



PROGRAMIRANJE I

P-07: Funkcije

prof. dr **Dražen Brđanin**
2023/24



P-07: Funkcije

■ Sadržaj predavanja

- potprogrami
- funkcije u jeziku C
- definicija i deklaracija funkcije
- kontrola toka i prenos argumenata
- doseg identifikatora
- životni vijek promjenljivih



Potprogrami

Potprogrami

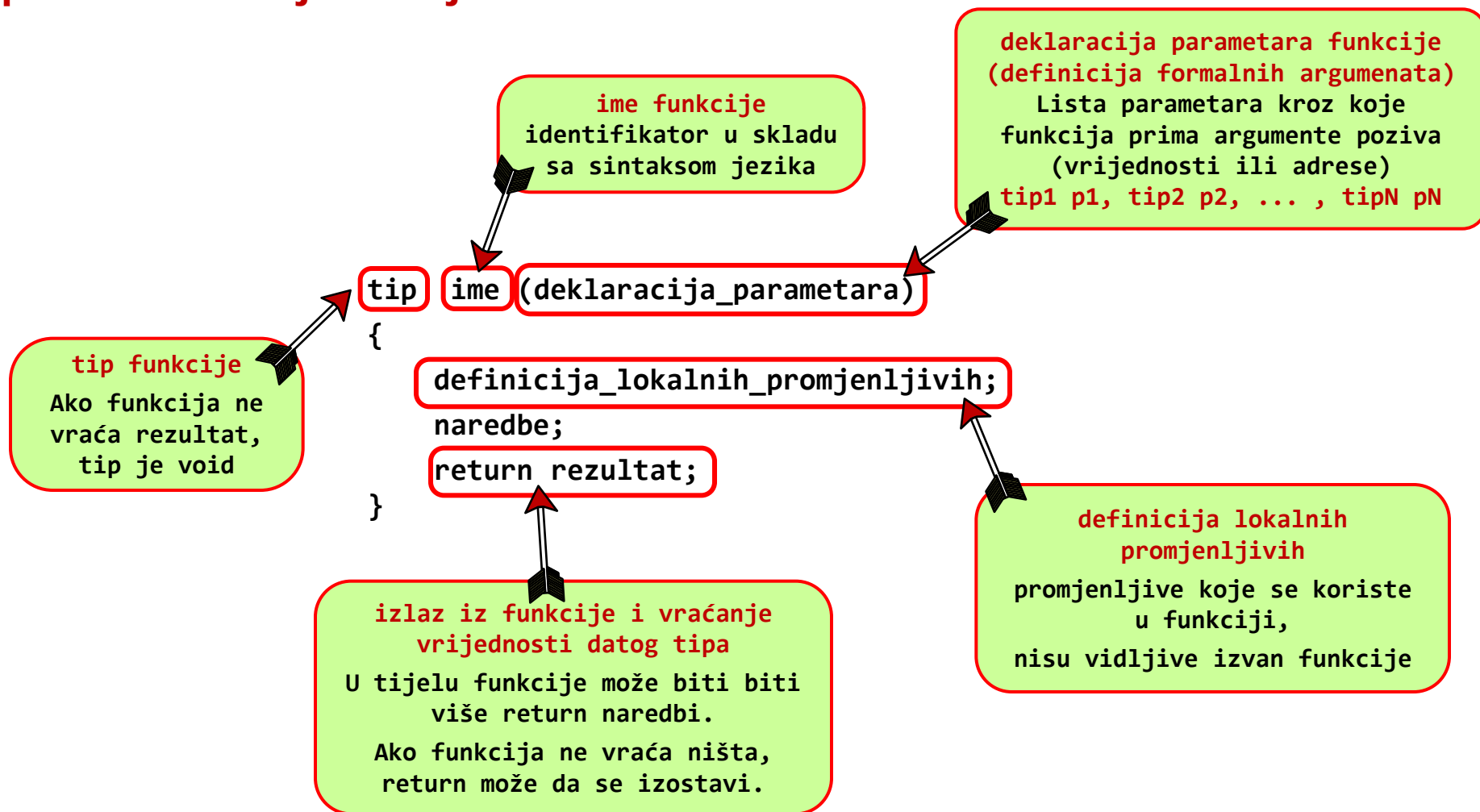
- Prilikom razvoja algoritma/programa mogu da se uoče logičke cjeline koje se više puta ponavljaju, najčešće nad različitim skupom podataka.
- Takve cjeline treba izdvojiti u zasebne programske cjeline – **potprograme**.
- Korištenjem potprograma:
 - povećava se preglednost programa,
 - smanjuje se veličina programa,
 - postiže se univerzalnost koda,
 - omogućava se lakše otklanjanje grešaka.
- U programskim jezicima obično se implementiraju:
 - proceduralni potprogrami (**procedure**)
 - funkcijski potprogrami (**funkcije**)
- Programski jezik C podržava samo funkcijske potprograme

Funkcije

- Funkcija je potprogram koji na osnovu određenog broja argumenata daje (vraća) jedan rezultat.
- Podatak koji funkcija vraća naziva se vrijednost funkcije.
- Pod tipom funkcije podrazumijeva se tip podatka koji vraća funkcija.
- Proceduralni potprogrami (u drugim programskim jezicima) mogu da vraćaju više podataka. Zbog toga, funkcije u C-u imaju mogućnost da osim rezultata (vrijednost funkcije) vrate više podataka kroz razmjenu argumenata.
- Efekat kad funkcija kroz argumente vraća rezultate naziva se bočni efekat.
- Funkcija može u potpunosti da se ponaša kao procedura, tj. može da ima samo bočne efekte.

Definicija funkcije u jeziku C

Opšti oblik definicije funkcije





Definicija funkcije u jeziku C

Primjer:

Definicija funkcije koja nema parametara i ne vraća rezultat

```
void f1()
{
    printf("Funkcija");
}
```

Primjer:

Definicija funkcije koja nema parametara i ne vraća rezultat

```
void f2()
{
    int i;
    for ( ; i ; i++ )
        printf("%d\n", i);
}
```

Primjer:

Definicija funkcije koja vraća kvadrat argumenta

```
double kvadrat(double a)
{
    return a*a;
}
```

Primjer:

Definicija funkcije koja računa i ispisuje faktorijel

```
void faktorijel(int n)
{
    int i=1, f=1;
    for ( ; i<=n ; f*=i++ );
    printf("%d!=%d\n", n, f);
}
```

Primjer:

Definicija funkcije koja računa i vraća faktorijel

```
long long faktorijel(int n)
{
    long long f=1;
    for ( ; n ; f*=n-- );
    return f;
}
```

Primjer:

Definicija funkcije koja provjerava da li je argument savršen broj

```
int savrsen(int n)
{
    int d=2, sd=1;
    for ( ; d<=n/2; d++ )
        if (n%d==0) sd+=d;
    return n==sd;
}
```



Poziv funkcije

Poziv funkcije

- Funkciju pozivamo navodeći njeno ime, a u malim zagradama redom argumente poziva (ako funkcija ima parametre)

Primjer:

Funkciju

```
void f1()
{
    printf("Funkcija");
}
```

pozivamo naredbom

f1();

Funkcija f1 nema parametara i ne očekuje da se prilikom poziva šalju argumenti.

