Programiranje I

Funkcije

Tip void

- Funkcija koja ne vraća vrijednost deklariše se kao void.
- Promjenljiva koja se deklariše kao void nema vrijednost i gotovo se nikad ne koristi.
- Funkcija koja se deklariše da vraća void * vraća pokazivač na neodređeni tip podatka.
- Promjenljiva koja se deklariše kao pokazivač na void, tj. void *b, pokazuje na promjenljivu koja je nekog od mogućih tipova podataka. Prije upotrebe ovog pokazivača, potrebno ga je konvertovati u konkretan tip.
- Ovo će biti jasnije kada budemo učili dinamičku alokaciju memorije.

Funkcija – Neki detalji

- Unutar funkcije se mogu deklarisati pomoćne promjenljive, kao u funkciji main. Tako se naša funkcija zbir1 mogla definisati kao:

```
int zbir1(int x, int y) {
        int zbir = x + y;
        return zbir;
}
```

Promjenljiva zbir se nakon završetka funkcije dealocira – briše iz memorije.

- Funkcije tipa void mogu se završiti svojom posljednjom naredbom ili im se izvršavanje može prekinuti naredbom return, bez argumenta nakon return.

Parametri i argumenti funkcije

- Parametri funkcije su promjenljive koje se navode u listi argumenata u zaglavlju, dok su argumenti vrijednosti koje se prosljeđuju funkciji iz drugih funkcija.
- Prilikom prosljeđivanja, vrijednost argumenata se kopira u parametre!

```
int main() {
   ...
   a = fun(x, 3);
   ...
}
```

Argumenti

```
int fun(int x, int y) {
   ...
}
```

Parametri

Ako u funkciji piše x++, ta promjena ni na koji način ne utiče na originalnu vrijednost argumenta iz pozivajuće funkcije, jer su to različite promjenljive!

Prototipovi funkcija

- Da bismo osigurali ispravan poziv funkcije, tj. da se ne dozvoli poziv sa neočekivanim argumentima, prije definisanja funkcije (i prije funkcije main) se navodi **prototip funkcije**.
- Prototip liči na zaglavlje, s tim što se ne moraju navoditi imena argumenata. Prototip se završava sa ;
- Primjer prototipa:

```
int zbir(int, int);   // ponekad se navode imena parametara
                      // da bi se objasnilo njihovo značenje
```

Prototip – Upotreba

```
#include <stdio.h>

int fun(int);

int main() {
    printf("%d\n", fun());
    return 0;
}

int fun(int n) {
    if (n == 0)
        return 1;
    else
        return n * fun(n - 1);
}
```

Prototip funkcije

Pokušaj poziva funkcije **fun** bez argumenata dovodi do greške, jer ta funkcija očekuje za argument cio broj.

Bez prototipa se ne vrši provjera prilikom poziva funkcije, pa poziv može rezultovati u besmislenim rezultatima ili „pucanju“ programa.

U slučaju da nije naveden prototip, kompajler će pretpostaviti deklaraciju. U liniji prvog poziva ćemo dobiti upozorenje **implicit declaration** o tome. Drugo upozorenje će biti tipa **conflicting types** i javiće se na mjestu definicije funkcije.

Poziv po referenci

- Pri radu sa nizovima, ne možemo proslijediti funkciji svaki član niza pojedinačno.
- Stoga je programerska praksa razvila **poziv po referenci** (eng. call by reference), gdje se proceduri prosljeđuje referenca na postojeći memorijski objekat (npr. niz), a ne njegova kopija.
- Promjene koje se izvrše nad memorijskim objektom (koristeći referencu) u pozvanoj proceduri reflektuju se na originalni objekat u pozivajućoj proceduri (jer se radi o istom objektu).
- Programski jezik C ne podržava poziv po referenci, već samo **poziv po vrijednosti** (eng. call by value), gdje se u parametre funkcije upisuje kopija vrijednosti argumenata.
- U C-u se poziv po referenci simulira koristeći pokazivače.

Prosljeđivanje preko pokazivača

```
int zbir(int *x, int *y) {
        (*x)++;
        (*y)++;
        return (*x) + (*y);
}
```

promjenljive **x** i **y** su pokazivači

x i **y** i dalje ostaju iste adrese, ali se sad mijenja (uvećava za jedan) ono što se nalazi na tim adresama. To ima efekat i na glavni program. ***x** je ono što se nalazi na adresi x.

