**0. ELEMENTI TEORIJE ALGORITAMA ..... 5**

**0. 1. INTUITIVNA DEFINICIJA ALGORITMA ..... 5**

**0. 2. OSOBINE ALGORITAMA ..... 6**

**0.3. ZAPIS (OPIS) ALGORITMA ..... 8**
0.3.1. Zapis algoritama skupom pravila ..... 8
0.3.2. Zapis algoritama dijagramom toka ..... 9

**0.4. BEKUSOVA NORMALNA FORMA (BNF) ..... 10**
0. 4. 1. SINTAKSNI DIJAGRAMI ..... 13

**1. OSNOVNI ELEMENTI PROGRAMSKIH JEZIKA ..... 13**

**1.1. Pseudojezik kao opšti model viših programskih jezika ..... 13**

**1.2. Azbuka jezika ..... 15**

**1.3. Identifikatori i rezervisane reči ..... 15**

**1.4. Konstante ..... 16**

**1.5. Promenljive ..... 17**

**1.6. Komentari ..... 17**

**1.7. Struktura programa ..... 18**
1.7.1. Struktura programa u C ..... 18

**2. TIPOVI PODATAKA ..... 18**
Statička i dinamička tipizacija ..... 20

**2.1. Koncept jakih tipova podataka ..... 21**

**2.2. Koncept slabih tipova ..... 21**

**2.3. Ekvivalentnost tipova ..... 22**

**2.4. Elementarni tipovi podataka ..... 22**
2.4.1. Celobrojni tipovi (Integer ili int) ..... 22
2.4.2. Realni tip (float ili real) ..... 23
2.4.3. Logički tip podataka ..... 23
2.4.4. Znakovni tipovi ..... 23

**2.5. Tipovi podataka u jeziku C ..... 24**
Celobrojni tipovi u C ..... 25
Realni tipovi podataka u C ..... 26
Tip char ..... 27
Konverzija tipova podataka i kast ..... 28
Vrste konverzije tipova ..... 29
Konstante ..... 29
Sizeof operator ..... 30
Osnovne aritmetičke operacije ..... 30
Operacije poređenja i logičke operacije ..... 33

**2.6. Diskretni tipovi podataka u programskim jezicima ... 34**
2.6.1. Nabrojivi tipovi u programskim jezicima ... 34

**2.8. Upotreba typedef iskaza u C ... 34**

**3. ULAZ I IZLAZ PODATAKA ... 35**

**3.1. Ulaz i izlaz podataka u jeziku C ... 35**
3.1.1. Funkcije printf() i scanf() ... 35
3.1.2. Direktive pretprocesora u C ... 38

**4. OSNOVNE UPRAVLJAČKE STRUKTURE ... 40**

**4.1. načini predstavljanja algoritama ... 41**

**4.2. Sekvenca naredbi i blok ... 47**
4.2.1. Globalne i lokalne promenljive ... 47

**4.3. Struktura selekcije ... 49**
4.3.1. If-then struktura ... 49
4.3.2. If-then-else struktura ... 50
4.3.3. Operator uslovnog prelaza u C ... 56

**4.4. Struktura višestruke selekcije ... 56**
4.4.1. Višestruka selekcija u C ... 57

**4.5. Programske petlje ... 60**
4.5.1. Programske petlje u C ... 61
While naredba u C ... 61
Primena while ciklusa u obradi teksta u C ... 65
Do-while naredba u C ... 65
For naredba u C ... 66

**4.6. Formalizacija repetitivnih iskaza ... 69**

**4.7. Nasilni prekidi ciklusa ... 71**

**4.8. Naredbe za bezuslovno grananje ... 72**
4.8.1. Oznake (labele) ... 72

**5. POTPROGRAMI ... 76**

**5.1. Funkcije ... 77**
5.1.1. Poziv i definicija funkcija u C ... 77
Return naredba ... 78
Prototip funkcije ... 79
5.1.2. Makroi u jeziku C ... 84

**5.2. Procedure ... 84**

**5.3. Prenos argumenata ... 85**
5.3.1. Prenos po vrednosti (Call by Value) ... 85
Prenos parametara po vrednosti u C ... 86
5.3.2. Prenos po rezultatu (Call by Result) ... 87
5.3.3. Prenos po vrednosti i rezultatu (Call by Value-Result) ... 87
5.3.4. Prenos po referenci (Call by Reference) ... 87
Poziv po adresi pomoću pokazivača u C ... 89
Prenos po referenci koristeći reference u C++ ... 93

Vraćanje višestrukih vrednosti ..... 95
Vraćanje višestrukih vrednosti po referenci ..... 97
Predavanje po referenci, zbog efikasnosti ..... 98

**5.4. Globalne promenljive kao parametri potprograma ..... 105**
Primeri ..... 107

**5.5. Rekurzivni potprogrami ..... 110**
5.5.1. Primeri rekurzivnih funkcija u C ..... 112

**5.6. Implementacija potprograma ..... 115**

**5.7. Scope rules (domen važenja) ..... 118**

**5.8. Memorijske klase u C ..... 119**
5.8.1. Životni vek objekata ..... 121
5.8.2. Vrste objekata po životnom veku ..... 121
Statički i automatski objekti ..... 121

**6. STRUKTURNI TIPOVI PODATAKA ..... 122**

**6.1. Polja u programskim jezicima ..... 122**

**6.2. Jednodimenzionalni nizovi u C ..... 123**

**6.3. Veza između nizova i pointera u C ..... 133**
6.3.1. Pointerska aritmetika ..... 135

**6.4. Nizovi i dinamička alokacija memorije u C ..... 136**

**6.5. Višedimenzionalna polja ..... 142**
6.5.1. Višedimenzionalni nizovi u C ..... 143
6.5.2. Pokazivači i višedimenzionalni nizovi u C ..... 143
6.5.3. Matrice i dinamička alokacija memorije ..... 170

**6.6. STRINGOVI ..... 173**
6.6.1. Stringovi u C ..... 173
Inicijalizacija i obrada stringova ..... 173
Testiranje i konverzija znakova ..... 175
Učitavanje i ispis stringova ..... 176
Standardne funkcije za rad sa stringovima u C ..... 181

**6.7. Strukture i nabrojivi tipovi u c ..... 184**
6.7.1. Članovi strukture ..... 184
6.7.2 Strukturni tipovi i pokazivači u C ..... 189
6.7.3. Definicija strukturnih tipova pomoću typedef ..... 193
6.7.4. Unije ..... 165

**6.8. Nabrojivi tip podataka u c ..... 168**

**6.9. Dinamičke matrice i strukture ..... 170**

**6.10. Datoteke ..... 172**
6.10.1. Pristup datotekama u C ..... 172
Otvaranje i zatvaranje datoteka ..... 173
Funkcije fgetc() i fputc() ..... 175
fputc ..... 175
Funkcije fprintf() i fscanf() ..... 176
Funkcija feof() ..... 178

fclose, _fcloseall ..... 180
feof ..... 180
6.10.2. Binarne datoteke ..... 181
Direktni pristup datotekama ..... 182

**Literatura ..... 183**

# 0. ELEMENTI TEORIJE ALGORITAMA

## 0. 1. INTUITIVNA DEFINICIJA ALGORITMA

Algoritam jeste jedan od osnovnih pojmova matematike i računarstva. Prve algoritme srećemo kod starih Grka. Svima su poznati Euklidov algoritam i Eratostenovo sito (250. g. p. n.e.) Algoritam predstavlja uputstvo za rešavanje nekog zadatka u cilju dobijanja rešenja (posle konačno mnogo vremena).

Pri rešavanju nekog složenog zadatka (problema) korsitimo rastavljanje (dekompoziciju) zadatka na prostije podzadatke (podprobleme). Postupak možemo nastaviti sve dok se ne dobije jedan konačan skup relativno jednostavnih podzadataka. Takve podzadatke nazvaćemo elementarnim koracima ili elementarnim operacijama. Svaki takav korak definiše koju operaciju i kojim redosledom se takva obrada izvršava u cilju dobijanja rešenja datog problema. Ukoliko je elementarni korak relativno složen tada se on može zadati u vidu uputstva (pravila) za rešavanje takvog podproblema. Zato elementarne korake nazivamo i (elementarnim) pravilima.

Opis toka odvijanja elementarnih operacija (koraka) u cilju rešavanja nekog zadatka (problema) naziva se procedura. Procedura se sastoji od konačno mnogo elementarnih koraka koji se mogu mehanički izvršavati u određenom strogo definisanom redosledu i to za konačno vreme.