Funkcija f1 ne vraća rezultat.

Funkcija koja ne vraća rezultat, ponaša se kao proceduralni potprogram (procedura).

Primjer:

Funkciju

```
void faktorijel(int n)
{
    int i=1, f=1;
    for ( ; i<=n ; f*=i++ );
    printf("%d!=%d\n", n, f);
}
```

pozivamo naredbom

faktorijel(argument);

Npr.

faktorijel(10);

faktorijel(x);

Funkcija faktorijel ima jedan parametar i očekuje da se prilikom poziva pošalje jedan podatak tipa int.

Funkcija faktorijel ne vraća rezultat.



Poziv funkcije

Primjer:

Funkciju

```
int savrsen(int n)
{
    int d=2, sd=1;
    for ( ; d<=n/2; d++ ) if (n%d==0) sd+=d;
    return n==sd;
}
```

pozivamo naredbom

`savrsen(argument);`

Npr.

```
for (int i=1; i<1000; i++)
    if (savrsen(i)) printf("%d ", i);
```

Za svako $i < 1000$ pozivamo funkciju `savrsen` i provjeravamo da li je to savršen broj.

Funkcija vraća rezultat koji koristimo u pozivajućem kodu (ako jeste savršen, biće ispisan).

Kao rezultat izvršavanja pozivajućeg koda, biće ispisani svi savršeni brojevi < 1000 .

Primjer:

Funkciju

```
void sviSavrseni(int a, int b)
{
    printf("Savrseni izmedju %d i %d:", a, b);
    for ( int i=a+1; i<b ; i++ )
        if (savrsen(i)) printf(" %d", i);
}
```

pozivamo naredbom

`sviSavrseni(arg1,arg2);`

Npr.

```
sviSavrseni(0,1000);
sviSavrseni(n,2*n);
```

Funkcija `sviSavrseni` ispisuje sve savršene brojeve iz intervala $<a,b>$.

Funkcija `sviSavrseni` koristi (poziva) funkciju `savrsen` za provjeru da li je neki broj iz intervala $<a,b>$ savršen.

Funkciju `sviSavrseni` pozivamo navodeći samo granice intervala.



Poziv funkcije

Primjer:

```
#include <stdio.h>

int savrsen(int n)
{
    int d=2, sd=1;
    for ( ; d<=n/2; d++ ) if (n%d==0) sd+=d;
    return n==sd;
}

void sviSavrseni(int a, int b)
{
    printf("Savrseni izmedju %d i %d: ", a, b);
    for ( int i=a+1; i<b ; i++ )
        if (savrsen(i)) printf("%d ", i);
}

int main()
{
    int x,y;
    do { printf("Od:"); scanf("%d", &x); } while (x<0);
    do { printf("Do:"); scanf("%d", &y); } while (y<x);
    sviSavrseni(x,y);
    return 0;
}
```

```
Od:1
Do:1000
Savrseni izmedju 1 i 1000: 6 28 496
```

Funkcija koju pozivamo mora biti definisana prije funkcije iz koje je pozivamo, tj.

pozvana funkcija (engl. **callee**) mora biti definisana prije pozivaoca (engl. **caller**).

U konkretnom primjeru:
main poziva funkciju sviSavrseni koja je definisana prije main,
sviSavrseni poziva funkciju savrsen koja je definisana prije sviSavrseni

Poziv funkcije

Funkcija koju pozivamo mora biti definisana prije funkcije iz koje je pozivamo.

Primjer:

```
#include <stdio.h>
void f(int n)
{
    printf("n=%d", n);
}
int main()
{
    f(2.3);
    return 0;
}
```

n=2

Prilikom poziva funkcije f zna se da f očekuje argument tipa int pa se 2.3 implicitno konvertuje u int

Ako pozvana funkcija nije definisana prije pozivaoca, tada se prilikom poziva neće izvršiti konverzije argumenata

Primjer:

```
#include <stdio.h>
int main()
{
    f(2.3);
    return 0;
}
void f(int n)
{
    printf("n=%d", n);
}
```

n=1717986918

Prilikom poziva funkcije f ne zna se da f očekuje argument tipa int pa se argument ne konvertuje u int, nego se šalje kao double

Primjer:

```
#include <stdio.h>
int main()
{
    f(2.5);
    return 0;
}
void f(int n)
{
    printf("n=%d", n);
}
```

n=0

Deklaracija funkcije

Deklaracija funkcije (prototip)

- Prototip ili deklaracija funkcije predstavlja njen "opis za spoljašnji svijet"
- Iz prototipa se vidi kako se komunicira s datom funkcijom, jer prototip sadrži:
 - tip funkcije,
 - ime funkcije,
 - tipove (a može i imena) argumenata.

Opšti oblik prototipa:

tip ime (tip1, tip2, ... , tipN);

Primjer:

```
void poruka ()  
{ printf("Zdravo!"); }
```

➡ void poruka ();

```
int zbir( int a, int b )  
{ return (a+b); }
```

➡ int zbir(int a, int b);

➡ int zbir(int, int);

```
long long fakt(int n)  
{  
    long long f=1;  
    for ( ; n ; f*=n-- );  
    return f;  
}
```

➡ long long fakt(int n);
long long fakt(int x);
long long fakt(int y);

➡ long long fakt(int);

Deklaracija funkcije

- Ponekad je nemoguće ili je veoma teško (npr. kad ima mnogo funkcija) **postići da svaka pozvana funkcija bude definisana prije pozivajuće funkcije**. Zbog toga se koriste prototipi.

