Programski jezik C

III deo

Polja struktura

• Polja struktura se formiraju na isti način kao i polja podataka elementarnih tipova.

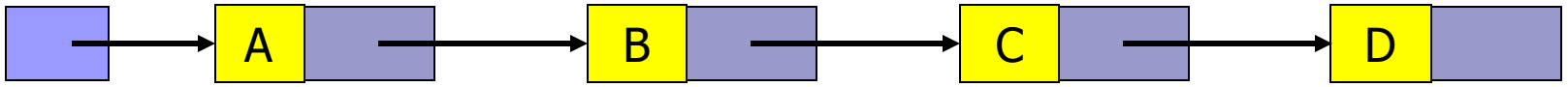
Primer:

```
struct tacka{
      int x;
      int y;
};
struct tacka tacke[50]; //deklaracija niza struktura
tacke[10].x = 5;        //pristup clanu strukture
tacke[10].y = 15;       // pristup clanu strukture
```

Samoreferencirajuće strukture

• Nije dozvoljeno da struktura sadrži instancu same sebe, ali je sasvim korektno da sadrži pokazivač na instancu same sebe.

Primer: (za strukturu lančane liste)




```
struct cvor{
        int sadrzaj;
        cvor *sledeći;
};
```

Typedef i strukture

• Često se u sprezi sa definisanjem strukture koristi i naredba za definisanje novog tipa.

typedef <poznati_tip> <ime_novog_tipa>;

Primer:

```
struct tacka{
      int x;
      int y;
};
typedef struct tacka Point; //definisanje novog tipa
Point pt1, pt2;          //deklaracija promenljivih
pt1.x = 15;        // pristup clanu strukture
```

Unije

• **Unija**, kao tip podataka, poseduje veoma mnogo sličnosti strukturi, ali dok je struktura agregatni tip podataka koji okuplja objekte (promenljive) čiji su tipovi (verovatno) različiti, unija je elementarni tip podatka od samo jednog elementa koji u nekom trenutku može da bude samo jedan od specificiranih tipova. Naravno, promenljiva je dovoljno velika da može da sadrži najveći od tipova članica unije.

• Opšta forma unije u C jeziku je:

```
union naziv_unije
{
        tip1 ime_promenljive_1;
        tip2 ime_promenljive_2;
       …
};
```

• Deklaracija promenljivih tipa unije i pristup elementima unije je potpuno isti kao i kod struktura.

Primer:

```
union naziv_unije
{
        int ival;
        float fval;
        char *sval;
} unija;    //definisanje unije i deklaracija pr.

unija.ival = 5; //pristup clanici unije
unija.fval = 15.55; //pristup clanici unije
/* unija sadrzi poslednju postavljenu vrednost */
```

Dinamička alokacija (zauzimanje) memorije

• Sve promenljive koje se deklarišu u programu su tzv. statičke promenljive u smislu da se za njih rezerviše potreban memorijski prostor pri pokretanju programa, a koji ostaje rezervisan do kraja izvršenja programa.

```
struct tacka tacke[50];
```

• Vrlo je često takav sistem neracionalan, tako de se često koristi način da se prostor za neke promenljive (tzv. dinamičke promenljive) rezerviše u toku izvršenja programa, a isto tako se može i "otkazati" rezervacija, pa se prostor oslobađa.

• Funkcije za upravljanje memorijom u C-u su:

- malloc()
- calloc()
- free()
- realloc()

```
#include <stdlib.h>

#include <malloc.h>
```

Nestandardne funkcije

Zauzimanje memorije

• Zauzimanje memorije se obično vrši pomoću funkcije malloc():

```
void *malloc(size_t size);
```

• Funkcija rezerviše prostor u memoriji veličine **size** bajtova, i kao rezultat vraća pokazivač na početak rezervisanog prostora.

• Ako se tražena količina memorije ne može rezervisati, kao rezultat se dobije NULL pointer (NULL pokazivač je definisana pokazivačka vrednost, ali ne označava stvarnu adresu; u stvari njena vrednost je 0).