Primena procedure može dovesti do sledeća tri slučaja:

1) Primena procedure predstavlja beskonačan proces. Rešenje zadatka se u tom slučaju ne može dobiti posle konačno mnogo koraka, odnosno, posle koančno mnogo vremena. Procedura ne daje rezultat.
2) Primena procedure se prekida posle konačno mnogo koraka i ne dobija se rezultat. Proces računanja se ne može nastaviti jer se ne zna korak na koji se prenosi izvršenje. Uzrok ove situacije mogu biti pogrešni ulazni podaci (ili pogrešno definisana procedura). U prethodna dva slučaja kažemo da imamo nerezultativnu proceduru.
3) Proces primene procedure se prekida posle konačno mnogo koraka sa dobijanjem rezultata. Rešenje zadatka postoji i može se dobiti primenom procedure (rezultativne).

Samo u trećem slučaju kažemo da procedura predstavlja algoritam koji se može primeniti za rešavanje datog zadatka. Na sličan način uvedene su i stroge matematičke definicije algoritama.

Na osnovu prethodnog možemo dati intuitivnu definiciju algoritma, gde umesto pojma elementarni korak koristimo naziv algoritamski korak (pravilo).

***Def*. 1: Algoritam je konačan (uređen) skup strogo definisanih algoritamskih koraka (pravila) čijom primenom na ulazne podatke (i međurezultate) dobijamo rešenje zadatka posle konačno mnogo vremena.**

Iz definicije uočavamo da svaki algoritam ima konačno mnogo algoritamskih koraka (koij je najčešće uređen). Svaki algoritamski korak mora biti strogo (nedvosmisleno) definisan (kao i ceo algoritam) tako da se može na isti način razumeti i izvršiti od strane bilo kog izvršioca.

Primena algoritma se završava posle konačno mnogo vremena a to znači da svaki algoritamski korak mora biti izvršiv i da se mora primeniti konačno mnogo puta.

Algoritam definiše preslikavanje (funkciju) koje svakom izboru vrednosti ulaznih veličina algoritma (zadatka) pridružuje odgovarajući rezultat. Kažemo da svaki algoritam ima svoju oblast

primenljivosti (ulaz).

## 0. 2. OSOBINE ALGORITAMA

Intuitivna definicija algoritma je nestroga definicija. Zato navodimo neke osobine koje karakterišu svaki algoritam. Na taj način dodatno preciziramo ovaj pojam tako da je ekvivalentan sa strogim matematičkim definicijama algoritma.

1. **Diskretnost**. Svaki algoritam predstavlja konačan uređen skup algoritamskih koraka. To je rezultat dekompozicije (diskretizacije) problema koji se rešava od strane čoveka. Pri tome, algoritam nije skup algoritamskih koraka, već uređen skup ili niz, jer je bitan redosled algoritamskih koraka u zapisu algoritma. Takođe kažmo da je proces izvršenja algoritma diskretan u vremenu. To znači da se u svakom vremenskom trenutku (intervalu) izvršava samo jedan algoritamski korak. Proces izvršavanja algoritma jeste konačan niz algoritamskih koraka koji se po strogo definisanom redosledu izvršavaju. To je slučaj kada imamo jednog izvršioca algoritma (jedan procesor). Postoje i paralelni algoritmi, koji omogućuju da se dva ili više algoritamskih koraka izvršavaju istovremeno.
2. **Determinisanost (određenost).** Svaki algoritamski korak treba da je definisan jasno, strogo (tačno) i nedvosmisleno. Tumačenje i izvršavanje pravila algoritma ne sme zavisiti od volje čoveka ili mašine. Posle izvršenja nekog algoritamskog koraka strogo je definisan prelaz na sledeći algoritamski korak. To znači da je izvršenje algoritma deterministički proces i da se može automatski izvršavati. Opis algoritma na prirodnom jeziku može dovesti do dvosmislenosti.
3. **Izvršivost.** Uspešnu definiciju izvršivosti dao je D. Knut. On kaže da je algoritamski korak izvršiv ako je čovek u stanju da ga izvrši za konačno vreme (pomoću olovke i papira). Kod algoritamski rešivih zadataka nema neizvršivih koraka. Međutim, moguće je lako formulisati pravilo koje nije izvršivo. Na primer: Ako razvoj broja $\pi$ sadrži 7 uzastopnih devetki tada sabrati sve preostale cifre.
4. **Konačnost**. Osobina konačnosti algoritma jeste zahtev da se izvršenje svakog algoritma završi posle konačno mnogo vremena. Drugim rečima,izvršenje svakog algoritma je postignuto posle konačno mnogo primena algoritamskih koraka. Zbog osobina 1. i 3. to znači da svaki algoritamski korak ma kog algoritma mora da se izvrši konačno mnogo puta. U suprotnom, nemamo algoritam. Takav je sledeći primer procedure koja se nikad ne završava :

   Korak 1 : Neka je $i = 0$ ;
   Korak 2 : Neka i dobije vrednost $i + 1$ ;
   Korak 3 : Pređi na korak 2 .

Iako imamo konačno mnogo algoritamskih koraka, pri čemu je svaki od njih strogo definisan i izvršiv, ovo nije algoritam. Kod složenijih algoritama (zadataka)

javlja se potreba formalnog dokaza konačnosti algoritma.

5. **Ulaz i izlaz algoritma.** Svaki algoritam ima dva posebno izdvojena (konačna) skupa podataka (veličina). Prva jeste skup ulaznih a druga skup izlaznih veličina. Broj veličina u ovim skupovima može biti i nula. Skup ulaznih veličina algoritma predstavlja polazne veličine (podatke) zadatka koji se rešava. Skup izlaznih veličina jeste traženo rešenje (rezultat) postavljenog zadatka. Ukratko, ove skupove nazivamo ulaz i izlaz algoritma. (Za podatke koji na pripadaju skupu ulaznih podataka algoritma kažemo da algoritam nije primenljiv).
6. **Masovnost (Univerzalnost).** Univerzalnost je osobina algoritma da se može primeniti na što širu klasu problema. To upravo znači da ulazne veličine algoritma mogu uzimati početne vrednosti iz što obilnijih (masovnijih) skupova podataka. Algoritam jeste opšte uputstvo koje se može primeniti na ma koji izbor vrednosti ulaznih veličina. Zbog ove osobine,možemo reći da je algoritam bolji ukoliko je univerzalniji (termin masovnost manje odgovara ovoj osobini).

7. **Elementarnost algoritamskih koraka.** Algoritam treba da sadrži algoritamske korake koji predstavljaju elementarne operacije koje korisnik algoritma može da razume ili izvršilac algoritma da izvrši. Za potrebe čoveka, algoritamski koraci mogu biti kompleksnije fundamentalne ili logičke celine u složenom algoritmu. Međutim, za potrebe pisanja programa, algoritam sadrži elementarne algoritamske korake koji odgovaraju naredbama ili pozivima potprograma, algoritam sadži elementarne algoritamske korake koji odgovaraju naredbama ili pozivima potprograma programskog jezika.
8. **Rezultativnost (usmerenost).** Algoritam je tako definisan da polazeći od proizvoljnih vrednosti ulaznih veličina primena algoritamskih koraka vodi (usmerava) strogo ka dobijanju traženog rezultata.

Analizom izvršenja algoritma mogu se uočiti tri načina izvršavanja algoritamskih koraka: sukcesivno (sekvencijalno), sa granicom i cikličko.

a) Sukcesivno izvršavanje algoritamskih koraka jeste takvo da se koraci izvršavaju jedanput i u redosledu kako su napisani. Takvi algoritamski koraci čine prostu linijsku algoritamsku strukturu ili sekvencu (nisku) algoritamskih koraka.
b) Izvršavanje algoritamskih koraka tako da se neki izvrše jedanput a neki nijednom predstavlja primer izvršavanja sa grananjem (ili prelazom). Takvu algoritamsku strukturu nazivamo razgranatom linijskom algoritamskom strukturom.
c) Cikličko izvršavanje se javlja kada se grupa algoritamskih koraka izvršava više puta. Takvu grupu naredbi nazivamo cikličkom algoritamskom strukturom ili ciklusom.

Korišćenjem (komponovanjem) ovakve tri algoritamske strukture možemo rešiti ma koji zadatak.

**Paralelnost (jednovremenost)**. Algoritam definišemo kao konačan skup algoritamskih koraka. Taj skup je uređen jer se njime definiše prirodan redosled izvršavanja koraka sa mogućnošću grananja ili ponovnog izvršavanja. Za izvršenje algoritma kažemo da je niz primena algoritamskih koraka. Postavlja se pitanje šta je pravilnije reći : skup ili niz. Prednost dajemo prvom terminu jer **postoje algoritmi u kojima se može promeniti redosled nekih algoritamskih koraka a da se rešenje ne menja. Takve algoritamske korake možemo izvršavati istovremeno (paralelno), pomoću više izvršioca (procesora). Algoritmi koji definišu paralelno izvršavanje algoritamskih koraka jesu paralelni algoritmi, a paralelni procesori izvršavaju takve algoritme.**

**Efikasnost (efektivnost)**. Dva različita algoritma koji rešavaju jedan isti zadatak možemo upoređivati u odnosu na neki izbrani kriterijum. Rezultat upoređivanja može biti da je jedan algoritam bolji ili efikasniji od drugog. Izbor kriterijuma može biti različit. **Najčešće se pravi kompromis između različitih kriterijuma (vreme – memorija, jednostavnost – brzina ).** Često puta neki efikasni algoritmi zavise od izbora ulaznih podataka algoritma. Tako imamo efikasne algoritme sortiranja podataka za slučajno izabrane (neuređene) podatke ili pak za delimično uređene.