Uobičajena je sljedeća struktura C programa:

pretprocesorske direktive
deklaracije funkcija
main
definicije funkcija

Primjer:

```
#include <stdio.h>

void f1() { f2(); }
void f2() { f1(); }

int main()
{
    f1(); f2();
    return 0;
}
```

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

```
void f1();
```

```
void f2();
```

```
int main()
```

```
{
```

```
    f1(); f2();
```

```
    return 0;
```

```
}
```

```
void f1() { f2(); }
```

```
void f2() { f1(); }
```

**Deklaracije
funkcija**

**Definicije
funkcija**



Deklaracija funkcije

Primjer:

```
#include <stdio.h>

int savrsen(int n)
{
    int d=2, sd=1;
    for ( ; d<=n/2; d++ ) if (n%d==0) sd+=d;
    return n==sd;
}

void sviSavrseni(int a, int b)
{
    printf("Savrseni izmedju %d i %d: ", a, b);
    for ( int i=a+1; i<b ; i++ )
        if (savrsen(i)) printf("%d ", i);
}

int main()
{
    int x,y;
    do
    {
        printf("Od:"); scanf("%d", &x);
        printf("Do:"); scanf("%d", &y);
    } while (x<0 || y<x);
    sviSavrseni(x,y);
    return 0;
}
```



```
#include <stdio.h>

int savrsen(int);
void sviSavrseni(int, int);

int main()
{
    int x,y;
    do
    {
        printf("Od:"); scanf("%d", &x);
        printf("Do:"); scanf("%d", &y);
    } while (x<0 || y<x);
    sviSavrseni(x,y);
    return 0;
}

int savrsen(int n)
{
    int d=2, sd=1;
    for ( ; d<=n/2; d++ ) if (n%d==0) sd+=d;
    return n==sd;
}

void sviSavrseni(int a, int b)
{
    printf("Savrseni izmedju %d i %d: ", a, b);
    for ( int i=a+1; i<b ; i++ )
        if (savrsen(i)) printf("%d ", i);
}
```



Prenos argumenata u funkciju

➤ Argumenti se prilikom poziva funkcije prenose **PO VRIJEDNOSTI**:

1. prilikom poziva funkcije šalje se vrijednost

- kao stvarni argument može da se navede: promjenljiva, konstanta ili izraz
- izračunava se vrijednost izraza i ta vrijednost se šalje u funkciju

2. funkcija prihvata proslijeđene vrijednosti u formalne argumente

- formalni i stvarni argumenti treba da se slažu u: broju, redoslijedu i tipu
- formalni argumenti (promjenljive) automatski nastaju prilikom ulaska u funkciju i preuzimaju proslijeđene vrijednosti

3. prilikom izlaska iz funkcije formalni argumenti automatski nestaju

- automatski nastaju i automatski nestaju pa se nazivaju **automatske promjenljive**

Prenos argumenata i vraćanje rezultata definisan je **POZIVNIM KONVENCIJAMA**.

Pozivne konvencije su predmet izučavanja u okviru kursa **PROGRAMIRANJE 2**

Ako se stvarni i formalni argumenti ne slažu po tipu, primjenjuju se standardna pravila za implicitnu konverziju (kao što je prethodno ilustrovano u primjerima za pozivanje funkcija)

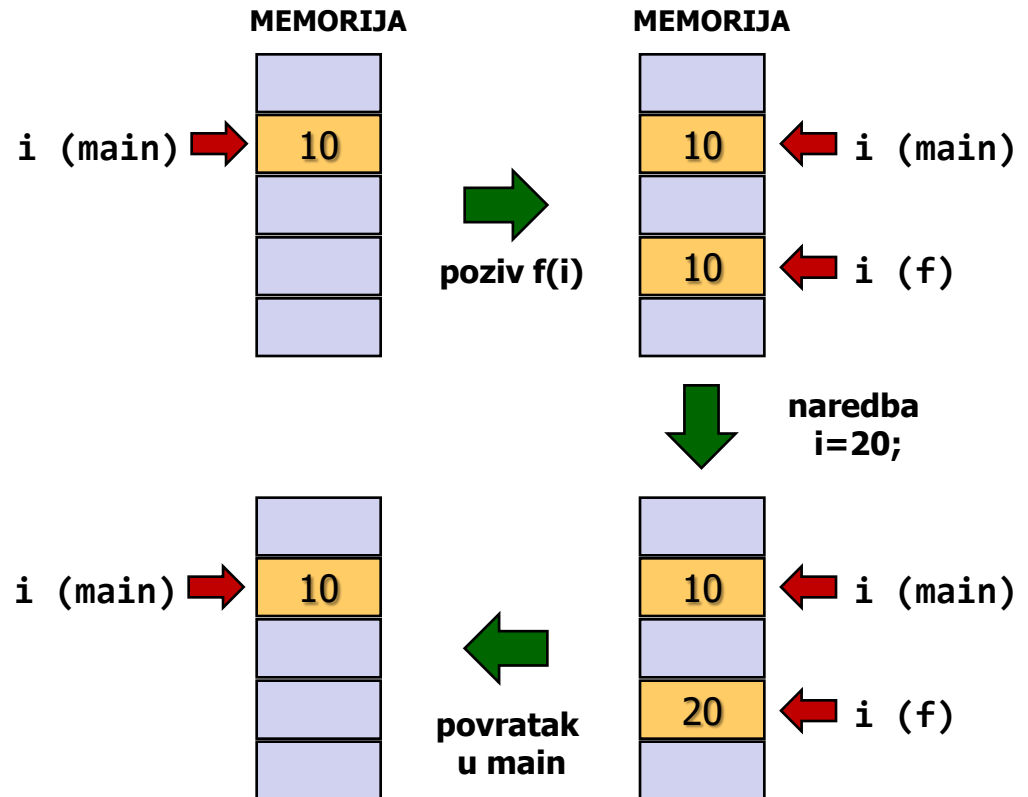
U funkciji se kreira slika (kopija) stvarnih argumenata i koriste se te kopije, a ne stvarne promjenljive iz glavnog programa, zato nakon izlaska iz funkcije promjenljive u glavnom programu ostaju nepromijenjene!!!

