

# Uvod u programiranje

- predavanja -

prosinac 2019.

---

## 11. Memorijski razredi

# Vidljivost ili doseg varijable

- Vidljivost ili doseg varijable (*visibility, scope*)
  - dio (ili dijelovi) programa u kojem se do objekta (varijable) može pristupiti korištenjem identifikatora tog objekta (imena varijable)
  - identifikator može u nekim dijelovima programa biti prekriven (skriven) nekim drugim identifikatorom

# Primjer

```
int main(void) {
```

```
...
```

```
int a = 2;
```

```
printf("%d", a);
```

```
{
```

```
...
```

```
double a = 5.;
```

```
...
```

```
printf(" %d", a);
```

```
printf(" %lf", a);
```

```
...
```

```
}
```

```
printf(" %d", a);
```

```
}
```

vidljivost

vidljivost

U ovom dijelu programa do objekta a (cjelobrojne varijable a) nije moguće pristupiti pomoću identifikatora tog objekta

2 5.000000 2

- premda je posve legalno na ovaj način prekriti ime varijable, radi jasnoće programa bolje je izbjegavati tu mogućnost

# Primjer

- uočiti i objasniti razliku u rezultatu sljedećih odsječaka programa

```
int i = 5;
printf("%2d", i);
if (i == 5) {
    i = 6;
    printf("%2d", i);
}
{
    i = 7;
    printf("%2d", i);
}
printf("%2d", i);
```

5 6 7 7

```
int i = 5;
printf("%2d", i);
if (i == 5) {
    int i = 6;
    printf("%2d", i);
}
{
    int i = 7;
    printf("%2d", i);
}
printf("%2d", i);
```

5 6 7 5

# Trajnost varijable

- Trajnost varijable (*duration, lifetime*) je period tijekom izvršavanja programa u kojem varijabla (u memoriji) čuva svoju vrijednost

```
int i;  
for (i = 0; i < 2; ++i) {  
    ...  
    {  
        int m;  
        printf("%d\n", m);  
        m = 5;  
    }  
    ...  
}  
...
```

trajnost m

trajnost i

U oba koraka petlje ispisati će se vrijednost na koju se ne može računati jer varijabla *m* prestaje postojati (prestaje se čuvati njena vrijednost) kada završi izvršavanje bloka u kojem je definirana

# Lokalne i globalne varijable

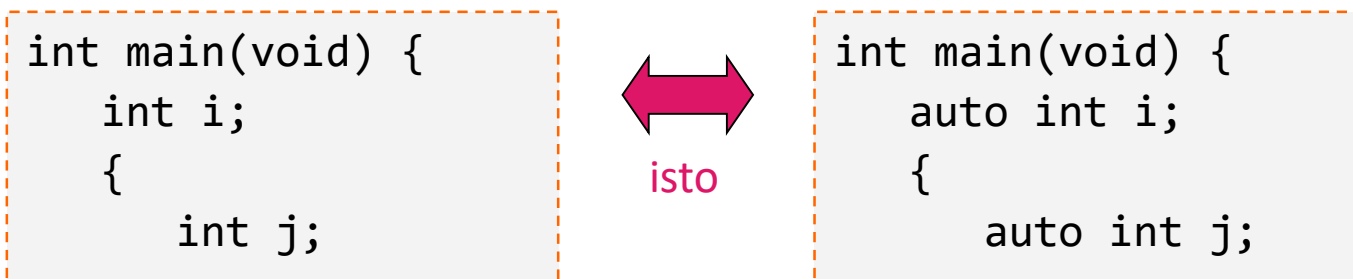
- *Lokalna varijabla* je varijabla čija je vidljivost ograničena na blok u kojem je definirana
  - taj blok uključuje i ugniježdene (podređene) blokove
- *Globalna varijabla* je varijabla čija se vidljivost proteže kroz više blokova
  - može biti vidljiva u više dijelova nekog modula (različitim blokovima), u jednom ili više modula ili čak u svim modulima

# Memorijski razredi

- Na vidljivost i trajnost varijabli može se utjecati mjestom u programu na kojem je varijabla definirana (unutar nekog bloka ili izvan svih blokova) i navođenjem (tijekom definicije varijable) jedne od specifikacija memorijskog razreda (*storage-class specifier*)
  - auto
  - register
  - static
  - extern

# Memorijski razred *auto*

- Automatska varijabla
  - može se definirati isključivo unutar bloka
  - vidljivosti i trajnost: od mjesta na kojem je definirana do kraja bloka u kojem je definirana
    - podsjetnik: vidljivost varijable može biti ograničena ako je njeno ime prekriveno definicijom neke druge istoimene varijable
  - podrazumijeva se, ako eksplicitno nije specificiran neki drugi memorijski razred, da je svaka varijabla definirana *unutar bloka* razreda *auto*