**Elegantnost**. Elegantnost može biti subjektivan kriterijum za izbor algoritma, ali se često koristi. **Elegantno rešenje zadatka jeste ono koje je prosto (jednostavno) i originalno**. Jednostavnost rešenja je uvek poželjno ali je pitanje da li su uvek moguća originalna rešenja. Pod originalnošću rešenja podrazumeva se njegova neočiglednost.

Originalno rešenje za izmenu mesta (vrednosti) dveju promenljivih $a$ i $b$ može biti sledeći algoritam :

Korak 1 : Neka je $a = a+b$ ;
Korak 2 : Neka je $b = a-b$ ;
Korak 3 : Neka je $a = a–b$ .

Međutim,ovo nije elegantno rešenje jer nije jednostavno, a takođe nije efikasno niti univerzalno (ne važi za sve moguće vrednosti a i b). Univerzalno rešenje koristi pomoćnu promenljivu :

1) $p = a$ ;
2) $a = b$ ;
3) $b = p$ .

**Rekurzija.** Nasuprot iteraciji, koja se najčešće koristi za rešavanje zadataka cikličke prirode,

rekurzija je razrađena od strane specijalista funadamentalne matematike kao način definisanja funkcija. Ona se koristi i u (računarstvu i) programiranju za rekurzivne definicije tipova i struktura podataka, kao i za rekurzivne potprograme (funkcije i procedure). Rekurzivne definicije se karakterišu time što objekat koji se definiše figuriše u samoj definiciji. Rekurzivna definicija funkcije *f*, koja zavisi od prirodnog broja *n*, sadrži dva dela. Najpre se definiše veza između funkcije $f(n)$ i funkcije $f(n-1)$. U drugom delu se definiše vrednost funkcije *f* za posebnu vrednost argumenta *n*. Na sličan način se iskazuju različite rekurzivne definicije. Navedimo sledeće rekurzivne definicije funkcija :

1) n-faktorijel : $$n! = \begin{cases} n \cdot (n-1)! & , n > 1 \\ 1 & , n = 0, 1 \end{cases}$$

2) stepen : $$a^n = \begin{cases} a \cdot a^{n-1}, & n > 0 \\ 1, & n = 0 \end{cases}$$

3) Fibonačijev niz : $$f(n) = \begin{cases} f(n-1)+f(n-2) & , n > 1 \\ 1 & , n = 0 \end{cases}$$

4) najveći zajednički delilac :

$$\text{NZD}(m,n) = \begin{cases} n, & m = \text{n} \\ \text{NZD}(n-m, m), & n > m \\ \text{NZD}(m-n, n), & m > n \end{cases}$$

# 0.3. ZAPIS (OPIS) ALGORITMA

Opis algoritma ima dva osnovna cilja :

1) da je (algoritam) razumljiv raznim korisnicima (ljudima) tako da se može koristiti ili prenositi;
2) da se na osnovu njega može pisati program na nekom (algoritamskom) programskom jeziku u cilju izvršavanja na računaru.

U tom cilju razvijeni su različiti sistemi (dogovori) za zapis (opis, zadavanje, predstavljanje) algoritama. Ovakvi sistemi zapisivanja algoritama treba da budu jednostavni ali (dovoljno) precizni tako da ih čovek može lako da razume. Takvi opisi algoritama predstavljaju 'univerzalne' jezike, koji su nezavisni od računara. Postoji nekoliko najčešće korišćenih načina za zapis algoritama,a to su :

a) skup pravila (pomoću prirodnog jezika);
b) dijagram toka (pomoću grafičkih tabela);
c) pseudokod.

Za tačan zapis algoritama tako da se može izvršiti na računaru koristimo algoritamske programske jezike. Ekvivalentan zapis algoritma na programskom jeziku jeste program. Program predstavlja zapis algoritma pomoću naredbi koje su ekvivalentne odgovarajućim algoritamskim koracima.

## 0.3.1. Zapis algoritama skupom pravila

Najprirodniji način zapisa algoritama jeste pomoću prirodnog jezika kojim komuniciraju ljudi. Tako se algoritam može saopštiti govorom ili tekstualno. Primeri za to su razni kulinarski recepti, uputstva za rukovanje mašinama, koja jesu neka vrsta algoritama. Takvi opisi mogu biti neprecizni, pa čak i dvosmisleni zbog složenosti prirodnog jezika.

Međutim, zapis algoritama pomoću skupa pravila ima zadatak da pregledno, jasno i tačno opiše algoritamske korake pomoću reči, simbola i rečenica prirodnog jezika. Oblik pravila algoritma je na određen način precizan tako da su ona jasna i razumljiva svakome. U zapisu algoritama skupom

pravila može se uvesti ime algoritma. Pored toga algoritamska pravila se najčešće označavaju rednim brojevima (0,1,2,. . . ), sa dodatnom simboličkom oznakom u vidu slova ili reči. Izvršenje algoritma počinje od pravila sa rednim brojem 0 ili 1. Zatim se pravila algoritma izvršavaju prirodnim redosledom (redno) dok se ne dođe do pravila za grananje,prelaz ili pravila za cikličko izvršavanje algoritamskih koraka.

Navodimo neka najčešće upotrebljavana pravila za zapis algoritma :

1. Pravilo dodeljivanja : <promenljiva> := <izraz>

Izračunata vrednost izraza dodeljuje se promenljivoj sa leve strane simbola dodeljivanja := . Alternativni simboli mogu biti = , ← , ⇐.

2. Pravilo uslovnog grananja : if <uslov> then <pravilo>

Ako je <uslov> ispunjen tada se izvršava <pravilo>,a u suprotnom prelazi se na sledeće pravilo.

3. Pravilo grananja : if <uslov> then <pravilo1>
else <pravilo2>

Ako je <uslov> ispunjen izvršava se <pravilo1>,a u suprotnom <pravilo2>.

4. Pravilo bezuslovnog grananja (skoka) : go to <oznaka pravila>

Ovo pravilo ukazuje da se prekida prirodni redosled izvršenja pravila algoritma i zahteva prelaz na pravilo označeno sa <oznaka pravila>.

5. Pravilo za prekid izvršenja algoritma : Kraj (Stop)

Posle izvršenja ovog pravila prekida se izvršenje algoritma.

6. Pravilo za ulaz podataka : Ulaz (<lista ulaznih veličina>)

7. Pravilo za izlaz podataka : Izlaz (<lista izlaznih veličina>)

8. Pravilo ciklusa : Ponavljaj
<pravilo 1>
<pravilo 2>
. . .
<pravilo n>
Sve dok se <uslov> ne ispuni

Ovo pravilo definiše ponovljeno izvršavanje niza pravila <pravilo 1>,. . . ,<pravilo n> sve dok <uslov> nije ispunjen. Kada se <uslov> ispuni prelazi se na sledeće pravilo.

Navedimo primer Euklidovog algoritma za nalaženje najvećeg zajedničkog delioca.

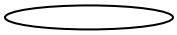
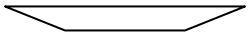
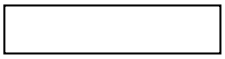
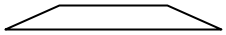
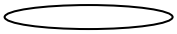
E0. Euklid
E1. Ulaz (*m*,*n*)
E2. *r*:= *m* MOD *n* {ostatak deljenja}
E3. Ako *r*=0 tada pređi na E7
E4. *m*:=*n*
E5. *n*:=*r*
E6. Pređi na E2
E7. nzd:=*n*
E8. Izlaz (nzd)
E9. Kraj

Slična algoritamska pravila imamo kod opisa algoritama pomoću pseudo koda. Pseudo kod koristi pravila bliska naredbama programskog jezika. Pravila su jednostavna, čitljiva i nedvosmislena.

## 0.3.2. Zapis algoritama dijagramom toka

Jedan od najjednostavnijih i često korišćenih načina zapisa algoritama jeste dijagram toka (ili blok šema) algoritma (tokovnik). Dijagram toka predstavlja jednu varijantu grafičkog opisa algoritma.