Prenos argumenata u funkciju

Primjer: Ilustracija prenosa argumenata po vrijednosti

```
#include <stdio.h>
void f(int);
int main()
{
    int i=10;
    printf("main: %d\n", i);
    f(i);
    printf("main: %d\n", i);
    return 0;
}
void f(int i)
{
    printf("f: %d\n", i);
    i=20;
    printf("f: %d\n", i);
}
```

```
main: 10
f: 10
f: 20
main: 10
```

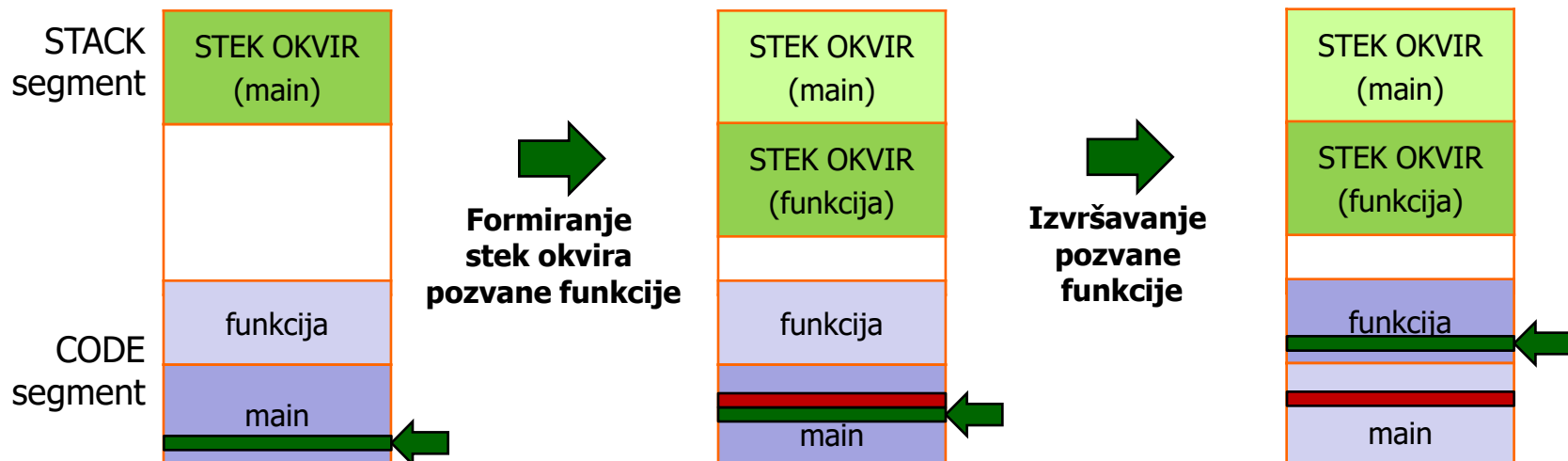


Parametar (formalni argument) "i" automatski nastaje prilikom ulaska u funkciju "f" i automatski nestaje prilikom izlaska iz funkcije "f"

Kontrola toka

Kontrola toka u programu sa funkcijama

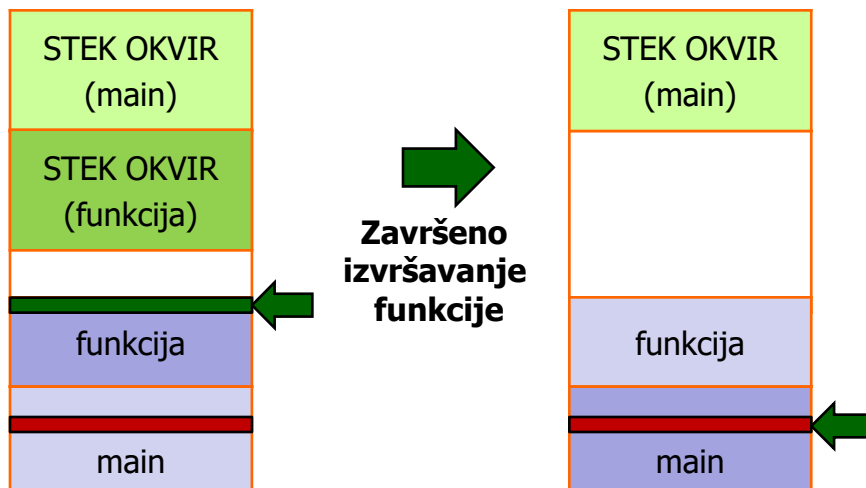
- **Kompletan programski kôd (preveden na mašinski jezik) tokom izvršavanja nalazi se u CODE SEGMENTU u memoriji.**
- Izvršavanje započinje funkcijom main
- **Na STEKU** (dio memorije za automatske promjenljive) **se formira STEK OKVIR** (engl. *stack frame*) **u kojem se nalaze sve lokalne promjenljive definisane u funkciji main**
- Prilikom poziva funkcije, na steku se formira **STEK OKVIR** koji pripada pozvanoj funkciji i u taj stek okvir se stavljaju:
 - **argumenti** (argumenti → parametri)
 - **adresa povratka** u pozivaocu (da bi se po završetku izvršavanja funkcije kontrola mogla vratiti u pozivaoca)
- Kontrola toka se prenosi na pozvanu funkciju i kreće izvršavanje pozvane funkcije



Kontrola toka

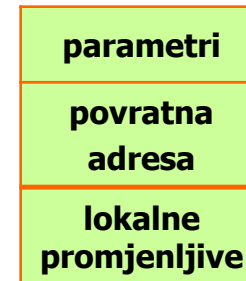
Kontrola toka u programu sa funkcijama (nastavak)

- Po završetku izvršavanja pozvane funkcije:
 - Iz stek okvira pozvane funkcije se uzima adresa povratka u pozivaocu**
 - "čisti" se stek (ukida se stek okvir)**
 - Kontrola se vraća pozivaocu i izvršavanje nastavlja od naredbe na kojoj je prekinuto izvršavanje**



Sadržaj stek okvira funkcije

- Stek okvir pozvane funkcije sadrži:
 - parametre**
 - adresu povratka u pozivaocu**
 - lokalne promjenljive**



Kontrola toka

Primjer:

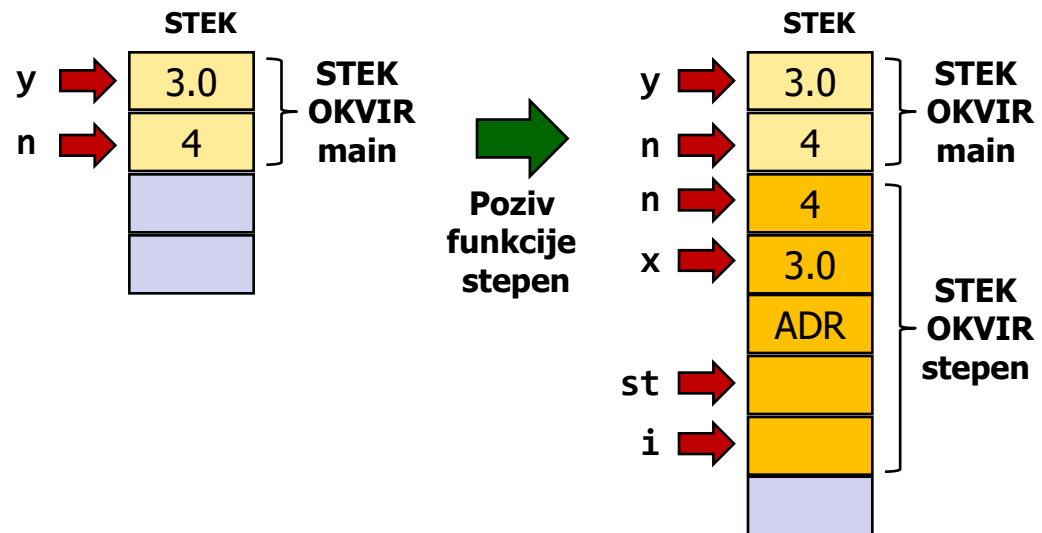
```
#include <stdio.h>

float stepen(float x, int n)
{
    float st=1;
    for (int i=1; i<=n; i++) st*=x;
    return st;
}

int main()
{
    float y;
    int n;
    printf("y="); scanf("%f", &y);
    do
    {
        printf("n="); scanf("%d", &n);
    } while (n<0);
    printf("%.2f^%d=%.4f\n", y,n,stepen(y,n));
    return 0;
}
```

Argumenti se u funkciju prenose po principu RTL
(Right-To-Left)

U konkretnom primjeru,
prvo se na stek stavlja n (4) pa y (3.0)



```
y=3
n=4
3.00^4=81.0000
```

Prenos niza u funkciju

Prenos niza vrši se putem adrese

1. prilikom poziva funkcije šalje se adresa niza

- kao stvarni argument navodi se ime niza (ime niza je početna adresa niza)

2. funkcija prihvata adresu niza

- formalni argument u definiciji funkcije mora da sadrži []
- broj elemenata niza ne mora da se navodi
- ne stvara se kopija niza nego se manipuliše stvarnim podacima

NIZ SE NE PRENOSI PO VRIJEDNOSTI, JER BI KOPIRANJE ČITAVOG NIZA BILO VREMENSKI I PROSTORNO ZAHTJEVNO!

PRILIKOM PRENOSA NIZA, NA STEK SE KOPIRA SAMO ADRESA (ADRESA SE PRENOSI PO VRIJEDNOSTI)

Primjer:

```
void print(int niz[], int n)
{
    for (int i=0; i<n; i++)
        printf("%d ", niz[i]);
}

int main()
{
    int a[] = {2,15,5,0,8};
    print(a,5);
    return 0;
}
```

Funkcija koja prima niz, treba da primi **adresu niza** i **stvarni broj elemenata u nizu**

Argumenti poziva su **adresa niza** i **stvarni broj elemenata u nizu**

Prenos niza u funkciju

Primjer:

```
#include <stdio.h>
void sort (int niz[], int n);
void pisi (int niz[], int n);
int main()
{
    int a[] = {2,15,5,0,8};
    int n=5;
    printf("Prije:"); pisi(a,n);
    sort(a,n);
    printf("Poslije:"); pisi(a,n);
    return 0;
}
void pisi (int niz[], int n)
{
    for (int i=0; i<n; i++)
        printf(" %d", niz[i]);
    printf("\n");
}
```

U funkciji se manipuliše stvarnim podacima, a ne kopijom niza. Zato funkcija može da promijeni niz, pa po povratku iz funkcije nemamo originalni nego modifikovani niz !!!

```
void sort (int niz[], int n)
{
    int i,j,rb,pom;
    for (i=0; i<n-1; i++)
    {
        for (rb=i, j=i+1; j<n; j++)
            if (niz[j]<niz[rb]) rb=j;
        if (rb!=i)
        {
            pom=niz[i];
            niz[i]=niz[rb];
            niz[rb]=pom;
        }
    }
}
```

```
Prije: 2 15 5 0 8
Poslije: 0 2 5 8 15
```

Prenos višedimenzionalnog niza u funkciju

Prenos nizova vrši se putem adrese

1. prilikom poziva funkcije šalje se adresa višedimenzionalnog niza

- kao stvarni argument navodi se ime niza (ime niza je početna adresa niza)

2. U deklaraciji formalnih argumenata

- prva dimenzija ne mora da se navede (može samo [])
- ostale dimenzije moraju da se navedu

NIZ SE NE PRENOSI PO VRIJEDNOSTI, JER BI KOPIRANJE ČITAVOG NIZA BILO VREMENSKI I PROSTORNO ZAHTJEVNO!

PRILIKOM PRENOSA NIZA, NA STEK SE KOPIRA SAMO ADRESA (ADRESA SE PRENOSI PO VRIJEDNOSTI)

Funkcija koja prima matricu, treba da primi adresu matrice i stvarni broj redova i stvarni broj kolona

Primjer:

```
void print(int mat[][10], int n, int m)
{
    for (int i=0; i<n; i++)
    {
        for (int j=0; j<m; j++)
            printf("%5d ", mat[i][j]);
        printf("\n");
    }
}

int main()
{
    int mat[10][10] = {{2,3,4},{5,6,7}};
    print(mat,2,3);
    return 0;
}
```

Argumenti poziva su adresa matrice i stvarni broj redova i stvarni broj kolona

Prenos višedimenzionalnog niza u funkciju

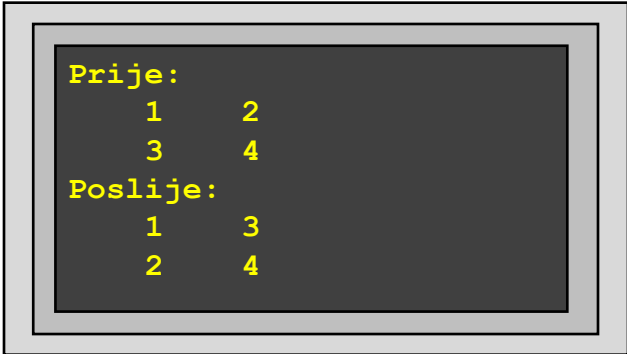
Primjer:

```
#include <stdio.h>

void tran (int mat[10][10], int n, int m);
void pisi (int mat[10][10], int n, int m);
int main()
{
    int a[10][10] = { {1,2},{3,4} };
    printf("Prije:\n"); pisi(a,2,2);
    tran(a,2,2);
    printf("Poslije:\n"); pisi(a,2,2);
    return 0;
}

void pisi (int mat[10][10], int n, int m)
{
    int i,j;
    for (i=0; i<n; i++)
    {
        for (j=0; j<m; j++)
            printf(" %4d", mat[i][j]);
        printf("\n");
    }
}
```

```
void tran (int mat[10][10], int n, int m)
{
    int i,j,pom;
    for (i=0; i<n; i++)
        for (j=i+1; j<m; j++)
        {
            pom=mat[i][j];
            mat[i][j]=mat[j][i];
            mat[j][i]=pom;
        }
}
```



```
Prije:
    1    2
    3    4
Poslije:
    1    3
    2    4
```