- parametri funkcije također imaju karakteristike automatskih varijabli



# Na kojem mjestu u programu definirati varijablu?

- Sve varijable na početku funkcije ili tek kad zatrebaju?
  - pitanje stila, ali definiranje varijable tek u dijelu programa u kojem će se ta varijabla koristiti može biti korisno radi uštede memorije ili radi povećanja razumljivosti koda

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

```
int main(void) {
```

```
    int izbor;
```

```
    scanf("%d", &izbor);
```

```
    if (izbor == 1) {
```

```
        int i, j;
```

```
        char matrica[1000][1000];
```

```
        ...
```

```
    } else {
```

```
        int i;
```

```
        double polje[100000];
```

```
        ...
```

```
    }
```

u ovom bloku rezervira se  
približno 1 MB memorije

u ovom bloku rezervira se  
približno 800 kB memorije

# Definicija kontrolne varijable u *for* petlji

- Kontrolne varijable petlje s poznatim brojem ponavljanja također se mogu definirati u *izrazu za inicijalizaciju u for petlji*
  - njihova vidljivost i trajnost su ograničene na tijelo petlje
  - dopušteni memorijski razredi su `auto` i `register`

```
int i;  
for (i = 1; i <= 10; ++i) {  
    printf("%d\n", i);  
}
```

// ako je varijabla `i` potrebna samo za petlju, tada može i ovako:

```
for (int i = 1; i <= 10; ++i) {  
    printf("%d\n", i);  
}
```

```
register int i, j;  
for (i = 1, j = 10;  
     i <= 10;  
     ++i, --j) {  
    printf("%d %d\n", i, j);  
}
```

// ako su varijable `i, j` potrebne samo za petlju, tada može i ovako:

```
for (register int i = 1, j = 10;  
     i <= 10;  
     ++i, --j) {  
    printf("%d %d\n", i, j);  
}
```

# Memorijski razred *register*

- Karakteristike jednake automatskim varijablama, uz dodatak
  - specifikacija `register` predstavlja preporuku prevodiocu da treba koristiti najbrži mogući pristup do sadržaja varijable
    - ako je moguće, sadržaj varijable će umjesto u memoriji biti smješten u nekom od registara procesora (CPU)
  - nad varijablom razreda `register` nije primjenjiv adresni operator

```
int main(void) {  
    register unsigned int i;  
    register unsigned long long fact = 1;  
    for (i = 2; i < 15; ++i) {  
        fact *= i;  
    }  
    unsigned int *pok;  
    pok = &i;  
    ...  
}
```

Nije dopušteno

# Memorijski razred *static*

- Statičke varijable
  - trajnost varijable: od početka do završetka izvršavanja programa
  - vidljivost
    - ako je definirana unutar bloka: od mjesta na kojem je definirana do kraja bloka u kojem je definirana
    - ako je definirana izvan (svakog) bloka: u cijelom modulu
  - statičku varijablu treba definirati
    - *unutar funkcije* ako varijabla treba biti vidljiva samo unutar te funkcije, a istovremeno je potrebno sačuvati vrijednost varijable i nakon završetka izvršavanja te funkcije
    - *izvan funkcije* ako istu varijablu koristi nekoliko funkcija unutar istog modula
  - ako varijabla nije eksplicitno inicijalizirana tijekom definicije, automatski se inicijalizira na numeričku vrijednost nula
  - ako je varijabla eksplicitno inicijalizirana tijekom definicije, inicijalizacija se izvršava samo jednom, na početku izvršavanja programa

# Primjer, vidljivost varijabli

```
static int i;
...
void f1(void) {
    static int a;
    int i;
    ...
}
...
void f2(void) {
    static int b;
    ...
}
...
```

The diagram illustrates the scope of variables in the provided code. Vertical red lines represent the boundaries of the scope for each variable. Arrows point from the variable declarations to their respective scope boundaries:

- `static int i;` has a scope that extends to the end of the program (indicated by the rightmost red line).
- `static int a;` (inside `f1`) has a scope that extends to the end of the function `f1` (indicated by the second red line).
- `int i;` (inside `f1`) has a scope that extends to the end of the function `f1` (indicated by the second red line).
- `static int b;` (inside `f2`) has a scope that extends to the end of the function `f2` (indicated by the third red line).