Svaki algoritamski korak predstavljen je grafičkim simbolom (blokom). Svi blokovi su povezani linijama sa mogućim strelicama. Na taj način se zadaje struktura algoritma i redosled izvršavanja algoritamskih koraka. Oblik grafičkog simbola ukazuje na vrstu algoritamskog koraka,odnosno njegovu funkciju u algoritmu. U tabeli su dati najčešće korišćeni grafički simboli i njihova funkcija.

| Grafički simbol algoritamskog koraka | Funkcija algoritamskog koraka |
|---|---|
|  | (Definiše) početak algoritma |
|  | Ulazne veličine algoritma (blok ulaza) |
|  | Obrada podataka (blok obrade) |
|  | Uslovni algoritamski korak (blok odluke) |
|  | Izlazne veličine algoritma (blok izlaza) |
|  | Kraj algoritma |

Proizvoljna algoritamska šema se u opštem slučaju može razložiti na elementarne šeme (ili elementarne algoritamske strukture). Postoje tri vrste elementarnih šema:

1) linijska (sekvenca)
2) razgranata (grananje)
3) ciklička (ciklus)

Pokazuje se da je dovoljno imati na raspolaganju jednu linijsku,jednu razgranatu i jednu cikličku elementarnu šemu da bi se realizovala proizvoljna algoritamska struktura. U praksi se često koriste različiti oblici razgranatih i cikličkih elementarnih šema.

Može se primetiti da svaka od ovih elementarnih šema ima jedinstven ulaz u šemu i jedinstven izlaz iz nje. To omogućuje lako komponovanje složenih algoritamskih šema pomoću elementarnih. Takve šeme zovemo dobro strukturiranim. Ukoliko se na izlaz jedne elementarne šeme poveže ulaz druge elementarne šeme i tako postupak nastavi dobijamo linijsku kompoziciju elementarnih šema. Međutim,ako na izlaz nekog algoritamskog koraka u okviru cikličke šeme nadovežemo ulaz elementarne cikličke šeme dobijamo koncentričnu kompoziciju cikličkih šema.

Višestruko grananje možemo realizovati kada (na izlaz nekog algoritamskog koraka) u okviru jedne od grana umetnemo novu razgranatu šemu.

# 0.4. BEKUSOVA NORMALNA FORMA (BNF)

Ovaj metajezik prvi je predložio Homski [56] za opis prirodnih jezika. Sistem zapisa koji opisujemo i koristimo pripada Bekusu, po kome je ovaj metajezik dobio naziv. Često se koristi termin Bekusova notacija (ili Bekus-Naurova forma što je donekle nepravilno). Ovim metajezikom opisana je nepotpuno sintaksa jezika ALGOL 60. Prema mišljenju Homskog, ovaj metajezik je pogodan za opis jednostavnijih (kontekstno-nezavisnih) jezika. Zato su lingvisti pristupili definisanju i korišćenju moćnijih metajezika.

Bekusova notacija je jednostavan metajezik. Osnovni pojmovi (konstrukcije) u ovom jeziku jesu: metaformula, metaizraz, metaoperacije, metapromenljiva i metakonstanta. Pomoću metaformula opisujemo sve sintaksne konstrukcije (pojmove) u izvornom (programskom) jeziku. Opišimo sve ove pojmove Bekusove notacije (polazeći od najjednostavnijih).

**Metakonstanta** (terminal, terminalni simbol) jeste jedan simbol ili niska simbola azbuke jezika koji opisujemo. Dakle, metakonstanta se na jedinstven način zapisuje u metajeziku preuzimanjem iz izvornog jezika. Da bi se to naglasilo (i ako ima potrebe) metakonstanta se može uokviriti znacima navoda (“) ili apostrofom (‘).

**Metapromenljiva** (neterminal, neterminalni simbol) u Bekusovoj notaciji jeste pojam (sintaksna jedinica) izvornog jezika. Metapromenljiva se mora definisati pomoću metaformule, jer se ne može preuzeti iz izvornog jezika na jedan jedini način kao metakonstanta. Metapromenljiva se zapisuje kao fraza prirodnog stavljena između ugalstih zagrada ( < , > ). Time se u Bekusovoj notaciji imenuje pojam iz izvornog jezika (Npr. <program>,<naredba>,<izraz>,<slovo>).

**Metaizraz** je:

1. metakonstanta ,
2. metapromenljiva ili
3. konačan niz metakonstanti i/ili metapromenljivih koje su međusobno razdvojene metaoperacijama.

**Metaoperacija** može biti operacija spajanja (nadovezivanja, katenacije) ili razdvajanja (izbora, alternacije).

***Operacija spajanja*** (nadovezivanja) piše se u obliku $\alpha\beta$ i čita se "$\alpha$ za kojim sledi $\beta$" , gde su $\alpha$ i $\beta$ metakonstante ili metapromenljive (ili metaizrazi u opštem slučaju sa zagradama ).

***Operacija razdvajanja*** (izbora,alternacije) piše se u obliku $\alpha|\beta$ i čita "$\alpha$ ili $\beta$". Simbol operacije razdvajanja "|" je simbol metajezika. Operacija nadovezivanja ima viši prioritet od operacije razdvajanja. Tako zapis $\alpha\beta|\gamma$ predstavlja izbor od dve mogućnosti $\alpha\beta$ ili $\gamma$, što znači da se najpre realizuje spajanje.

***Metaformula*** (definiciona jednačina, gramatičko pravilo, (pravilo) produkcija(e), pravilo redukcije, pravilo zamene, itd.) jeste uređeni par $(\alpha,\beta)$ , koji se zapisuje u obliku

$$\alpha ::= \beta ,$$

i čita "$\alpha$ po definiciji je $\beta$" (ili "$\alpha$ to je $\beta$"),gde je $\alpha$ metapromenljiva , $\beta$ metaizraz a metasimbol ::= jeste simbol (operacije) definisanja.

Alternativni simboli definisanja mogu biti : $\rightarrow$ , $\leftarrow$ , = , := .

Metapromenljiva $\alpha$ na levoj strani metaformule predstavlja pojam (konstrukciju) u izvornom jeziku koji se definiše. Metaizraz $\beta$ definiše postupak generisanja ispravne konstrukcije na osnovu jednostavnijih pojmova (metaoperanada) primenjujući operacije izbora i nadovezivanja. Rezultujuća vrednost metaizraza jeste metakonstanta (niska terminalnih simbola) koja se može dodeliti (upotrbiti) matapromenljivoj $\alpha$ sa leve strane metaformule.

Metaizraz se može upotrebiti za generisanje svih mogućih sintaksno korektnih terminalnih niski koje predstavlja pojam $\alpha$.

Metaformule se mogu,dakle,upotrebiti kao gramatička pravila za građenje ispravnih konstrukcija u izvornom jeziku (potreba programera). Druga njihova uloga je da se mogu koristiti za proveru sintaksne korektnosti prethodno napisanih terminalnih niski (potreba prevodilaca).

*Primer* 1. Opisati pojam celog broja u programskom jeziku.

1. <ceo broj> ::= <znak broja> <ceo broj bez znaka> | <ceo broj bez znaka>
2. <znak broja> ::= + | –
3. <ceo broj bez znaka> ::= <cifra> <ceo broj bez znaka> | <cifra>
4. <cifra> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 .

## Proširena Bekusova Normalna Forma (PBNF)

U literaturi se za opis sintakse različitih jezika koriste i neki drugi načini zapisa (varijante Bekusove notacije). Ukazaćemo na neka korisna proširenja koja olakšavaju i skraćuju zapis (metaformula). Ovakve nove mogućnosti imaju istu moć kao i Bekusova normalna forma,ali su pogodnije za korišćenje. Proširenja uvode ( male , srednje i velike ) zagrade kao i neke druge mogućnosti.

### 1. Male zagrade

Ukazali smo da operacija spajanja (nadovezivanja) ima viši prioritet od operacije razdvajanja. Ukoliko

želimo da promenimo ovaj prioritet tada možemo upotrebiti male zagrade kao u aritmetičkom izrazu. Tako izraz $\alpha ( \beta | \gamma )$ znači izbor $\alpha\beta$ ili $\alpha\gamma$ , ( za razliku od zapisa $\alpha\beta | \gamma$ ). Na taj način možemo reći da je operacija spajanja distributivna u odnosu na operaciju razdvajanja. Obrnuto ne važi jer je $(\alpha\beta) | \gamma = \alpha\beta | \gamma$ zbog uvedenog prioriteta.

Male zagrade možemo upotrebiti kao metasimbole u cilju skraćivanja zapisa (faktorizacije) u sledećem slučaju.

Metaformulu

$$\alpha ::= xy | xz | \ldots | xt$$

možemo zapisati u obliku

$$\alpha ::= x ( y | z | \ldots | t )$$

što je mnogo preglednije. I ovo je primena distribucije prema razdvajanju.