Primer:

```
malloc(100);      //rezervise 100 bajtova u memoriji
```

Operator sizeof

- Ovaj operator vraća veličinu operanda u bajtovima.
- Na primer:

```
sizeof(int);
```

će vratiti 4 ili 2, u zavisnosti od kompajlera, jer je veličina podatka tipa int obično 4 ili 2 bajta.

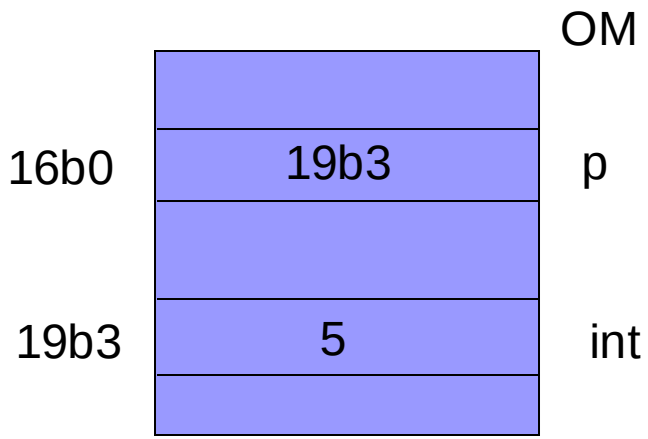
Primer:

```
malloc(sizeof(int)); //rezervise 4 (2) bajta u memoriji

double x;
int s1=sizeof(x);  /* s1 = 8 */
int s2=sizeof(int) /* s2 = 4 (2) */
```

• Pokazivač koji je rezultat funkcije malloc() je tipa **void**, što znači da nije definisano za koji tip podatka u njemu može biti smeštena adresa. Međutim, rezultat funkcije se može pridružiti nekom pokazivaču preko *cast* operatora:

```
int *p; //rezervise se prostor za p
p = (int*)malloc(sizeof(int)); //rezervise se
      //prostor za neku promenljivu int i ta adresa
      //se smesta u promenljivu p

*p = 5; //indirektna dodela vrednosti
```

kastovanje



- Inače, operator sizeof može da se primeni i na polja i na strukture.

Primer:

```
int ar[25];
int i = sizeof(ar);      // i = 25*4
int j = sizeof(ar[0]);   // j = 4
int len = i/j;           // len = 25

struct st {
      int i;
      char c;
};
int k = sizeof(st);      // k = 8 = 4+4
```

Zauzimanje memorije

• Zauzimanje memorije se može izvršiti i pomoću funkcije calloc():

```
void *calloc( size_t num, size_t size );
```

• Funkcija rezerviše prostor u memoriji (i inicijalizuje je na vrednost 0) za **num** elemenata veličine **size** bajtova, i kao rezultat vraća pokazivač na početak rezervisanog prostora.

• Ako se tražena količina memorije ne može rezervisati, kao rezultat se dobije NULL.

• Za oslobađanje memorije zauzete funkcijama malloc i calloc se koristi funkcija:

```
void free( void *memblock );
```

• Argument memblock je pokazivač na memorijski blok koji treba osloboditi. Funkcija ne vraća nikakvu vrednost.

Primer:

```
//Alociranje memorije za 40 long podatka
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>

void main( void )
{
long *buffer;
buffer = (long *)calloc( 40, sizeof( long ) );
if( buffer != NULL )
      printf( "Alocirano je 40 long intedzera\n" );
else
      printf( "Ne moze se alocirati memorija\n" );
free( buffer );//oslobadjanje memorije
}
```

Realociranje memorije

• Realociranje (ponovno zauzimanje memorije) se vrši pomoću funkcije realloc():

```
void *realloc( void *memblock, size_t size );
```

• Funkcija oslobađa rezervisani blok na koji pokazuje **memblock** i rezervise novi veličine **size** bajtova, i kao rezultat vraća pokazivač na početak rezervisanog prostora.

• Ako se tražena količina memorije ne može rezervisati, kao rezultat se dobije NULL pointer.

