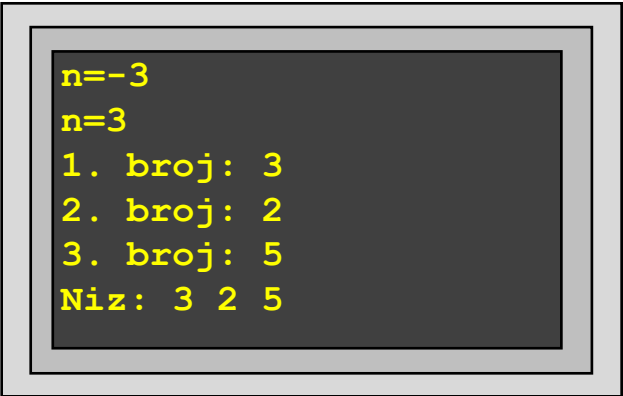


Dinamička alokacija memorije

Primjer (dinamički niz):

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

int main()
{
    int i, n, *niz;
    do
    {
        printf("n="); scanf("%d",&n);
    }
    while (n<0);
    niz = (int*) malloc(n*sizeof(int));
```



```
n=-3
n=3
1. broj: 3
2. broj: 2
3. broj: 5
Niz: 3 2 5
```

```
    if (niz) // uspjesna alokacija
    {
        for (i=0; i<n; i++)
        {
            printf("%d. broj: ", i+1);
            scanf("%d", niz+i);
        }
        printf("Niz:");
        for (i=0; i<n; i++)
            printf(" %d", *(niz+i));

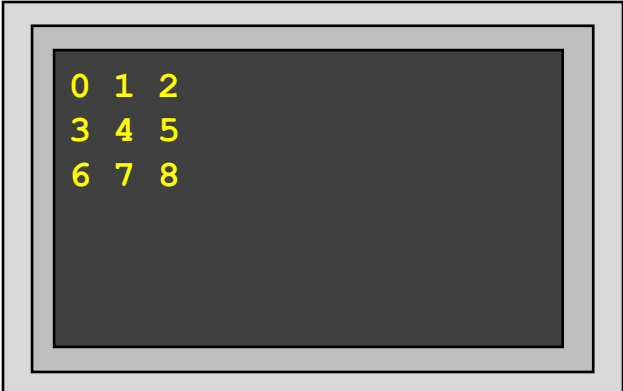
        free(niz);
        niz = NULL;
    }
    else
        printf("Neuspjesna alokacija!");
    return 0;
}
```

Dinamička alokacija memorije

Primjer (dinamička matrica):

```
#include <stdio.h>
#include <stdlib.h>
#define RED 3
int main()
{
    int **mat, i, j, b=0;
    mat = (int**) calloc(RED, sizeof(int*));
    for (i=0; i<RED; i++)
    {
        mat[i]=(int*) calloc(RED, sizeof(int));
        for (j=0; j<RED; j++, b++)
            mat[i][j] = b;
    }
    for (i=0 ; i<RED ; i++ )
    {
        for (j=0; j<RED; j++)
            printf("%d ", (*(mat+i)+j));
        printf("\n");
    }
    for (i=0; i<RED; i++) free(mat[i]);
    free(mat);
    return 0;
}
```

$\text{mat}[i][j] \longleftrightarrow *(*(mat+i)+j)$



0	1	2
3	4	5
6	7	8



Dinamička alokacija memorije

Realokacija