Zagrade se mogu i dalje umetati,na sličan način kao kod aritmetičkih izraza. Na primer, ako je $y = f\, y_1$, $z = f\, z_1$ , tada možemo zapisati

$$\alpha ::= x ( f ( y_1 | z_1 ) | \ldots | t ).$$

### 2. Srednje zagrade

Srednje zagrade koristimo kada želimo da ukažemo na opcionu (fakultativnu) nisku (konstrukciju). To znači da se takva niska može ali ne mora upotrebiti. Tako u opisu celog broja znak broja je opciona konstrukcija pa možemo pisati:

< ceo broj > ::= [ < znak >] < ceo broj bez znaka > .

U opštem slučaju možemo pisati :

$$[ \alpha ] = \lambda | \alpha .$$

### 3. Velike zagrade

Pomoću rekurzije (rekurzivnih pravila) u Bekusovoj normalnoj formi možemo definisati (opisati) listu (spisak,niz) proizvoljno mnogo elemenata. Tako možemo pisati

( 1 ) < ceo broj bez znaka > ::= < cifra > < ceo broj bez znaka > | < cifra >
( 2 ) <spisak imena> ::= < ime > | < ime > , < spisak imena > .

Velike zagrade možemo upotrebiti kao metasimbole da bi ukazali da se niska uokvirena velikim zagradama može ponoviti proizvoljan broj puta i da se može izostaviti. U tom slučaju gornja pravila možemo napisati na sledeći način:

( 1’ ) < ceo broj bez znaka > ::= < cifra > { < cifra > }
( 2’ ) < spisak imena > ::= < ime > { , < ime > } .

Za proizvoljnu konstrukciju (nisku) $\alpha$ možemo pisati

$$\{\alpha\} = \lambda | \alpha | \alpha\alpha | \alpha\alpha\alpha | \ldots$$

To znači da zapis $\{\alpha\}$ predstavlja drugi način zapisa za iteraciju $\{\alpha\}^* = A^*$ jednoelementnog skupa $A=\{\alpha\}$, i pišemo

$$\{\alpha\}^* = \{\lambda\} \cup \{\alpha\} \cup \{\alpha\} \cup \{\alpha\} \cup \ldots$$
$$= \{\lambda\} \cup \{\alpha\} \cup \{\alpha\} \cup \{\alpha\alpha\alpha\} \cup \ldots .$$

Zato umesto $\{\alpha\}$ možemo pisati i $\{\alpha\}^*$. Takođe zapis $\{\alpha\}$ za proizvoljnu iteraciju možemo upotrebiti umesto $\alpha|\alpha\alpha|\alpha\alpha\alpha|. \ldots$

Pomoću gornjuh i donjih indeksa možemo zadati maksimalni i minimalni broj ponavljanja neke konstrukcije.

Sreću se i sledeći zapisi za konačne liste elemenata koji se ponavljaju pomoću tri tačke:

$$e_1 , e_2 , \ldots , e_n$$

## 0. 4. 1. SINTAKSNI DIJAGRAMI

Sintaksni dijagrami su jedna vrsta grafičkog metajezika jer koriste grafičke simbole za opis sintakse jezika. Zato su oni pregledniji i čitljiviji od Bekusove notacije. Ovaj jedinstven metajezik je ekvivalentan sa Bekusovom normalnom formom. I proširene Bekusove normalne forme se lako realizuju pomoću sintaksnih dijagrama.

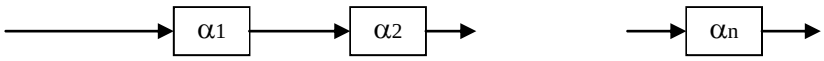
Sintaksni dijagram je imenovani orijentisani graf sa jednim ulazom i jednim izlazom sa čvorovima grafa koji predstavljaju terminalne ili neterminalne simbole. Svaki prolaz kroz sintaksni dijagram od ulaza prema izlazu generiše jednu sintaksno pravilnu konstrukciju jezika. Linije (sa strelicama) koje spajaju čvorove grafa realizuju metaoperacije nadovezivanja i razdvajanja (izbora) na prirodan način.

Za terminalni simbol *t* koristimo grafički simbol kružnog ili elipsoidnog oblika. Za neterminalne simbole predviđen je pravougaoni simbol..

Metaoperacija iz Bekusove notacije i proširenja (upotreba zagrada) se realizuju na jedostavan način. U nastavku navodimo ekvivalentne realizacije operacija u oba metajezika.

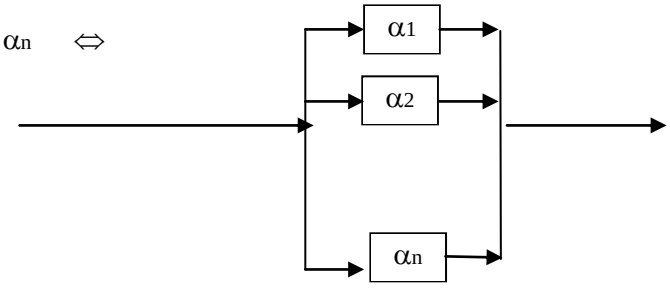
1. Spajanje (nadovezivanje) :

$\alpha_1\alpha_2 \ldots \alpha_n \quad \Leftrightarrow \quad \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \ldots \rightarrow \alpha_n \rightarrow$




2. Razdvajanje (izbor) :

$\alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n \quad \Leftrightarrow$




3. Iteracija (pozitivna iteracija) :

$\{\alpha\}^* \Leftrightarrow \{\alpha\} = \lambda \mid \alpha \mid \alpha\alpha \mid \ldots$
$\{\alpha\} \Leftrightarrow \alpha \mid \alpha\alpha \mid \alpha\alpha\alpha \mid \ldots$

4. Opciona konstrukcija :

$[\alpha] = \lambda \mid \alpha$

5. Ograničeno ponavljanje :

$\{\alpha\} = \lambda \mid \alpha \mid \alpha\alpha \mid \ldots \mid \alpha\alpha \ldots \alpha$
$\{\alpha\} = \alpha \mid \alpha\alpha \mid \ldots \mid \alpha\alpha \ldots \alpha$

# 1. Osnovni elementi programskih jezika

## 1.1. Pseudojezik kao opšti model viših programskih jezika

Za definiciju pseudojezika kao opšteg modela viših programskih jezika neophodno je obuhvatiti sledeće četiri fundamentalne komponente:

(1) Tipovi i strukture podataka koje pseudojezik podržava:

- statički skalarni tipovi,
- statički strukturirani tipovi,
- dinamički tipovi sa promenljivom veličinom,
- dinamički tipovi sa promenljivom strukturom.

(2) Osnovne kontrolne strukture koje se primenjuju u pseudojeziku:

- sekvenca,
- selekcije,
- ciklusi,
- skokovi.

(3) Operacije ulaza i izlaza podataka:

- ulaz/izlaz podataka za ulazno-izlazne uređaje i datoteke,
- konverzija tekstualnog i binarnog formata podataka.

(4) Tehnike modularizacije programa:

- nezavisne funkcije i procedure,
- interne funkcije i procedure,
- rekurzivne funkcije i procedure.

Razni tipovi podataka neophodni su u programskim jezicima da bi podražavali razne tipove objekata koje srećemo u matematičkim modelima. Podaci mogu biti skalarni ili strukturirani. Podatak je strukturiran ukoliko se sastoji od više komponenti koje se nalaze u precizno definisanom odnosu. Primer strukturiranog objekta je pravougaona matrica realnih brojeva kod koje svaki element predstavlja komponentu koja se nalazi u određenom odnosu sa ostalim komponentama. Podatak je skalaran ukoliko se ne sastoji od jednostavnijih komponenti. Jedan od skalarnih tipova podataka na svim programskim jezicima je celobrojni tip podataka. Lako je uočiti da za svaki tip podataka postoje operacije koje za njih važe, a ne važe za druge tipove podataka. Tako je na primer inverzija matrice operacija koja se ne primenjuje na celobrojne skalare, na isti način kao što se operacija celobrojnog deljenja dva skalara ne može primeniti na matrice.

Osnovne kontrolne strukture su izuzetno važna komponenta svakog programskog jezika. Pomoću njih se određuje redosled izvršavanja operacija koje računar obavlja. Osnovni tipovi kontrolnih struktura su sekvenca kod koje se instrukcije obavljaju onim redosledom kojim su napisane u programu, i strukture sa grananjima, ciklusima i skokovima. Grananja se koriste za izbor jednog od više paralelnih programskih segmenata, ciklusi realizuju ponavljanje nekog niza instrukcija, a skokovi služe za kontrolisani izlaz iz petlji, iz programa (kraj rada) i za realizaciju pojedinih osnovnih kontrolnih struktura.