Najčešće greške

- "Viseći" pointeri

```
free(p);
...
*p = 10;
```

```
free(p);
p = NULL;
...
```

Pristup već oslobođenoj lokaciji !!!

- "Curenje" memorije (*memory leaks*).

```
p = NULL;
...
```

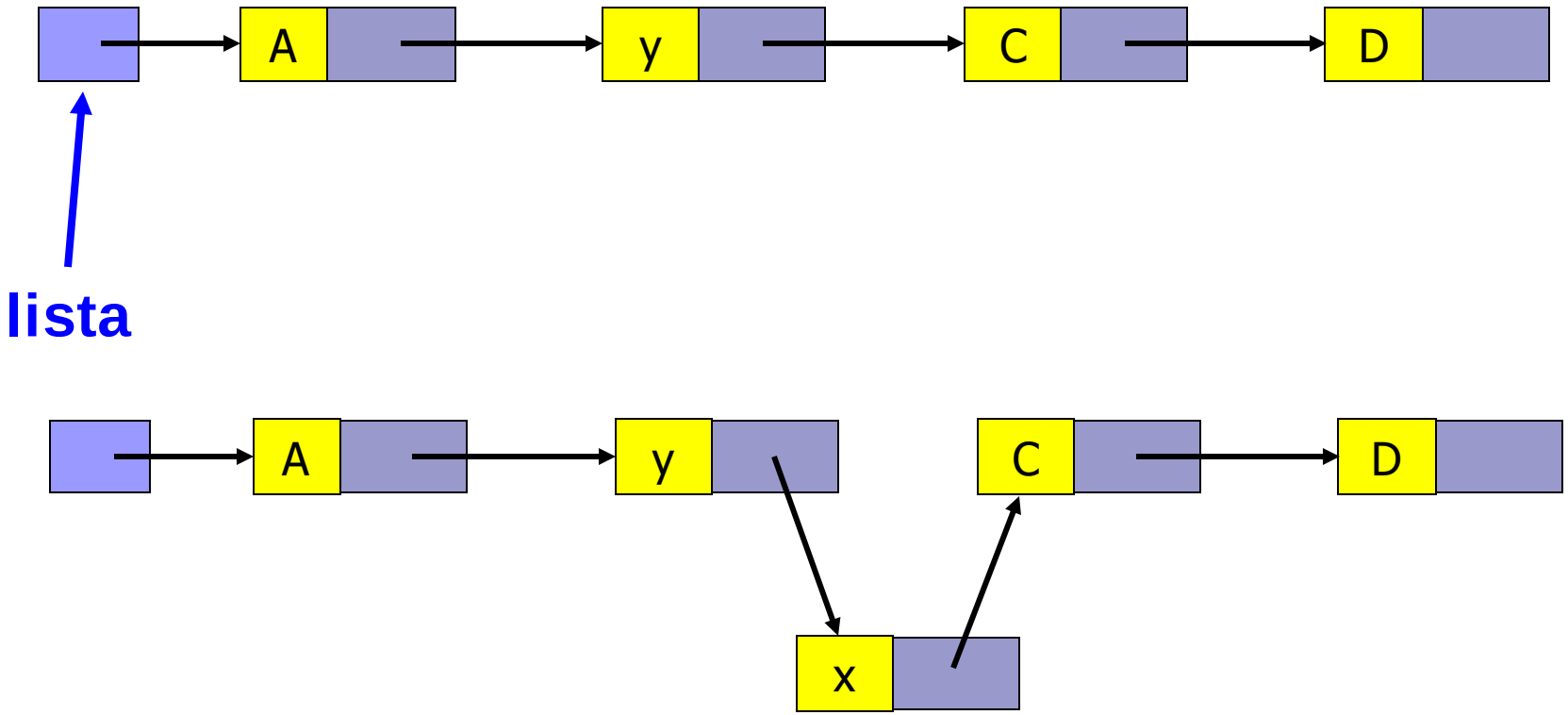
```
free(p);
p = NULL;
...
```

Nije oslobođena memorija !!!