Poziv funkcije zbir se obavlja:

```
int x=1, y=2;
c = zbir(&x, &y);
```

argument je adresa promjenljive

Primjer: Suma niza

- Prethodni primjer sa prosljeđivanjem promjenljivih preko pokazivača (adresa) nije tipičan.
- Dajmo sad dva mnogo karakterističnija primjera. Prvi je sumiranje članova niza:

```
int sumaniza(int *a, int n) {
        int i, s = 0;
        for(i = 0; i < n; i++)
                s += a[i];
        return s;
}
```

Funkciji prosljeđujemo adresu niza (tj. adresu njegovog prvog člana), kao i broj članova niza (kod nizova gotovo uvijek postoji jedan ovakav par argumenata funkcije).

alternativna deklaracija kod nizova
int sumaniza(int a[], int n)

članovima niza se pristupa na uobičajen način

Primjer: Dupliraj svaki drugi

- U ovom primjeru ćemo duplirati svaki drugi član niza.

```
void duplo(int *a, int n) {
        int i;
        for(i = 0; i < n; i += 2)
                a[i] *= 2;
}
```

Funkcija je deklarisana kao void jer ne vraća rezultat, ali sve promjene koje se izvrše na nizu se odražavaju i na argument (niz) u pozivajućoj funkciji.

- Druga tipična primjena pokazivača kod funkcija je u slučaju da funkcija treba da vrati više od jednog rezultata. Samo jedan rezultat se može vratiti direktno preko return-a, a ostali se rezultati moraju vraćati preko pokazivača.

Maksimum i pozicija maksimuma

- Napisati funkciju koja vraća maksimum i poziciju maksimuma niza. Jedan od rezultata vraćamo naredbom return, a drugi preko pokazivača. Moguća realizacija:

```
int max(int a[], int n, int *poz) {
    int i, m = a[0];
    *poz = 0;
    for(i = 1; i < n; i++)
        if(a[i] > m) {
            m = a[i];
            *poz = i;
        }
    return m;
}
```

poz je pokazivač preko kojeg se vraća pozicija maksimuma. Sami protumačite funkciju.

Funkcija se može pozvati kao a = max(x, n, &p).
Šta su sada x, n i p?

String kao argument funkcije

- String se funkciji prosljeđuje kao argument tipa **char ***, ali sada nije potrebno prosljeđivati njegovu dužinu.
- Razlog je taj što funkcija implicitno zna dužinu stringa preko terminacionog karaktera.
- Na primjer, realizacija naše verzije strlen funkcije bi bila:

```
int duzinaStringa(char *s) {
        int i = 0;
        while(s[++i] != '\0') ;
        return i;
}
```

Protumačite kako ciklus koji sadrži samo praznu naredbu obavlja po nas korisnu aktivnost.

Matrica kao argument funkcije

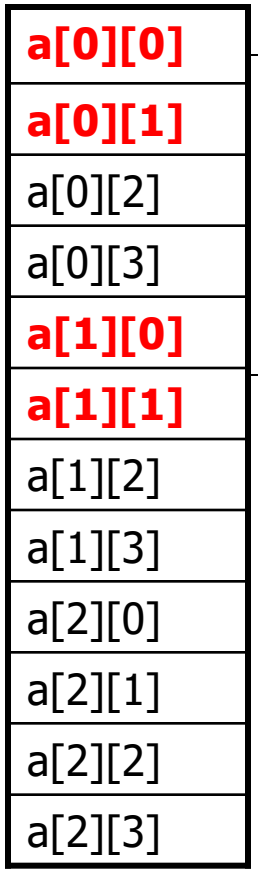
- Na prvi pogled ne očekuje se ništa što bi razlikovalo rad sa matricama od rada sa nizovima jer se očekuje da se elementima matrice u funkciji pristupa preko pokazivača na isti način kao što je slučaj sa nizovima.
- Postoji, ipak, jedna sitna izmjena. Neka je u programu deklarisana matrica

  ```
  float m[3][4];
  ```

- Dimenzionisanje se po pravilu vrši na najveću veličinu očekivane matrice u programu. Neka korisnik koristi dio matrice N×M = 2×2.

Matrica kao argument funkcije

- Elementi matrice su alocirani na sljedeći način:



Kod matrice može da se dogodi da su djelovi koji se koriste razdvojeni u memoriji. Stoga funkciji treba sugerisati kakav je razmak (zato se često kaže da funkcija mora da zna kako da odbrojava po memoriji). To se može postići deklaracijom

```
tip fun(float a[][4], int N, int M, ...)
```

Pravi broj vrsta i kolona

Argument je niz pokazivača na nizove od 4 realna broja, odnosno, kompajler zna da je razmak između vrsta 4 podatka tipa float.