# Primjer, trajnost varijabli

```
static int i;
```

```
...
```

```
void f1(void) {  
    static int a;  
    int i;  
    ...  
}
```

```
...
```

```
void f2(void) {  
    static int b;  
    ...  
}
```

```
...
```

# Primjer

- Što će se ispisati izvršavanjem sljedećeg programa?

```
#include <stdio.h>
static int a;

void f1(int param) {
    static int b;
    printf("%d %d %d\n", param, a++, b++);
}

void f2(void) {
    a = 50;
    f1(2);
}

int main(void) {
    f1(1);
    f2();
    f1(3);
    ...
}
```

iz f2 se može pristupiti do varijable a, ali ne i do varijable b

1 0 0  
2 50 1  
3 51 2

# Primjer

- Što će se ispisati izvršavanjem sljedećeg programa?

```
#include <stdio.h>

void f1(void) {
    static int a = 10;
    printf("%d\n", a);
    ++a;
}

int main(void) {
    f1();
    f1();
    ...
}
```

inicijalizira se samo jednom, a ne u svakom pozivu funkcije

10

11



# Primjer

- Programski zadatak
  - Linearni kongruentni generator je generator niza pseudoslučajnih cijelih brojeva. Član niza pseudoslučajnih brojeva  $x_{i+1}$  se izračunava na temelju prethodnog člana niza  $x_i$ .

$$x_{i+1} = (A \cdot x_i + C) \bmod M$$

- Cjelobrojne konstante  $A$ ,  $C$  i  $M$  se mogu definirati unaprijed (npr. kao simboličke konstante), a član niza  $x_0$ , tj. početni član niza pseudoslučajnih brojeva, koji se naziva sjeme ili *seed*, generatoru se zadaje neposredno prije početka generiranja niza pseudoslučajnih brojeva (inicijalizacija generatora pseudoslučajnih brojeva).

# Primjer (nastavak)

- Npr. ako se koriste sljedeće vrijednosti simboličkih konstanti:

```
#define A    9001
#define C 2531011
#define M    32717
```

te se generator inicijalizira na početni član  $x_0 = 100$ , daljnji članovi niza će biti:

$$x_1 = (9001 \cdot 100 + 2531011) \bmod 32717 = 28543$$

$$x_2 = (9001 \cdot 28543 + 2531011) \bmod 32717 = 744$$

$$x_3 = (9001 \cdot 744 + 2531011) \bmod 32717 = 1561$$

$$x_4 = (9001 \cdot 1561 + 2531011) \bmod 32717 = 26770$$

...

# Primjer (nastavak)

- U modulu `rand.c` napisati funkciju (prikazan je prototip)

```
void setSeed(int seed);
```

kojom se postavlja početni član za linearni kongruentni generator, te funkciju (prikazan je prototip)

```
int getRand(void);
```

koja kod svakog poziva vraća sljedeći član niza,  $x_1, x_2, \dots$

- Koristiti sljedeće konstante:

```
A = 9001; C = 2531011; M = 32717
```

- U glavnom programu početni član postaviti (tj. inicijalizirati generator) na  $x_0 = 100$  te na zaslon ispisati članove niza pseudoslučajnih brojeva  $x_1 \dots x_{10}$ . Zatim generator inicijalizirati na  $x_0 = 51$ , te ispisati članove niza  $x_1 \dots x_5$ .

# Primjer (nastavak)

```
void setSeed(int seed);  
int getRand(void);
```

**rand.h**

```
#include "rand.h"
```

**rand.c**

```
#define A    9001
```

```
#define C 2531011
```

```
#define M    32717
```

```
static int clan;
```

```
void setSeed(int seed) {  
    clan = seed;  
}
```

```
int getRand(void) {  
    clan = (A * clan + C) % M;  
    return clan;  
}
```

# Primjer (nastavak)

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

int main(void) {
    int i;
    printf("Postavi na 100:\n");
    setSeed(100);
    for (i = 1; i <= 10; ++i)
        printf("%5d\n", getRand());

    printf("Postavi na 51:\n");
    setSeed(51);
    for (i = 1; i <= 5; ++i)
        printf("%5d\n", getRand());
    return 0;
}
```

glavni.c