Da bi se omogućilo komuniciranje računara sa spoljnim okruženjem potrebne su instrukcije ulaza i izlaza podataka. Pri tome podaci mogu da se učitavaju sa tastature ili iz datoteka i da se prenose na ekran, štampač, ili u datoteke.

Pod datotekom (*file*) podrazumevamo osnovnu organizaciju podataka koja obuhvata proizvoljan broj manjih jednoobrazno strukturiranih celina koje se nazivaju zapisi. Svaki zapis sadrži podatke o jednoj jedinici posmatranja. Na primer, zapis može sadržati strukturirane podatke o motornom vozilu (tip, godina proizvodnje, snaga motora, vlasnik, itd.), dok skup većeg broja ovakvih zapisa predstavlja datoteku motornih vozila. Alfanumerički podaci (slova, cifre i specijalni znaci) se čuvaju u tekst datotekama, a numerički podaci mogu da se čuvaju bilo u tekstualnoj bilo u kompaktnoj binarnoj formi. Za sve podatke postoji format kojim se određuje njihova struktura, a pri nekim prenosima podataka može se automatski vršiti i konverzija formata.

Ako bi programi bili formirani kao neprekidni nizovi instrukcija, onda bi kod velikih programa bilo nemoguće razumevanje načina rada celine, i na taj naćin bilo bl znatno otežano održavanje programa. Zbog toga su mehanizmi modularizacije programa od vitalnog značaja za uspeh nekog programskog jezika. Pod modularizacijom programa podrazumevamo razbijanje programa na manje (najčešće nezavisne) celine kod kojih su precizno definisani ulazni i izlazni podaci, kao i postupak kojim se na

osnovu ulaznih podataka dobijaju izlazni podaci. U ovu grupu spadaju nerekurzivne i rekurzivne funkcije i procedure.

Prilikom definicije jezika polazi se od osnovnog skupa znakova, azbuke jezika koja sadrži sve završne simbole (terminalne simbole) jezika. Nad azbukom jezika definišu se ostali elementi jezika, konstante, rezervisane reči, identifikatori od kojih se dalje grade druge složene sintaksne kategorije kao što su na primer opisi, upravljačke naredbe i slično.

## 1.2. Azbuka jezika

Azbuka jezika predstavlja osnovni skup simbola (znakova) od kojih se grade sve sintaksne kategorije jezika. Broj znakova azbuke se obično razlikuje od jezika do jezika i kreće se od najmanje 48 do 90 znakova. Azbuka programskog jezika obično obuhvata skup velikih i malih slova engleske abecede, skup dekadnih cifara i određeni skup specijalnih znakova. Dok je kod starijih programskih jezika bilo uobičajeno da se koriste samo velika slova (na primer FORTRAN IV), danas se skoro redovno dozvoljava i ravnopravno korišćenje malih slova abecede. Azbuke programskih jezika se najviše razlikuju po skupu specijalnih znakova koje obuhvataju. Narednih nekoliko primera azbuka programskih jezika to ilustruje.

**Azbuka jezika C**

velika slova: A | B| C | D| E| F|G|H|I|J|K|L|M|N|0| P|Q| R|S|T|U|V|W|X|Y|Z
cifre: 0 |1 |2|3|4|5|6|7|8|9
specijalni znaci: + | - | * | / |= | ( | ) | { \} \[ | ] \< | > |' | " | ! | # | \ | % | & | | | |_ | ^| , | ~ \, \ ; | : | ?
znak blanko
mala slova: a | b | c | d | e| f|g|h| i| j| k|l|m|n|o|p|q|r|s| t|u| v|w|x|y|z

Danas se obično skup specijalnih znakova azbuke programskog jezika standardizuje i svodi na skup znakova međunarodnog standardnog koda ISO7 (ASCII kod).

b | ! | " | $ | % | & | ' | ( | ) | * | + | , | - | . | / | : | ; | < | = | > | ? | @ | [ | ] | \ | ^ ili _ | ` | { | } | ~

Često se pored osnovnog skupa specijalnih znakova koriste i složeni simboli, obično dvoznaci, kao na primer:

| ** | >= | <= | => | =< | << | <> | >< | := | -> | /* | */ |

U nekim programskim jezicima (FORTRAN), zbog nedovoljnog broja odgovarajućih znakova umesto specijalnih znakova koriste se posebne simboličke oznake .EQ., .NE., .GT., .GE., .LT., .LE., kojima se označavaju relacije jednako, različiti, veće, veće ili jednako, manje i manje ili jednako, redom.

## 1.3. Identifikatori i rezervisane reči

Identifikatori su uvedene reči kojima se imenuju konstante, promenljive, potprogrami, programski moduli, klase, tipovi podataka i slično.

U svim programskim jezicima postoji slična konvencija za pisanje identifikatora. Identifikatori su nizovi koji se obično sastoje od slova i cifara i obavezno započinju slovom. Ovo ograničenje omogućava jednostavniju implementaciju leksičkog analizatora i razdvajanje identifikatora od drugih sintaksnih kategorija (numeričkih i znakovnih konstanti na primer). U nekim jezicima dozvoljeno je da se i neki specijalni znaci pojave u okviru identifikatora. Najčešće je to crtica za povezivanje "_" kojom se postiže povezivanje više reči u jedan identifikator. Na primer, u jeziku C crtica za povezivanje se može koristiti na isti način kao i slova, što znači da identifikator može da započne ovim znakom. Slede primeri nekih identifikatora:

```
ALFA  A  B1223  Max_vrednost  PrimerPrograma
```

U jeziku C velika i mala slova se razlikuju. Programski jezik PASCAL ne razlikuje velika i mala slova.

Dobra je programerska praksa da identfikatori predstavljaju mnemoničke skraćenice.

Nizovi znakova azbuke koji u programu imaju određeni smisao nazivaju se lekseme. Leksema može da bude i samo jedan znak.

Reč jezika čije je značenje utvrđeno pravilima tog jezika naziva se *rezervisana reč*.

Rezervisane reči mogu da budu zabranjene, kada se ne mogu koristiti kao identifikatori u programu. Takav je slučaj u programskom jeziku C. Međutim, i u jezicima u kojima je to dozvoljeno ne preporučuje se korišćenje ključnih reči kao identifikatora jer može da smanji preglednost programa, a u nekim slučajevima da dovede i do ozbiljnih grešaka u programu. Poznat je, na primer, slučaj greške sa DO naredbom koji je doveo do pada letilice iz satelitskog programa Geminy 19. U programu za upravljanje letilicom stajala je DO naredba napisana kao:

```
DO 10 I = 1.10
```

umesto ispravnog koda

```
DO 10 I = 1,10.
```

Greška pri prevođenju međutim nije otkrivena jer je leksički analizator ovu liniju kôda protumačio kao naredbu dodeljivanja

```
DO10I  = 1.10
```

u kojoj se promenljivoj D010I dodeljuje vrednost 1.10. Greška je otkrivena tek u fazi izvršavanja programa kada je prouzrokovala pad letilice.

**Rezervisane reči jezika C:**
auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, int, long, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, voatile, while.

U poređenju sa ostalim programskim jezicima, C poseduje mali broj službenih reči.

## 1.4. Konstante

Bilo koji niz znakova u programu, posmatran nezavisno od njegovog logičkog značenja, nad kojim se mogu izvršavati određena dejstva (operacije) naziva se *podatak*. Deo podatka nad kojim se mogu izvršavati elementarne operacije naziva se *element podatka*. Elementu podatka u matematici približno odgovara pojam skalarne veličine. Podatak je uređeni niz znakova kojim se izražava vrednost određene veličine.

Veličina koja u toku izvršavanja programa uvek ima samo jednu vrednost, koja se ne može menjati, naziva se *konstanta*. Kao oznaka konstante koristi se ona sama.

U nekim programskim jezicima (Pascal, Ada, C) postoji mogućnost imenovanja konstante. Konstantama se dodeljuju imena koja se u programu koriste umesto njih. Na taj način nastaju *simboličke konstante*.

Tipovi konstanti koje se mogu koristiti u određenom programskom jeziku određeni su tipovima podataka koje taj jezik predviđa.

U jeziku C se koristi više vrsta konstanti i to su:

- celobrojne konstante; 2, 3, +5, -123
- karakteri: 'a', 'b', ... ;
- stringovi: sekvence karaktera između navodnika.