Eventualno dodatni argumenti funkcije. Napominjemo da postoji operator ... koji sugeriše da funkcija ima proizvoljan broj argumenata, ali ga mi nećemo koristiti, već ovo znači da funkcija ima još argumenata.

Matrica – argument funkcije

- Alternativna deklaracija je:

  tip fun(float (*a)[4], int N, int M, ...)

- Poziv naše funkcije **fun** u programu bi bio:

  fun(a, 2, 2,...)

- Rad sa matricama u funkcijama očigledno nosi određene probleme, ali sa dosta opreza nije problem raditi ni sa ovim tipom argumenata funkcije.
- Vidjeli smo da funkcija može da vrati rezultat koji je standardnog tipa podataka uključujući modifikacije tipa unsigned, long, itd.
- Funkcija može da vrati i pokazivač.

Složene deklaracije funkcija

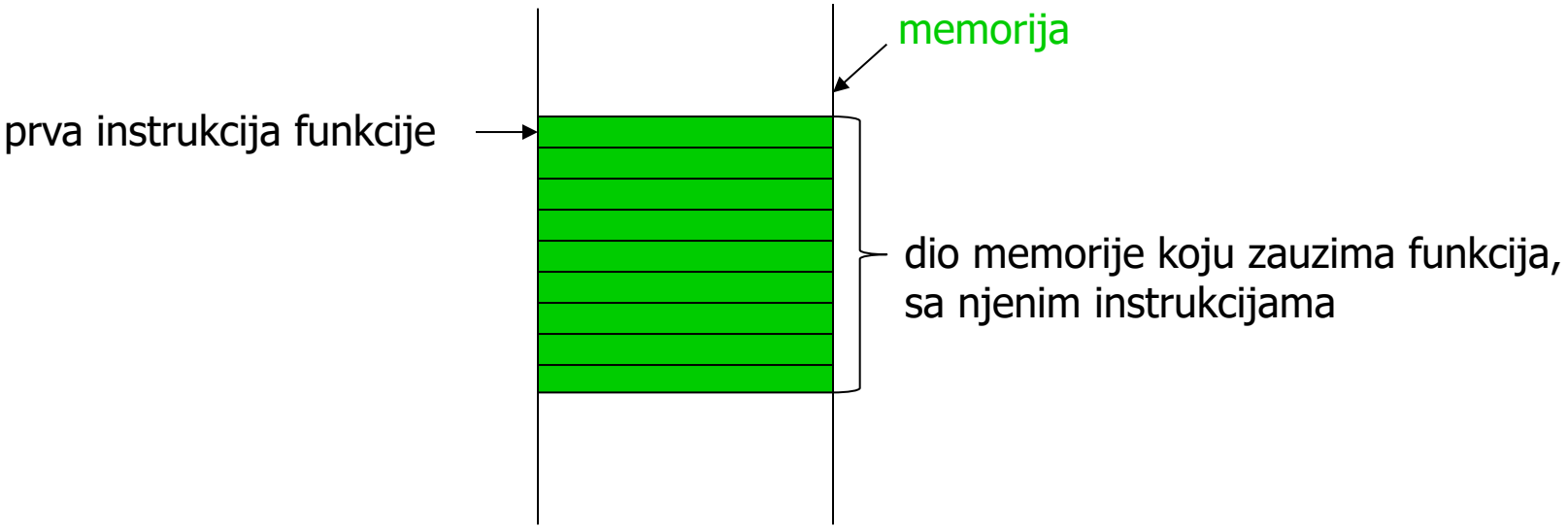
- int *f() je funkcija koja vraća pokazivač na cijeli broj.
- Ovo zahtjeva veliki oprez u radu, jer promjenljiva na koju dati pokazivač pokazuje ne smije da bude privremena i da nestane izlaskom iz funkcije.
- Dakle, rezultat rada funkcije koja vraća pokazivač mora da bude adresa promjenljive koju koristi funkcija (ili glavni program) koja je pozvala predmetnu funkciju.
- Česta je greška da se u funkciji deklariše niz i da funkcija vraća pokazivač na taj niz. Izlaskom iz funkcije, dealociraju se sve promjenljive alocirane u funkciji.
- Mi ćemo, kad god je to moguće (a neće biti uvijek moguće), izbjegavati ovakve funkcije.