```
Postavi na 100:
28543
 744
1561
26770
 7867
23081
10933
 6999
29576
 7149
Postavi na 51:
12815
32192
30242
14604
 5500
```

# Definicija i deklaracija

- Definicijom varijable određuje se ime i tip varijable te se rezervira područje u memoriji u kojem će varijabla biti pohranjena
  - definicija iste varijable smije se u jednom programu pojaviti samo jednom
- Deklaracija varijable je uputa (objava) prevodiocu: za deklarirano ime varijable negdje u programu, u istom ili nekom drugom modulu, postoji (tj. negdje je definirana) varijabla s istim takvim imenom, na koje se ovo deklarirano ime odnosi
  - ime varijable (identifikator) deklaracijom povezujemo sa stvarnim objektom (koji je negdje drugdje definiran)
  - deklaracija iste varijable može se pojaviti više puta u istom programu

# Memorijski razred *extern*

- Eksterne varijable
  - moraju biti **definirane** u **jednom** modulu  
*tip ime\_varijable;*
  - mogu biti **deklarirane** u više modula  
**extern** *tip ime\_varijable;*
  - trajnost varijable: od početka do završetka izvršavanja programa
  - vidljivost
    - ako je ***deklarirana ili definirana*** izvan funkcije: od mjesta deklaracije ili definicije do kraja modula
    - ako je ***deklarirana*** unutar bloka: od mjesta na kojem je deklarirana do kraja bloka u kojem je deklarirana
  - ako varijabla nije eksplicitno inicijalizirana tijekom definicije, automatski se postavlja na numeričku vrijednost nula

# Memorijski razred *extern*

- Varijabla definirana izvan svih blokova (uobičajeno na početku modula) i koja nema specifikaciju `static`, je eksterna varijabla.
  - npr. uz pretpostavku da se program sastoji od samo dva modula

```
int var1;  
  
void f1() {  
    ...  
    ...  
}  
  
void f2() {  
    ...  
    ...  
}
```

trajnost var1

trajnost var2

modul1.c

vidljivost var1

```
int var2;  
  
void f3() {  
    ...  
    ...  
}  
  
void f4() {  
    ...  
  
    ...  
}
```

trajnost var1

trajnost var2

modul2.c

vidljivost var2



# Memorijski razred *extern*

- Eksterna varijabla **definirana** u jednom modulu može postati vidljiva u drugom modulu ako je u drugom modulu **deklariramo**
  - deklaracija se piše sa specifikacijom `extern`

```
int var1;

extern int var2;
void f1() {
    ...
}

void f2() {
    ...
}
```

modul1.c

vidljivost var2

vidljivost var1

```
int var2;

void f3() {
    ...
}

void f4() {
    ...
    extern int var1;
    ...
}
```

modul2.c

vidljivost var2

vidljivost var1

# Česte pogreške

- Mogućnost globalnog pristupa varijablama, što znači da se sadržaj varijabli može i pročitati i promijeniti iz bilo kojeg mjesta u modulu ili programu, neiskusne programere navodi na *zloupotrebu* statičkih i eksternih varijabli
  - zašto se "mučiti" s prijenosom argumenata u funkcije, kad se mogu definirati statičke ili eksterne varijable kojima se može pristupati i iz pozivajuće razine programa i iz funkcije?
- Statičke i eksterne varijable treba koristiti samo onda kada za njihovo korištenje postoje opravdani razlozi (npr. za čuvanje vrijednosti zadnjeg člana niza u generatoru niza pseudoslučajnih brojeva)

# Memorijski razredi

Zašto se eksterne i statičke varijable ne smiju koristiti za "prijenos" argumenata i povratak rezultata funkcija?

# Primjer

- Ispravan način realizacije funkcije pow

```
double pow(double x, double y);
```

**math.h**

```
double pow(double x, double y) {  
    /* programski kod za izracunavanje "x na y" */  
    ...  
    return konacniRezultat;  
}
```

**math.c**

# Primjer (nastavak)

- Kako pomoću ispravne implementacije funkcije pow izračunati  $x^{(y^z)}$

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

int main(void) {
    double x, y, z;
    scanf("%lf %lf %lf", &x, &y, &z);
    printf("%lf na (%lf na %lf) = %lf\n", x, y, z,
           pow(x, pow(y, z)));
    return 0;
}
```

prog.c

# Primjer (nastavak)

- Neprihvatljiv način implementacije funkcije pow

```
/* deklaracije eksternih varijabli koje ce se koristiti za      math.h
   "prijenos" argumenata i rezultata */
extern double powResult, xPow, yPow;
void pow(void);
```

```
/* definicije eksternih varijabli za "prijenos" argumenata      math.c
   i rezultata*/
double powResult, xPow, yPow;
void pow(void) {
    /* programski kod za izracunavanje "xPow na yPow" */
    ...
    powResult = ...; /* konacni rezultat spremi u powResult */
    return;
}
```

# Primjer (nastavak)

- Kako pomoću neispravne implementacije funkcije pow izračunati  $x^{(y^z)}$

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

int main(void) {
    double x, y, z;
    scanf("%lf %lf %lf", &x, &y, &z);
    xPow = y;
    yPow = z;
    pow();
    yPow = powResult;
    xPow = x;
    pow();
    printf("%lf na (%lf na %lf) = %lf\n", x, y, z, powResult);
    return 0;
}
```

prog.c

Mora se paziti čak i na to da se niti jedna lokalna ili statička varijable ne nazove xPow, yPow ili powResult, jer su ta imena "rezervirana" isključivo za "prijenos" argumenata i rezultata funkcije pow.