Slede neki primeri različitih vrsta konstanti:

Celobrojne dekadne konstante: `1; 50; 153; +55; -55`

Realne konstante u fiksnom zarezu: `3.14; 3.0; -0.314; -.314; +.314`

Realne konstante u pokretnom zarezu:

`3.14E0; -0.314E1; -.314E+0; +.314E-2` (FORTRAN, Pascal, Ada, C)

Realne konstante dvostruke tačnosti:

Oktalne konstante (C): `0567; 0753; 0104`

Heksadecimalne konstante (C): `0X1F; 0XAA; 0X11`

Long konstante (C): `123L; 527L;321L;+147L`

Logičke konstante: `true; false` (Pascal, Ada)

`.TRUE.; .FALSE.` (FORTRAN)

Znakovne konstante: `'A'; 'B'` (Pascal, Ada, C)

String konstante: `"Beograd"; "Alfa 1"` (C)

String konstante: `'Beograd'; 'Alfa 1'` (Pascal)

Simboličke konstante: `ZERO; ZEROS; SPACE` (COBOL)

`Pi, E` (MATHEMATICA)

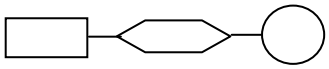
Racionalni brojevi: `2/3; -4/5` (MATHEMATICA)

# 1.5. Promenljive

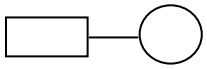
Veličine čije se vrednosti menjaju u toku izvršavanja programa nazivaju se promenljive. Promenljivoj se u programu dodeljuje ime, i u svakom trenutku ona je definisana svojom vrednošću. Kažemo da je svaka promenljiva u programu povezana sa tri pojma:

imenom - identifikatorom promenljive.
referencom - pokazivačem koji određuje mesto promenljive u memoriji i
vrednošću - podatkom nad kojim se izvršavaju operacije.

Veza između imena, reference i vrednosti promenljive može se predstaviti sledećim dijagramom:




Ovaj dijagram u slučaju imenovane konstante dobija oblik:




Na primer naredbu dodeljivanja `X:=3.141592` čitamo: *X* dobija vrednost 3.141592, pri čemu imamo u vidu da je *X* simboličko ime memorijske lokacije gde se u toku izvršavanja programa pamti vrednost 3.141592. Pri tome je potrebno imati na umu sledeće pojmove:

- vrednost 3.141592, koja predstavlja vrednost promenljive *X*,
- adresu memorijske lokacije u kojoj se pamti vrednost 3.141592,
- ime promenljive *X*, identifikator koji se u datom programu koristi kao ime promenljive koja ima datu brojnu vrednost.

# 1.6. Komentari

U svim programskim jezicima postoji mogućnost proširenja izvršnog koda programa komentarima kojima se kod dodatno pojašnjava. Ovi komentari se u toku prevođenja programa ignoršu od strane kompilatora i ne ulaze u sastav izvršnog koda koji se generiše prevođenjem programa. Međutim komentari su veoma važan deo programa kojim se podešava njegova dokumentarnost, i bitno utiču na efikasnost analize programa. Konvencija za zapisivanje komentara se razlikuje od jezika od jezika. Slede primeri komentara u nekim programskim jezicima:

```
/* Ovo je primer komentara u jeziku C */
// Kratak C++ komentar u okviru jednog reda
```

## 1.7. Struktura programa

Globalna struktura programa zavisi od toga da li jezik zahteva deklaraciju tipova promenljivih kao i da li su u jeziku zastupljeni stariji koncepti sa implicitnim definicijama tipova promenljivih ili je zastupljen noviji koncept sa eksplicitnim definicijama.

### 1.7.1. Struktura programa u C

Programi pisani u C jeziku imaju strukturu bloka, ali se za ograničavanje bloka koriste vitičaste zagrade umesto zagrada *begin* i *end*. Zastupljen je takođe stariji koncept bloka, poznat iz jezika Algol 60 i PL/1, gde se opisi elemenata koji pripadaju jednom bloku nalaze unutar zagrada { i }, za razliku od novijeg koncepta koji se koristi u jeziku Pascal, kod kojih opisi elemenata bloka prethode zagradi *begin* kojom se otvara blok. Svaki blok u C-u takođe može da sadrži opise promenljivih i funkcija.

C program se može nalaziti u jednom ili više fajlova. Samo jedan od tih fajlova (glavni programski modul) sadrži funkciju *main* kojom započinje izvršenje programa. U programu se mogu koristiti funkcije iz standardne biblioteke. U tom slučaju se može navesti direktiva pretprocesora oblika

```
#include <ime>.
```

česta je direktiva oblika #include<stdio.h> kojom se uključuju funkcije iz fajla stdio.h (standard input/output header).

Glavni program, main(), takođe predstavlja jednu od funkcija. Opis svake funkcije se sastoji iz zaglavlja i tela funkcije. U ovom slučaju, zaglavlje funkcije je najjednostavnije, i sadrži samo ime funkcije i zagrade `()`. Iza zaglavlja se navodi telo funkcije koje se nalazi između zagrada { i }. Između ovih zagrada se nalaze operatori koji obrazuju telo funkcije. Svaki prost operator se završava znakom ';' a složeni operator se piše između zagrada { i }.

U jeziku C sve promenljive moraju da se deklarišu. Opšti oblik jednostavnog programa je:

```
void main()
{
 <deklaracije>
 <naredbe>
}
```

# 2. TIPOVI PODATAKA

Jedan od najznačajnijih pojmova u okviru programskih jezika je pojam tipa podataka. Atribut tipa određuje skup vrednosti koje se mogu dodeljivati promenljivima, format predstavljanja ovih vrednosti u memoriji računara, skup osnovnih operacija koje se nad njima mogu izvršavati i veze sa drugim tipovima podataka. Na primer, promenljivoj koja pripada celobrojnom tipu mogu se kao vrednosti dodeljivati samo celi brojevi iz određenog skupa. Nad tako definisanim podacima mogu se izvršavati osnovne aritmetičke operacije sabiranja, oduzimanja, mnozenja, deljenja, stepenovanja, kao i neke specifične operacije kao što je određivanje vrednosti jednog broja po modulu drugog.

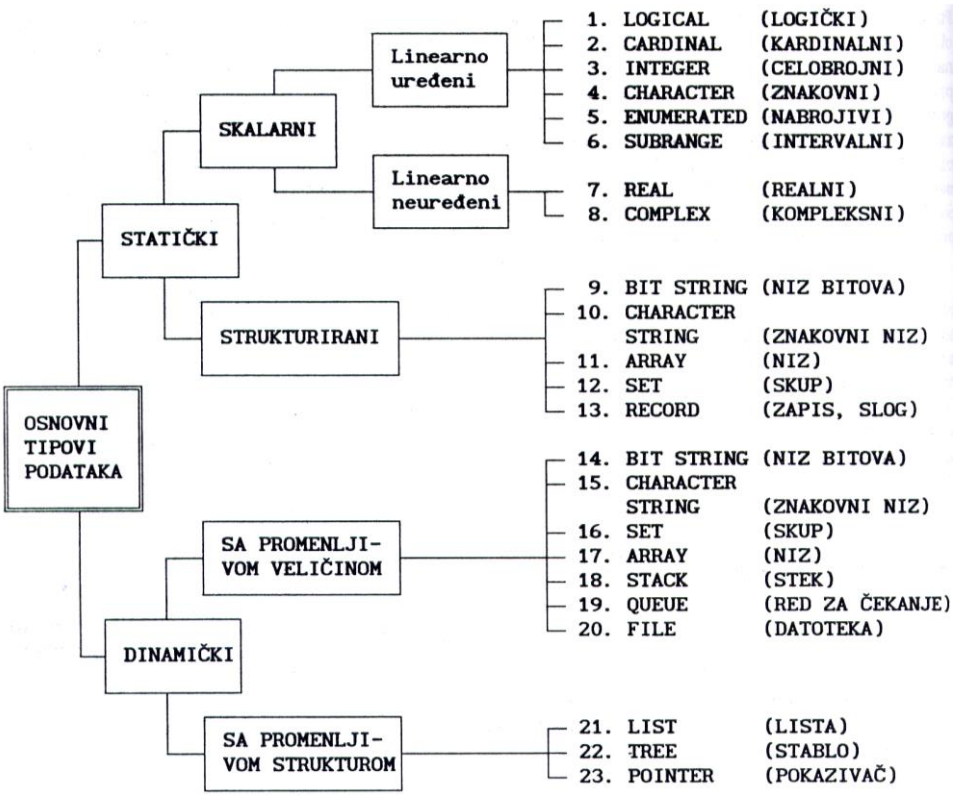
Koncept tipova podataka prisutan je, na neki način, već kod simboličkih asemblerskih jezika gde se za definiciju tipa koriste implicitne definicije preko skupa specijalnih znakova kojima se određuju podaci različitog tipa.