Složene deklaracije funkcija

- Kombinacija niz-pokazivač-funkcija dovodi do veoma komplikovanih deklaracija.
- Pored pravila koja se koriste kod nizova uvodi se dodatno pravilo:
  - Par [] ima isti prioritet kao par ().
- Posmatrajmo primjer int (*f) () (zanemarimo parametre):
  - U prvom koraku eliminišemo desnu zagradu; dakle zaključujemo da je *f funkcija koja vraća cijeli broj.
  - U drugom koraku eliminišemo * i zaključujemo da je f pokazivač na cjelobrojnu funkciju, odnosno adresa cjelobrojne funkcije.
- Pogledajte ostale primjere iz knjige.

Adresa funkcije

- Svi elementi programa zauzimaju memoriju.
- Tako i kôd funkcije zauzima memoriju.



Adresa funkcije je adresa prvog bajta funkcije, odnosno pokazivač na funkciju sadrži adresu prvog bajta prve instrukcije funkcije, što je analogno pokazivačima na sve tipove promjenljivih.

Adresa funkcije

- Adresa funkcije koja se zove fun() je &fun, ali se može koristiti i samo fun.
- Ovo je slično adresi niza, kod kojeg ime niza ima vrijednost adrese (prvog elementa) niza.
- Najvažnija primjena adresa funkcija je u proslijeđivanju funkcija kao argumenta drugih funkcija.
- Naime, ponekad je zgodno omogućiti jednoj funkciji da poziva više drugih funkcija u zavisnosti od situacije.

Funkcija sa argumentom funkcijom

- Zamislimo sljedeću situaciju. Napisali smo funkciju koja na osnovnu nekog sofisticiranog metoda računa integral funkcije **sin(x)**.
- U slučaju da nam treba funkcija koja računa integral kosinusa morali bi da napišemo (kopiramo) prethodnu funkciju za računanje i da editujemo funkciju koja je argument (**cos(x)**).
- Ovaj postupak bi morali obaviti za svaku funkciju za koju želimo da nađemo integral.
- Mala greška u našem početnom dizajnu dovodi do potrebe da prepravljamo sve funkcije, što je neprihvatljivo.

Funkcija sa argumentom funkcijom

- Drugi problem koji postoji kod pretpostavljenog rješenja je situacija kada želimo da našu funkciju komercijalizujemo.
- Korisniku bismo morali omogućiti da primjeni naš algoritam na bilo koju funkciju, a to bi značilo da korisnik mora biti upoznat sa našim kodom.
- Ovim bismo zapravo potpuno razotkrili svoju programersku vještinu i u potpunosti ugrozili komercijalizaciju našeg programa.
- Umjesto toga, naša funkcija treba da ima kao argument funkciju i da korisnik, pa i mi, sa istom funkcijom realizuje svaki projekat, a jedino znanje koje je tada potrebno o našoj funkciji je njeno zaglavlje (lista i tipovi argumenata).

Primjer

- Pretpostavimo sljedeći jednostavan problem (u svakom slučaju jednostavniji nego što je postavka problema o kojem smo diskutovali).
- Želimo da sumiramo niz:

$$\sum_{i=0}^{n-1} f(a[i])$$

gdje je a[i], i=0,...,n-1 dati niz, recimo, cijelih brojeva, a f() proizvoljna funkcija.

Primjer - Realizacija

```
#include <stdio.h>
int sumaFunkcija(int (*)(), int*, int);
int linija(int);
int kvadrat(int);
```

pretprocesor i prototipovi funkcija

```
int linija(int n) { return n; }
int kvadrat(int n) { return n*n; }
int sumaFunkcija(int (*r)(), int *a, int n) {
    int i, s = 0;
    for(i = 0; i < n; i++)
      s += r(a[i]);
    return s;
}
```

Realizacija funkcija - obratite pažnju na funkciju sumaFunkcija, a posebno na način sumiranja.

```
int main(){
   int a[5] = {1,2,2,3,1};
   int b = sumaFunkcija(&linija,a,5);
   int c = sumaFunkcija(kvadrat,a,5);
   printf("%d %d", b, c);
}
```

Glavni program sa pozivima funkcije. Obratite pažnju na podvučeni dio i pokušajte da odgovorite na pitanje zašto rade ispravno obje varijante i zašto se nigdje ne naglašava da je podvučeni argument funkcija?

Funkcija main

- Glavni program ima oblik bilo koje druge funkcije – ima zaglavlje, tijelo funkcije koje obično započinje sa sekcijom za deklaraciju, nastavlja se naredbama i završava sa return.
- Postavlja se pitanje - kome funkcija main vraća rezultat?
- Funkcija main, kao i svaka druga funkcija, vraća rezultat onome ko je tu funkciju pozvao, a u ovom slučaju je to **operativni sistem**.
- UNIX/LINUX operativni sistemi lijepo rade sa ovakvim rezultatima, koji obično sugerišu da je funkcija sa uspjehom obavila posao.