Zadatak: Lančane liste

Napisati funkciju na C-u koja dodaje novi čvor sa vrednošću **x** u lančanu listu lista, odmah iza prvog čvora koji ima vrednost **y**.

Rešenje:



```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
struct cvor{
        int sadrzaj;
        cvor *sledeći;
};
void dodaj(struct cvor *lista, int x, int y){
struct cvor *p, *novi;
p = lista;
while (p->sadrzaj != y && p != NULL)
        p = p->sledeci;
if (p != NULL){
        novi = (struct cvor *)malloc(sizeof(struct cvor));
        novi->sadrzaj = x;
        novi->sledeci = p->sledeci;
        p->sledeci = &novi;
}
else
        printf("Ne postoji elemenat sa zadatim kljucem\n");
}
```

Preprocesorske direktive

• Preprocesorske direktive u C jeziku omogućavaju jednostavniji razvoj i modifikaciju programa i što je posebno bitno, omogućavaju veću prenosivost programa pisanih u C jeziku na različita hardverska i softverska okruženja.

• U okviru svakog C programskog prevodioca postoji preprocesor koji je njegov sastavni deo i koji omogućava prihvatanje specijalnih iskaza koji su pisani zajedno sa iskazima C jezika.

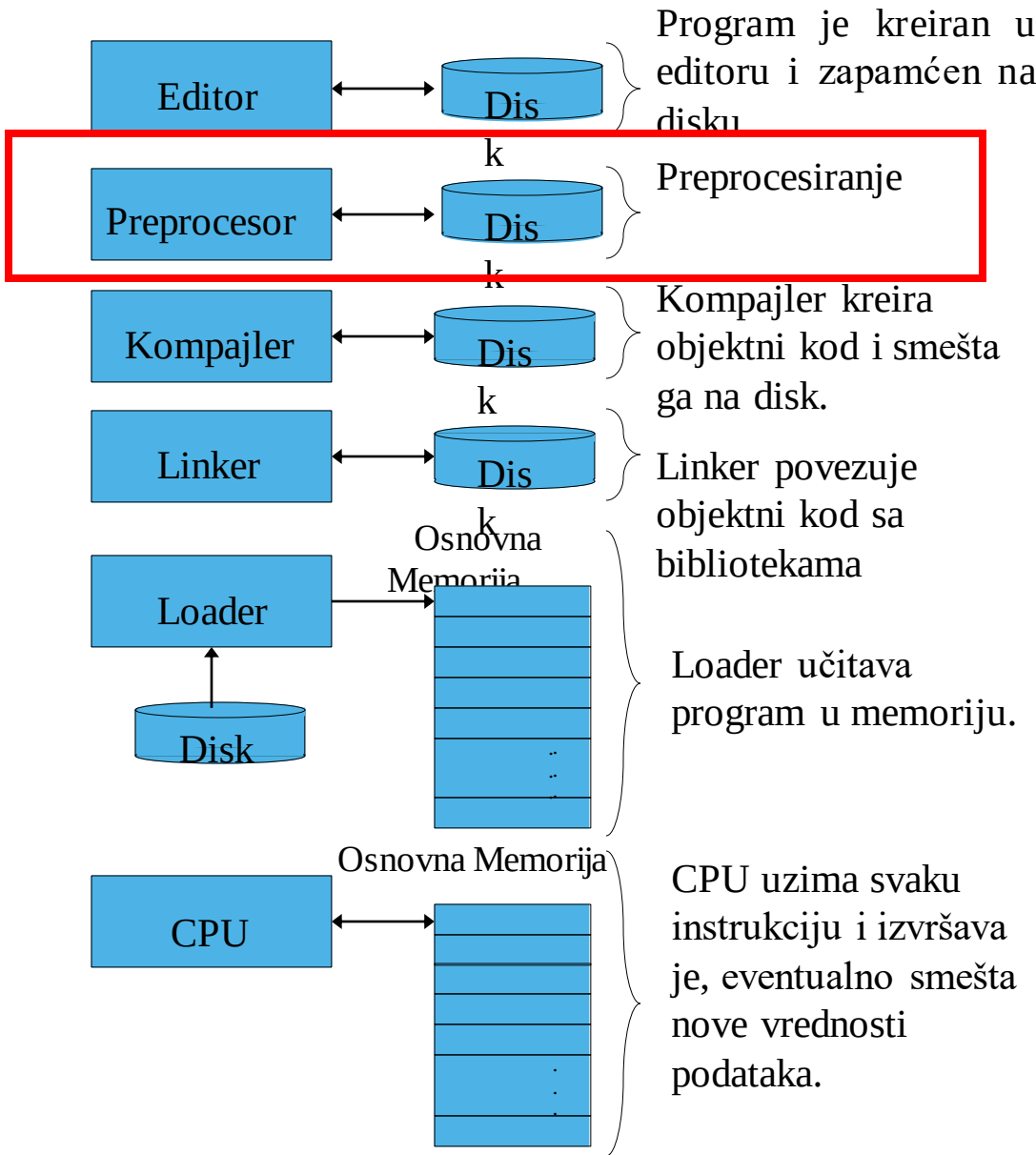
• C preprocesor vrši analizu teksta programa pre C prevodioca i omogućava identifikaciju preprocesorskih naredbi uključivanjem specijalnog karaktera #. Ovaj karakter mora biti prvi u preprocesorskoj naredbi jer je sintaksa iskaza preprocesora različita od sintakse iskaza C jezika.