Prenos višedimenzionalnog niza u funkciju

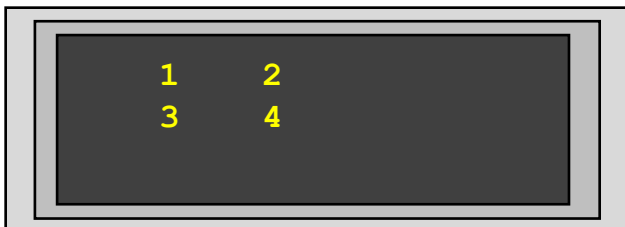
NEUSKLAĐENOST DIMENZIJA MATRICA MOŽE REZULTOVATI NEOČEKIVANIM REZULTATIMA !!!

PREPORUKA: U PROGRAMU TREBA KORISTITI MATRICE ISTIH DIMENZIJA

```
#include <stdio.h>

void pisi (int mat[4][4], int n, int m)
{
    for (int i=0; i<n; i++)
    {
        for (int j=0; j<m; j++)
            printf(" %4d", mat[i][j]);
        printf("\n");
    }
}

int main()
{
    int a[4][4] = { {1,2},{3,4} };
    pisi(a,2,2);
    return 0;
}
```

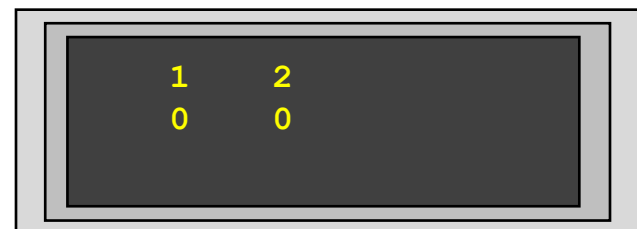


1	2		
3	4		

```
#include <stdio.h>

void pisi (int mat[2][2], int n, int m)
{
    for (int i=0; i<n; i++)
    {
        for (int j=0; j<m; j++)
            printf(" %4d", mat[i][j]);
        printf("\n");
    }
}

int main()
{
    int a[4][4] = { {1,2},{3,4} };
    pisi(a,2,2);
    return 0;
}
```



1	2		
0	0		



Prenos strukture u funkciju

Prenos strukture u funkciju

- **I struktura se, kao i drugi skalarni podaci, može prenijeti u funkciju po vrijednosti**
 - prilikom prenosa strukture po vrijednosti, u stek okvir pozvane funkcije stavlja se kopija strukture
 - pozvana funkcija koristi kopiju originalne strukture (formalni argument/parametar)
 - po izlasku iz funkcije, stek okvir se ukida i kopija automatski nestaje, a original u pozivaocu ostaje neizmijenjen

Primjer:

```
#include <stdio.h>

struct vrijeme { int hh, mm, ss; };

void print(struct vrijeme v)
{
    printf("%02d:%02d:%02d", v.hh, v.mm, v.ss);
}

int main()
{
    struct vrijeme t={12};
    print(t);
    return 0;
}
```

Alternativa sa definicijom tipa:

```
#include <stdio.h>

typedef struct vrijeme { int hh, mm, ss; } TIME;

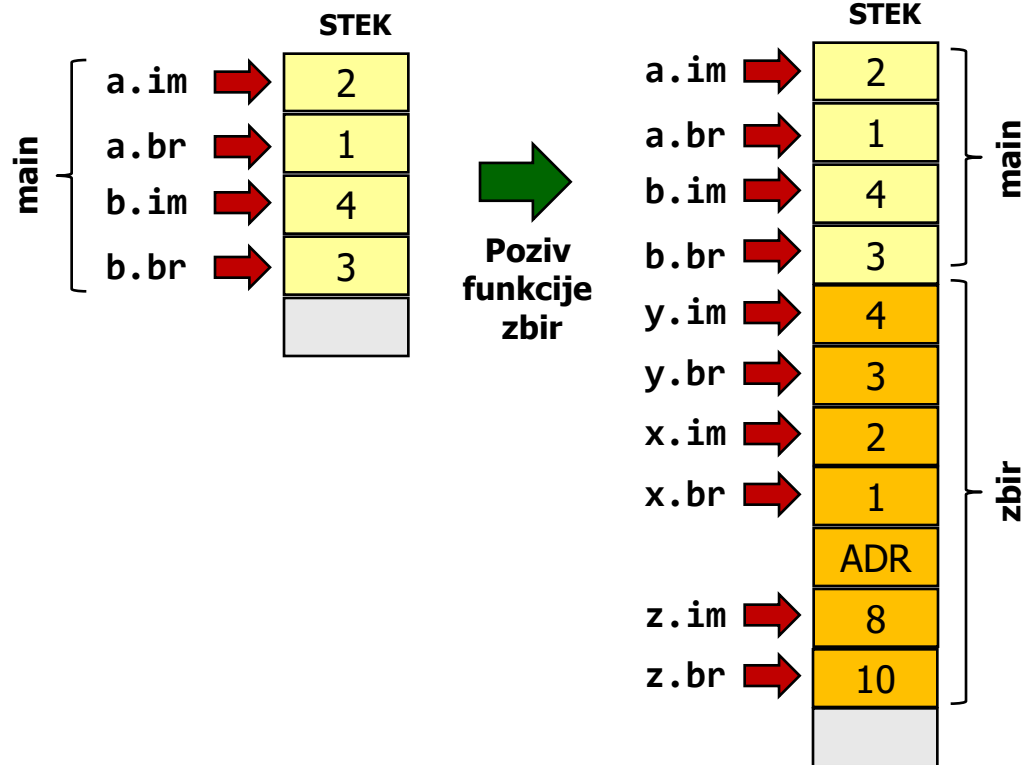
void print(TIME v)
{
    printf("%02d:%02d:%02d", v.hh, v.mm, v.ss);
}

int main()
{
    TIME t={12};
    print(t);
    return 0;
}
```