```
void *realloc(void *p, size_t size)
```

- Omogućava promjenu veličine dinamički alociranog bloka na koji pokazuje pokazivač p (size je nova veličina)
- U slučaju uspješne realokacije **funkcija vraća adresu alociranog bloka**, a u slučaju neuspješne realokacije **funkcija vraća 0**
- **Zahtjev za smanjivanje bloka uvijek uspijeva** (tipično blok ostaje na istoj adresi, samo se skraćuje)
- U slučaju da se zahtijeva povećavanje bloka:
 - ako iza datog bloka postoji dovoljno slobodnog prostora – blok ostaje na istoj adresi i jednostavno se proširuje
 - ako iza datog bloka ne postoji dovoljno slobodnog prostora – blok se premješta na drugo mjesto gdje ima dovoljno prostora (sadržaj postojećeg bloka se kopira na to novo mjesto, nakon čega se stari blok oslobađa), što može biti vremenski zahtjevno
- Programer može da realizuje promjenu veličine dinamički alociranog bloka i bez poziva funkcije realloc, ali tada mora da:
 - alocira novi blok pozivom funkcija malloc ili calloc
 - prekopira sve podatke u novi blok
 - oslobodi stari blok
- Budući da proširenje bloka može biti neuspješno (funkcija vraća 0), postoji mogućnost da se izgube stari podaci ako bi se funkcija pozivala na sljedeći način
p = (TIP*) realloc(p,size);



Dinamička alokacija memorije

Primjer (dinamičko proširivanje niza):

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int *niz = NULL; // niz prazan na pocetku
    int n = 0;        // trenutni broj podataka
    int kap = 0;      // kapacitet niza
    int b;

    do
    {
        printf("Unesi broj (0 za kraj): ");
        scanf("%d", &b);
        // prosiri niz ako nema mjesta
        if (n == kap)
        {
            kap += 5;
            niz = (int*)realloc(niz, kap*sizeof(b));
            if (niz == NULL) return 1;
        }
        niz[n++] = b;
    } while (b);
```

```
// ispis
printf("Uneseno brojeva: %d\n", n);
printf("Kapacitet niza: %d\n", kap);
printf("Brojevi su : ");
for (b=0; b<n; b++)
    printf("%d ", niz[b]);

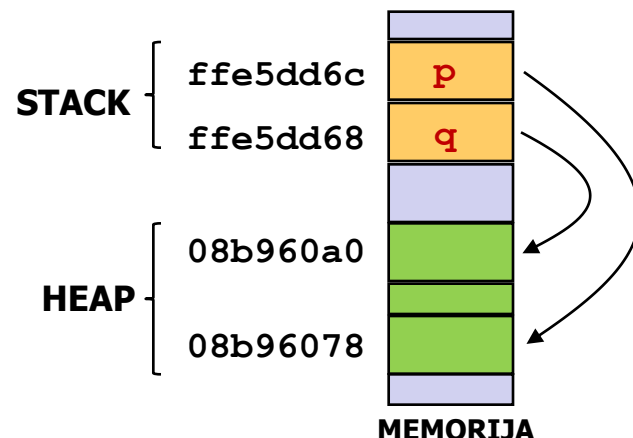
// oslobadjanje memorije
free(niz);
return 0;
}
```

```
Unesi broj (0 za kraj): 1
Unesi broj (0 za kraj): 2
Unesi broj (0 za kraj): 3
Unesi broj (0 za kraj): 4
Unesi broj (0 za kraj): 5
Unesi broj (0 za kraj): 6
Unesi broj (0 za kraj): 0
Uneseno brojeva: 7
Kapacitet niza: 10
Brojevi su : 1 2 3 4 5 6 0
```

Dinamička alokacija memorije

Dinamički životni vijek

- Prostor za dinamički alociranu memoriju nalazi se u **hîp (heap)** segmentu.
- Alokacija i dealokacija vrši se funkcijama iz standardne biblioteke (malloc, calloc, realloc).
- Ukoliko je alokacija uspješna, funkcije malloc, calloc, realloc vraćaju adresu u hip segmentu.
- Dinamički alocirani blokovi nisu imenovani, već im se pristupa indirektno (isključivo preko pokazivača koji sadrži početnu adresu bloka).
- **Dinamički alocirani blokovi nisu imenovani pa nemaju definisan doseg.**
- Doseg pokazivača pomoću kojih se pristupa dinamički alociranim blokovima podliježe standardnim pravilima.
- **Dinamički alocirani blokovi imaju dinamički životni vijek** – memorija se alocira i dealocira isključivo na eksplicitan zahtjev tokom izvršavanja programa.