• Preprocesorske direktive se obično navode na početku programa i ne završavaju se ;.

Razvoj C programa

Faze:

1. *Editovanje*
2. *Preprocesiranje*
3. *Kompilacija*
4. *Linkovanje*
5. *Učitavanje(Load)*
6. *Izvršenje*



Preprocesorske direktive

- U preprocesorske direktive spadaju:
  1. Direktive za definisanje simboličkih konstanti i makroa.
  2. Direktive za uključivanje datoteka.
  3. Direktive za uslovnu kompilaciju

Simboličke konstante i makroi

- Simboličke konstante i makroi se definišu korišćenjem preprocesorske direktive:

`#define`

Primer:

```
#define PI 3.141592
...
```

Primer:

```
#define TRUE 1     /* Nema tačke-zareza!!! */
#define FALSE 0
#define AND &&
#define OR ||
#define EQ ==

.................

game_over = TRUE;
while (i EQ j AND j < 3){...

.................
```

Zadatak:

Napisati funkciju na C-u koja izračunava vrednost PDV-a na osnovni iznos.

Rešenje:

```
#include <stdio.h>
#define PDV 0.20
void main()
{
 float glavnica;
 float porez;
 glavnica = 72.10;
 porez = glavnica * PDV;
 printf("Porez na %.2f iznosi %.2f\n",glavnica,porez);
}
```

Preprocesor prvo zamjenjuje sve simboličke konstante pre no što je program preveden, tako da nakon preprocesora (i pre prevođenja), program izgleda ovako:

```
#include <stdio.h>
#define PDV 0.20
void main()
{
 float glavnica;
 float porez;
 glavnica = 72.10;
 porez = glavnica * 0.20;
 printf("Porez na %.2f iznosi %.2f\n",glavnica,porez);
}
```

Mogućnost greške

```
#include <stdio.h>
#define PDV 0.20
void main()
{
 float glavnica;
 float porez;
 glavnica = 72.10;
 PDV = 0.15;
 porez = glavnica * PDV;
 printf("Porez na %.2f iznosi %.2f\n",glavnica,porez);
}
```

Simboličkoj konstanti se ne može dodeljivati vrednost !!!

Makroi sa argumentima

- **Makro** je definicija simbola u kojoj se javljaju jedan ili više argumenata.
- Na primer:

```
#define KVADRAT(x)  ((x)*(x))
```

definiše makro koji izračunava drugi stepen argumenta. Argument x se u programu zamenjuje stvarnim argumentom proizvoljnog tipa. Na primer:

```
KVADRAT(r+1)
```

postaje:

```
((r+1)*(r+1))
```

Mogućnost greške:

```
#define KVADRAT(x)  (x*x)  ->  (r+1*r+1)
```

Neispravno !!!