Prenos strukture u funkciju

Primjer: Manipulacija razlomcima

```
#include <stdio.h>
typedef struct { int br, im; } R;
void print(R r)
{
    printf("%d/%d", r.br, r.im);
}
R zbir(R x, R y)
{
    R z;
    z.im = x.im*y.im;
    z.br = x.br*y.im + y.br*x.im;
    return z;
}
int main()
{
    R a={1,2}, b={3,4};
    print(zbir(a,b));
    return 0;
}
```



10/8



Vraćanje rezultata funkcije

Vraćanje rezultata

- **Vraćanje rezultata definisano je pozivnim konvencijama** (u fokusu kursa PROGRAMIRANJE 2)

Funkcija može da vrati:

- bilo koji skalarni podatak (podatak bilo kojeg ugrađenog prostog tipa)

Npr.

```
int f1();  
float f2();
```

Funkcija može da vrati:

- podatke korisnički definisanih tipova (unija i struktura)

Npr.

```
struct Complex { float re, im; };  
struct Complex zbir(struct Complex x, struct Complex y);
```

Npr.

```
typedef struct Complex { float re, im; } COMPLEX;  
COMPLEX zbir(COMPLEX x, COMPLEX y);
```

Funkcija ne mora da vraća rezultat

- tip funkcije je **void**

Npr.

```
void f1();
```

Funkcija ne može da vrati:

- niz

Npr.

```
int[10] f(); ❌
```

```
int[10][10] f2(); ❌
```



Doseg identifikatora

Doseg identifikatora (engl. *identifier scope*)

- **Doseg (domen ili oblast definisanosti) identifikatora (promjenljive/funkcije) je područje programa u kojem je taj identifikator dostupan (vidljiv)**
- Doseg je određen načinom/mjestom definisanja identifikatora u programu
- U jeziku C važe **statička pravila za određivanje dosega** – **doseg može da se odredi analizom izvornog koda bez obzira na način izvršavanja programa**
- **U jeziku C postoje sljedeći nivoi dosega:**
 - **nivo datoteke**
(engl. *file level scope*)
 - **nivo bloka**
(engl. *block level scope*)
 - **nivo funkcije**
(engl. *function level scope*)
 - **nivo prototipa funkcije**
(engl. *function prototype scope*)

globalni
doseg

lokalni
doseg

➤ Najznačajniji nivoi dosega:

- **doseg nivoa datoteke**
ime važi od tačke uvođenja do kraja datoteke
- **doseg nivoa bloka**
ime važi od tačke uvođenja do kraja datog bloka

➤ Manje značajni nivoi dosega:

- **doseg nivoa funkcije**
ime važi u cijeloj funkciji u kojoj je uvedeno ovaj doseg imaju jedino labele
- **doseg nivoa prototipa funkcije**
ime važi u okviru prototipa funkcije ovaj doseg imaju samo parametri navedeni u prototipu funkcije – može da olakša razumijevanje i dokumentovanje koda.

Doseg identifikatora

Primjer:

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

```
int a; ← a je globalna promjenljiva / doseg nivoa datoteke
```

```
void f(int c); ← f je globalna funkcija / doseg nivoa datoteke      c ima doseg nivoa prototipa
```

```
int main()
{
    f();
    return 0;
}
```

```
void f(int c) ← f je globalna funkcija / doseg nivoa datoteke      c je lokalna promjenljiva / doseg nivoa bloka
{
```

```
    int d; ← d je lokalna promjenljiva / doseg nivoa bloka
```

```
    void g() { printf("zdravo"); } ← g je lokalna funkcija / doseg nivoa bloka
```

```
    for (d = 0; d < 3; d++)
    {
        int e; ← e je lokalna promjenljiva / doseg nivoa bloka
    }
```

```
    kraj: ← kraj je lokalna labela / doseg nivoa funkcije
```

```
}
```



Doseg identifikatora

➤ Jezik C dozvoljava “**konflikt identifikatora**”

- U programu može da postoji više identifikatora istog imena
- Ako su dosezi istoimenih identifikatora jedan u okviru drugog, tada identifikator u užoj oblasti dosega skriva identifikator u široj oblasti dosega

OSNOVNO PRAVILO DOSEGA IDENTIFIKATORA

Identifikator je dostupan u bloku u kojem je definisan, kao i u svim ugnježđenim blokovima, osim ako u njima nije maskiran drugim identifikatorom sa istim imenom!

Primjer:

```
#include <stdio.h>

int main()
{
    int x=1;
    printf("Prije bloka: %d\n", x);
    {
        int x=0;
        printf("U bloku: %d\n", x);
    }
    printf("Poslije bloka: %d\n", x);
    return 0;
}
```

```
Prije bloka: 1
U bloku: 0
Poslije bloka: 1
```

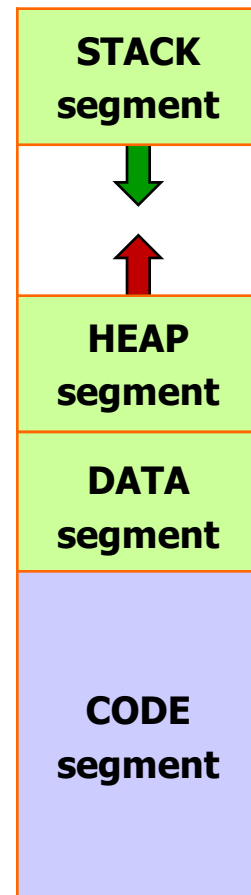
Životni vijek promjenljivih

Životni vijek (engl. *storage duration, lifetime*)

- **Životni vijek** promjenljive (vrijeme postojanja promjenljive) je dio vremena izvršavanja programa u kojem se garantuje da je za tu promjenljivu rezervisan dio memorije i da se ta promjenjliva može koristiti
- Životni vijek promjenljive je u bliskoj vezi sa dosegom odnosno identifikatora
- Životni vijek promjenljive određuje se na osnovu:
 - **pozicije u kodu** na kojoj je identifikator uveden i
 - **eksplicitnog korištenja kvalifikatora**:
 - **auto** (automatski životni vijek) ili
 - **static** (statički životni vijek)

➤ U jeziku C postoje sljedeći nivoi životnog vijeka:

- **automatski** (engl. *automatic*)
 - promjenljive koje automatski nastaju i automatski nestaju (npr. parametri)
 - smještene u stek segmentu
- **dinamički** (engl. *dynamic*)
 - promjenljive koje se alociraju i dealociraju na eksplicitan zahtjev programera
 - smještene u dinamičkoj zoni memoriji (*heap segment*)
- **statički** (engl. *static*)
 - promjenljiva dostupna tokom cijelog izvršavanja programa
 - smještene u segmentu podataka (*data segment*)





Životni vijek promjenljivih

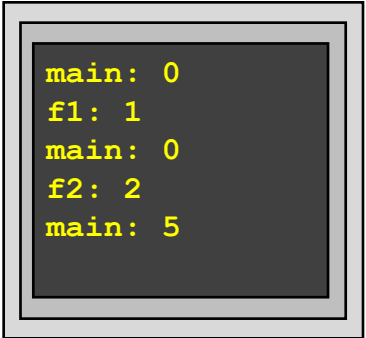
Automatski životni vijek

- **Automatske promjenljive automatski nastaju i automatski nestaju (prilikom ulaska/izlaska u/iz bloka)**
- **Tokom životnog vijeka smještene u stek segmentu**
- **Moraju biti lokalne**, ne mogu biti globalne
- **Početna vrijednost se ne podrazumijeva!** (neka vrijednost koja se "zatekla" na datim lokacijama na steku)
- Mogu da se koriste samo u bloku u kojem su definisane – to znači da isto ime možemo da koristimo nezavisno u različitim blokovima/funkcijama
- Prilikom definicije lokalne promjenljive, ne mora se eksplicitno navoditi da se radi o automatskoj promjenljivoj (podrazumijeva se da je lokalna promjenljiva automatska), ali može pomoću kvalifikatora **auto**

auto tip ime=vrijednost;

Primjer:

```
#include <stdio.h>
void f1()
{
    int i=1;
    printf("f1: %d\n", i);
}
void f2()
{
    int i=2;
    printf("f2: %d\n", i);
    i=10;
}
int main()
{
    int i=0;
    printf("main: %d\n", i);
    f1();
    printf("main: %d\n", i);
    i=5;
    f2();
    printf("main: %d\n", i);
    return 0;
}
```



```
main: 0
f1: 1
main: 0
f2: 2
main: 5
```



Životni vijek promjenljivih

Automatski životni vijek – registarske promjenljive

- **Registarske promjenljive su automatske promjenljive za koje je izražena želja da se drže u **registrima procesora**, a ne u memoriji, kako bi se dobilo na brzini**
- Poželjno je da registarske promjenljive budu one promjenljive koje se često koriste (npr. brojač)
- Ne garantuje se da će promjenljiva stvarno i biti registarska – to zavisi od konkretnog prevodioca, procesora, ukupnog broja registarskih promjenljivih do trenutka definicije date promjenljive itd.
- Broj registarskih promjenljivih je ograničen i malen
- **Registarske promjenljive definišu se uz pomoć kvalifikatora register**
register tip ime=vrijednost
Npr.

```
register int i=10;  
register int j;
```
- Pošto savremeni kompajleri tokom optimizacije koda mogu veoma dobro da odrede koje promjenljive ima smisla držati u registrima, ovaj kvalifikator se sve rjeđe koristi.



Životni vijek promjenljivih

Statički životni vijek

- Statičke promjenljive imaju trajan životni vijek – postoje tokom cijelog izvršavanja programa
 - Tokom životnog vijeka smještene u segmentu podataka (data segment)
 - Statički životni vijek imaju:
 - globalne promjenljive
 - lokalne promjenljive koje su u definiciji kvalifikovane kao statičke
- static tip ime=vrijednost**
- Početna vrijednost statičke promjenljive određuje se prije početka izvršavanja programa
 - uzima se vrijednost kojom je promjenljiva inicijalizovana
 - podrazumijeva se početna vrijednost nula ako promjenljiva nije inicijalizovana
 - Vrijednost lokalne statičke promjenljive ostaje sačuvana nakon izlaska iz funkcije do sljedećeg poziva date funkcije

Primjer (lokalne statičke promjenljive):

```
#include <stdio.h>

void f()
{
    static int s=1;
    auto int a=1;
    printf("static:%d  auto:%d\n", s, a);
    s++;
    a++;
}

int main()
{
    int i;
    for (i=1; i<=3; i++) f();
    return 0;
}
```

static:1	auto:1
static:2	auto:1
static:3	auto:1



Životni vijek promjenljivih

Statički životni vijek (globalne promjenljive)

- Globalne promjenljive definišu se izvan svih funkcija
- Ako se u definiciji ne navede početna vrijednost, podrazumijeva se nula!
- Globalna promjenljiva dostupna je u svim funkcijama koje su definisane nakon definicije date promjenljive
- Ako se u nekoj funkciji koristi neka globalna promjenljiva, to može da se eksplicitno naglasi deklaracijom korištenjem ključne riječi **extern** (iako ne postoji obaveza da se globalne promjenljive deklariraju u funkciji)
extern tip promjenljiva;
- Ako se u nekoj funkciji definiše lokalna promjenljiva sa istim imenom kao i neka globalna, tada ta lokalna promjenljiva maskira globalnu i globalnoj ne može da se pristupi.
- Globalne promjenljive omogućavaju da više funkcija manipuliše istim skupom podataka, pa omogućavaju efikasnu razmjenu podataka između funkcija (sve funkcije pristupaju istim memorijskim lokacijama, ne gubi se vrijeme na prenos podataka)
- Treba ih izbjegavati, jer se gubi univerzalnost funkcije i fleksibilnost primjene na različite setove podataka

Životni vijek promjenljivih

Primjer (manipulacija globalnom promjenljivom):

```
#include <stdio.h>
void f1 ()
{
    int i=1;
    printf("f1: %d\n", i);
}
int i;
void f2 ()
{
    extern int i;
    printf("f2: %d\n", i);
    i=10;
}
int main()
{
    printf("main: %d\n", i);
    f1();
    printf("main: %d\n", i);
    i=5;
    f2();
    printf("main: %d\n", i);
    return 0;
}
```

Globalna promjenljiva `i` nije vidljiva u funkciji `f1()` jer je definisana nakon funkcije. U funkciji `f1()` dostupna je lokalna promjenljiva `i`.

U funkciji `f2()` koristi se globalna promjenljiva `i`. Deklaracija `extern int i` mogla je da se izostavi

globalna
promjenljiva
`i=0`

```
main: 0
f1: 1
main: 0
f2: 5
main: 10
```