Primjer (razmještaj u memoriji):

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int *p = (int*) malloc(sizeof(int));
    int *q = (int*) malloc(sizeof(int));
    printf("&p:%p    p:%p\n", &p, p);
    printf("&q:%p    q:%p\n", &q, q);
    return 0;
}
```

```
&p:ffe5dd6c  p:8b96078
&q:ffe5dd68  q:8b960a0
```




Dinamička alokacija memorije

Greške u radu sa dinamičkom memorijom

curenje memorije (eng. *memory Leak*)

- Jedna od najopasnijih grešaka u radu sa dinamički alociranom memorijom
- **Curenje memorije je situacija kada se u programu izgubi adresa dinamički alociranog, a neoslobođenog bloka**
- **U tom slučaju program više nema mogućnost da oslobodi taj blok i da mu pristupa do kraja izvršavanja (dio hipa ostaje "zarobljen" – blok memorije je zauzet, ali ne može da se koristi, jer je program izgubio adresu tog bloka)**
- **Curenje memorije se tipično teško uočava.** Obično se tokom razvoja, program testira kratkotrajno i sa malim ulazima. Međutim, kada se program pusti u dugotrajan rad (potencijalno i bez prestanka) i da obrađuje veće količine ulaznih podataka, curenje memorije postaje vidljivo, čini program neupotrebljivim (operativni sistem može da prekine izvršavanje programa ili može doći do degradiranja performansi) i može da uzrokuje velike štete.

➤ Primjeri curenja memorije:

- **preuzimanje adrese drugog bloka:**

```
TIP *p;  
p = (TIP*) malloc(size);  
p = (TIP*) calloc(n,size);
```

- **preuzimanje neke druge adrese:**

```
TIP *p, *q;  
p = (TIP*) malloc(size);  
p = q;
```

- **prestanak životnog vijeka pokazivača (npr. lokalni pokazivači u petljama):**

```
while (uslov)  
    TIP *p = (TIP*) malloc(size);
```

- **prestanak životnog vijeka pokazivača (npr. lokalni pokazivači u funkcijama):**

```
void f()  
{  
    TIP *p = (TIP*) malloc(size);  
}
```



Dinamička alokacija memorije

Greške u radu sa dinamičkom memorijom

pristup oslobođenoj memoriji

- **Nakon poziva `free(p)`, blok na koji pokazuje pokazivač `p` se oslobađa i više ne bi trebalo da ga koristimo.**
- Poziv `free(p)` ne mijenja vrijednost pokazivača `p` pa `p` i dalje pokazuje na isti blok. Moguće je da će nakon oslobađanja datog bloka, operativni sistem ponovo alocirati dati blok i da će taj blok biti korišten u neku drugu svrhu (drugi pokazivač bi pokazivao na taj blok) pa bi slučajno moglo da dođe do neželjenog pristupa i kompromitovanja sadržaja tog bloka pomoću pogrešnog pokazivača.
- Da slučajno ne bi došlo do ponovnog pristupa oslobođenom bloku, **nakon oslobađanja bloka treba odnosni pokazivač postaviti na `NULL`!**

```
TIP *p;
p = (TIP*) malloc(size);

if (p)
{
    // uspjesna alokacija
    // indirektan pristup bloku
    // pomocu pokazivaca (*p)

    free (p);
    p = NULL;
}
else
    // neuspjesna alokacija
```



Dinamička alokacija memorije

Greške u radu sa dinamičkom memorijom

višestruko oslobađanje istog bloka

- Nakon poziva `free(p)`, svaki naredni poziv `free(p)` za istu vrijednost pokazivača `p` prouzrokuje nedefinisano ponašanje programa i trebalo bi ga izbjegavati.
- Višestruka oslobađanja mogu da dovedu do pada programa, a mogu da budu i izvor sigurnosnih problema.

oslobađanje neispravnog pokazivača

- Funkciji `free(p)` dozvoljeno je proslijediti isključivo adresu koju je vratila funkcija `malloc/calloc/realloc`.
- Prosljeđivanje bilo koje druge adrese (npr. `p+10`), a ne početne adrese (adresa koju je vratila `malloc/calloc/realloc`) nekog dinamički alociranog bloka, može da uzrokuje probleme

prekoračenje i potkoračenje bloka

- **Pristup dinamički alociranom bloku dozvoljen je samo unutar granica bloka.**
- Kao i u slučaju statički alociranih nizova, pristup elementima van granica može da prouzrokuje ozbiljne probleme u radu programa (posebno je problematičan upis van granica bloka).
- **U slučaju dinamički alociranih blokova, obično se nakon samog bloka smještaju dodatne informacije potrebne za uspješno upravljanje dinamičkom memorijom. Zbog toga i malo prekoračenje granice bloka prilikom upisa može da izmijeni te dodatne informacije i tako uzrokuje pad sistema za dinamičko upravljanje memorijom.**