Parametri funkcije main

- Do sada smo koristili funkciju main bez parametara. Međutim, ova funkcija može imati parametre, a to su uvijek cijeli broj i niz pokazivača na karaktere.
- Ako funkcija main ima parametre, ona se deklariše kao:

  int main (int argc, char *argv[])
- Što su ovi parametri i kako im zadati vrijednost?
- Objasnimo na primjeru. Kreirali smo program i kompajlirali ga, što je rezultovalo fajlom test.exe. U komandnoj liniji oper. sistema, program se izvršava pozivom imena test.
- Zamislimo sada da je program pozvan sa

  test abc -4.5 57

Parametri funkcije main

- Vrijednost argc će biti 4, dok će pojedini stringovi biti:

  argv[0] = "test"
  argv[1] = "abc"
  argv[2] = "-4.5"
  argv[3] = "57"

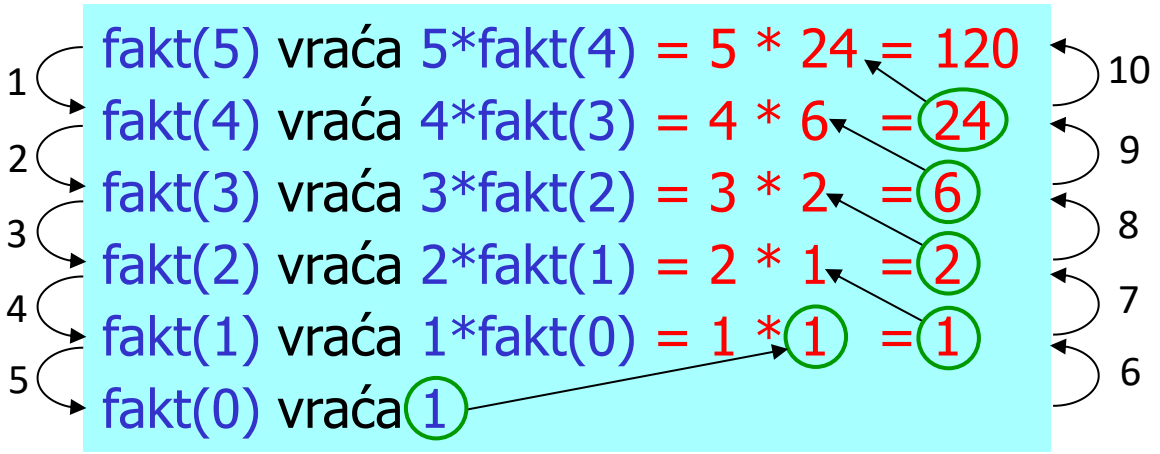
  Ime exe fajla je uvijek prvi string

- Za vježbu napisati program koji sa komandne linije učitava dva stringa. Program vrši njihovo nadovezivanje i štampanje.
- Za vježbu napisati program koji sa komandne linije učitava niz cijelih brojeva. Potrebno je izvršiti štampanje ovog niza.

Rekurzija

- Savremeni programski jezici dozvoljavaju da funkcija pozove samu sebe, što se naziva **rekurzijom**.
- Rekurzivno se funkcija faktorijel može realizovati korišćenjem pravila: n! = n*(n-1)! i 0! = 1.

```
int fakt(int n) {
  if(n==0)
    return 1;
  else
    return n * fakt(n-1);
}
```



Rekurzija

- Rekurzivna rješenja su veoma elegantna. Neki programerski problemi se veoma teško rješavaju bez korišćenja rekurzije.
- Rekurzija može biti memorijski zahtjevna (što se dešava prilikom poziva fakt(120)), ali i vremenski, jer operacije sa stekom i alokacionim zapisom traju neko vrijeme.
- Stoga, rekurzivno treba rješavati one probleme kod kojih je prirodno korisiti rekurzivni pristup (binarna pretraga, quick sort itd.).
- Kao primjer loše rekurzije, realizovati rekurzivnim putem računanje Fibonacci-evih brojeva, za koje važi FIB(1) = 0, FIB(2) = 1 i FIB(I) = FIB(I-1) + FIB(I-2) za I>2.
- Program realizujte rekurzivno i odredite broj pozivanja funkcije FIB za I=6 ili I=15. Zatim dajte alternativno iterativno rješenje.