Primer:

```
#define KVADRAT(x) ((x)*(x))
#define KOCKA(x) (KVADRAT(x)*(x))
#define MAX(x,y) ((x)>(y)?(x):(y))
#define PRINT(x) printf("x je %d \n", x)
.................
int x = 50;
int y = 501;

int veci = MAX(x,y);
PRINT(veci);
int povrsina = KVADRAT(veci);
int zapremina = KOCKA(veci);

.................
```

Uklanjanje definicija

- Definicije simboličkih konstanti i makroa se mogu ukinuti direktivom:

```
#undef
```

Primer:

```
#define WIDTH 80
#define ADD( X, Y ) (X) + (Y)

. . .

#undef WIDTH
#undef ADD
```

Uključivanje datoteka

- Preprocesor u C jeziku omogućava da se u izvorni kod programa uključe kompletne datoteke i na taj način znatno ubrzava razvoj programa. Direktiva:

```
#include
```

uključuje u izvornu datoteku takozvane datoteke zaglavlja (*header file*) koji imaju ekstenziju **.h**: Postoje dva načina uključivanja datoteke zaglavlja u programu:

```
#include <ime_zaglavlja.h>
```

```
#include "ime_zaglavlja.h"
```

- Prvi način pretražuje sistemske direktorijume u potrazi za datotekama zaglavlja, dok drugi način pretražuje radni direktorijum na kome se nalazi program.
- U zaglavljima se mogu nalaziti, simbolička imena konstanti, definicije makroa i struktura i deklaracije promenljivih i funkcija.

Primer:

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


#include "myfile.h"
.................


.................
```

Uslovno prevođenje (kompilacija) programa

- Koncept uslovnog prevođenja se često koristi u programima namenjenim za prenos na više različitih hardverskih ili softverskih platformi.
- U okviru C preprocesora definisane su naredbe koje omogućavaju uključivanje i isključivanje određenih blokova naredbi tokom procesa prevođenja. Takve naredbe su preprocesorske direktive:

1. `#ifdef`
2. `#ifndef`
3. `#else`
4. `#if`
5. `#elif`
6. `#endif`

- Direktiva:

  ```
  #ifdef identif
  ```

  omogućava uključivanje koda koji sledi direktivu u proces prevođenja ukoliko je identifikator *identif* prethodno definisan.

- Direktiva

  ```
  #else
  ```

  omogućava alternativu za suprotni slučaj.

- Direktiva:

  ```
  #ifndef identif
  ```

  omogućava uključivanje koda koji sledi direktivu u proces prevođenja ukoliko identifikator *identif* nije prethodno definisan.

- Direktiva:

  `#if izraz`

  omogućava uključivanje koda koji sledi direktivu u proces prevođenja ukoliko je identifikator *izraz* različit od 0.

`#if defined(PERA)` je ekvivalentno sa `#ifdef PERA`

`#if !defined(PERA)` je ekvivalentno sa `#ifndef PERA`

- Direktiva

  `#elif`

  je ekvivalentna izrazu:

  `#else if`

- Direktiva:

  `#endif`

  zatvara sve prethodno navedene ***if*** direktive.

Primer:

```
#if SYSTEM == UNIX
     #define HDR "unix.h"
#elif SYSTEM == MSDOS
     #define HDR "dos.h"
#elif SYSTEM == MSWIN
     #define HDR "windows.h"
#endif
#include HDR
```

Primer:

```
#include <stdio.h>
#define PDV 0.18
#define PROVERA

void main()
{
 float glavnica;
 float porez;
 glavnica = 72.10;
      #ifdef PROVERA
            printf("Glavnica je %.2f \n",glavnica);
      #endif
 porez = glavnica * PDV;
 printf("Porez na %.2f iznosi %.2f\n",glavnica,porez);
}
```

Memorijske klase identifikatora

- Promenljive i funkcije u C-u imaju dva atributa: **memorijsku klasu** i **tip**.
- Memorijska klasa određuje lokaciju i vreme postojanja memorijskog bloka dodeljenog funkciji ili promenljivoj.
- Tip određuje značenje memorisane vrednosti u memorijskom bloku.
- Uobičajena podela identifikatora je na:

    1. Lokalne i
    2. Globalne

Lokalne promenljive

• Ove promenljive postoje samo unutar određene funkcije koja ih kreira. Nepoznate su drugim funkcijama i glavnom programu.

• Lokalne promenljive prestaju da postoje kada se izvrši funkcija koja ih je kreirala. Ponovno se kreiraju svaki put kada se funkcija poziva ili izvršava.

Globalne promenljive

• Ovim promenljivama može pristupiti svaka funkcija iz programa. Ne kreiraju se ponovno ako se funkcija ponovno poziva.

Primer:

```
#include <stdio.h>
int add_numbers( void ); /* ANSI prototip funkcije */
int value1, value2, value3;   // Ovo su globalne promenljive i može
                              // im pristupiti svaka funkcija
void main()
{
 int result;    //lokalna promenljiva
 value1 = 10;
 value2 = 20;
 value3 = 30;
 result = add_numbers();
 printf("Suma %d + %d + %d iznosi %d\n",
        value1, value2, value3, final_result);
 }
int add_numbers( void )
{
 int result;    //lokalna promenljiva
 result = value1 + value2 + value3; /* koristi globalne promenljive*/
 return result;
}
```

- Generalno, u C jeziku postoje četiri osnovne memorijske klase :

1. Automatska **(auto)**
2. Eksterna **(extern)**
3. Statička i **(static)**
4. Registarska **(register)**

- Promenljive deklarisane u funkciji ili na početku bloka su po definiciji automatske. Nastaju pri ulasku u blok ili funkciju i nestaju kada se blok ili funkcija završe. Deklaracije ovih promenljivih mogu opciono imati ključnu reč **auto**.

Primer:

```
/* ove deklaracije su ekvivalentne */
int x, y;
auto int x, y;
```

- Promenljive deklarisane u jednom modulu (fajlu) a koriste se u drugom modulu su eksterne. Deklaracije ovih promenljivih moraju imati ključnu reč **extern**.
- Ove promenljive omogućavaju modularno programiranje, odnosno pisanje programa korišćenjem više datoteka (fajlova). Svaka globalna promenljiva je i eksterna promenljiva. Da bi se ona videla iz drugog modula mora se koristiti **extern**.

Primer:

Moduo1.c

```
int spoljasnja;
void main ()
{
...
}
```

rezerviše memoriju

Moduo2.c

```
int fun1()
{
  extern int spoljasnja;
   spoljasnja = 5;
...
}
```

ne rezerviše memoriju

- Promenljive koje zadržavaju vrednost kada se blok ili funkcija gde su deklarisane završe su statičke. Deklaracije ovih promenljivih moraju imati ključnu reč **`static`**.

Primer:

```
#include <stdio.h>
void demo( void );
void demo( void )
{
        auto int avar = 0;
        static int svar = 0;
        printf("auto = %d, static = %d\n", avar, svar);
        ++avar;
        ++svar;
}
void main()
{
        int i = 0;
        while( i < 3 ) {
        demo();
        i++;
        }
}
```

auto = 0, static = 0

auto = 0, static = 1

auto = 0, static = 2

• C jezik omogućava programeru da utiče na efikasnost programa. Naime, ako program često koristi neku promenljivu, programer može sugerisati prevodiocu da tu promenljivu smesti u brze registre centralnog procesora. Ovakve promenljive se zovu registarske i u deklaraciji se mora koristiti ključna reč **`register`**.

• Naravno, prevodilac može i ignorisati ovakvu preporuku programera.

Primer:

```
/* deklaracija registarske promenljive */
register char x;
...
```