Neka je zadat skup $T$ sačinjen od $n$ proizvoljnih apstraktnih objekata: $T:=\{v_1,...,v_n\}$, $n>1$. Ukoliko su svi objekti istorodni, u smislu da se u okviru nekog programskog jezika na njih može primenjivati jedan određeni skup operatora, onda se $T$ naziva tip podataka, Ako za promenljive $x$ i $y$ uvek važi $x\in T$ i $y\in T$ onda su one tipa $T$. To u okviru programa formalno definišemo iskazom

```
DEFINE x, y :  T.
```

**Vrednost** je bilo koji entitet kojim se može manipulisati u programu. Vrednosti se mogu evaluirati, zapamtiti u memoriji, uzimati kao argumenti, vraćati kao rezultati funkcija, i tako dalje. Različiti programski jezici podržavaju različite tipove vrednosti.

Jedna od klasifikacija osnovnih tipova podataka prikazana je na slici.




Tipovi podataka su podeljeni u dve osnovne grupe: **statički** tipovi i **dinamički** tipovi. Pod statičkim tipovima (ili tipovima sa statičkom strukturom podrazumevamo tipove podataka kod kojih je unapred i fiksno definisana unutrašnja struktura svakog podataka, a veličina (t.j. memorijska zapremina) fiksno se definiše pre (ili u vreme) izvršavanja programa koji koristi podatke statičkog tipa.

Statički tipovi podataka obuhvataju **skalarne** i **strukturirane** podatke. Pod **skalarnim** tipovima podrazumevamo najprostije tipove podataka čije su vrednosti skalari, odnosno takve veličine koje se tretiraju kao elementarne celine i za koje nema potrebe da se dalje razlažu na komponente. U tom smislu realne i kompleksne brojeve tretiramo kao skalare u svim programskim jezicima koji ih tretiraju kao elementarne celine (neki jezici nemaju tu mogućnost pa se tada kompleksni brojevi tretiraju kao struktura koja se razlaže na realni i imaginarni deo; na sličan način mogao bi se realni broj razložiti na normalizovanu mantisu i eksponent). Pod **strukturiranim** tipovima podataka podrazumevamo sve složene tipove podataka koji se realizuju povezivanjem nekih elementarnih podataka u precizno definisanu strukturu. U ulozi elementarnih podataka obično se pojavljuju skalari ili neke jednostavnije strukture. Kompozitna vrednost ili struktura podataka (data structure) jeste vrednost koja je komponovana iz jednostavnijih vrednosti. Kompozitni tip jeste tip čije su vrednosti kompozitne. Programski jezici podržavaju mnoštvo kompozitnih vrednosti: strukture, slogove, nizove, algebarske tipove, objekte, unije, stringove, liste, stabla, sekvuencijalni fajlovi, direktni fajlovi, relacije, itd.

Skalarni tipovi podataka mogu biti **linearno uređeni** ili **linearno neuređeni**. Linearno uređeni tipovi podataka su tipovi kod kojih se vrednosti osnovnog skupa $T$ preslikavaju na jedan interval iz niza celih brojeva, t.j. za svaki podatak $x \in T$ zna se redni broj podatka. Stoga svaki podatak izuzev početnog ima svog prethodnika u nizu, i slično tome, svaki podatak izuzev krajnjeg ima svog sledbenika u nizu.

Pod **dinamičkim** tipovima podataka podrazumevamo tipove podataka kod kojih se veličina i/ili struktura podataka slobodno menja u toku obrade. Kod dinamičkih tipova sa promenljivom veličinom podrazumevamo da je struktura podataka fiksna, ali se njihova veličina dinamićki menja tokom obrade tako da se saglasno tome dinamički menjaju i memorijski zahtevi. Na primer, dopisivanjem novih zapisa u sekvencijalnu datoteku veličina datoteke raste uz neizmenjenu strukturu. Kod dinamičkih tipova sa promenljivom strukturom unapred je fiksno definisan jedino princip po kome se formira struktura podataka dok se sama konkretna struktura i količina podataka u memoriji slobodno dinamički menjaju.

## Statička i dinamička tipizacija

Pre izvršenja bilo koje operacije, moraju se proveriti tipovi operanada da bi se sprečila greška u tipu podataka. Na primer, pre izvršenja operacije množenja dva cela broja, oba operanda moraju biti proverena da bi se osiguralo da oni budu celi brojevi. Slično, pre izvršenja neke *and* ili *or* operacije, oba operanda moraju biti proverena da bi se osiguralo da oni budu tipa *boolean*. Pre izvršenja neke operacije indeksiranja niza, tip operanda mora da bude *array* (a ne neka prosta vrednost ili slog). Ovakva provera se naziva proverom tipa (***type checks***). Provera tipa mora da bude izvršena pre izvršenja operacije.

Međutim, postoji izvestan stepen slobode u vremenu provere: provera tipa se može izvršiti ili u vremenu kompilacije (compile-time) ili u vremenu izvršenja programa (at run-time). Ova mogućnost leži u osnovi važne klasifikacije programskih jezika na **statički tipizirane** (statically typed) i **dinamički tipizirane** (dynamically typed). U nekom statički tipiziranom jeziku, svaka varijabla i svaki izraz imaju fiksni tip (koji je ili eksplicitno postavljen od strane programera ili izveden od strane kompajlera). Svi operandi moraju biti proverenog tipa (type-checked) u vremenu kompilovanja programa (*at compile-time*). U dinamički tipiziranim jezicima, vrednosti imaju fiksne tipove, ali varijable i izrazi nemaju fiksne tipove. Svaki put kada se neki operand izračunava, on može da proizvede vrednost različitog tipa. Prema tome, operandi moraju imati proveren tip posle njihovog izračunavanja, ali pre izvršenja neke operacije, u vremenu izvršenja programa (*at run-time*). Mnogi jezici visokog nivoa su statički tipizirani. SMALLTALK, LISP, PROLOG, PERL, i PYTHON jesu primeri dinamički tipiziranih jezika.

S druge strane, moderni funkcionalni jezici (kao ML, HASKELL, MATHEMATICA) omogućavaju da sve vrednosti, uključujući i funkcije, imaju sličnu obradu.

Izbor između statičke i dinamičke tipizacije je pragmatičan:

• Statička tipizacija je efikasnija. Dinamička tipizacija zahteva (verovatno ponovljenu) proveru tipova u vremenu izvršenja programa (run-time type checks), što usporava izvršenje programa. Statička tipizacija zahteva jedino proveru tipa u vremenu kompilacije programa (compile-time type checks), čija je cena minimalna (i izvršava se jedanput). Osim toga, dinamička tipizacija primorava sve vrednosti da budu etiketirane (tagged) (da bi se omogućila provera u vreme izvršenja), a ovakvo označavanje povećava upotrebu memorijskog prostora. Statička tipizacija ne zahteva ovakvo označavanje.

• Statička tipizacija je sigurnija: kompajler može da proveri kada program sadrži greške u tipovima. Dinamička tipizacija ne omogućava ovakvu sigurnost.

• Dinamička tipizacija obezbeđuje veliku fleksibilnost, što je neophodno za neke aplikacije u kojima tipovi podataka nisu unapred poznati.

U praksi veća sigurnost i efikasnost statičke tipizacije imaju prevagu nad većom fleksibilnošću dinamičke tipizacije u velikoj većini aplikacija. Većina programskih jezika je statički tipizirana.

Na osnovu toga kako je postavljen koncept tipova podataka, programski jezici mogu da se svrstaju u dve grupe: na programske jezike sa **slabim tipovima podataka** i na jezike sa **jakim tipovima podataka.**

# 2.1. Koncept jakih tipova podataka

Koncept jakih tipova podataka obuhvata nekoliko osnovnih principa:

- Tip podataka određuju sledeći elementi:
  - skup vrednosti,
  - format registrovanja podataka,
  - skup operacija koje se nad podacima mogu izvršavati,
  - skup funkcija za uspostavljanje veza sa drugim tipovima podataka.
- Sve definicije tipa moraju da budu javne, eksplicitne. Nisu dozvoljene implicitne definicije tipova.
- Objektu se dodeljuje samo jedan tip.
- Dozvoljeno je dodeljivanje vrednosti samo odgovarajućeg tipa.
- Dozvoljene su samo operacije obuhvaćene tipom.
- Tip je zatvoren u odnosu na skup operacija koji obuhvata. Ove operacije se mogu primenjivati samo nad operandima istog tipa. Mešoviti izrazi nisu dozvoljeni.
- Dodeljivanje vrednosti raznorodnih tipova moguće je samo uz javnu upotrebu funkcija za transformaciju tipa.

Koncept jakih tipova povećava pouzdanost, dokumentarnost i jasnoću programa. Kako se zahteva eksplicitna upotreba operacija za transformaciju tipa, onda je nedvosmisleno jasno da je određena transformacija na nekom mestu namerna i potrebna. Ovaj koncept omogućava da se informacija o tipu može koristiti u fazi kompilovanja programa i na taj način postaje faktor pouzdanosti programa.

# 2.2. Koncept slabih tipova

U slučaju jezika sa slabim tipovima podataka informacija o tipu promenljive koristi se, i korektna je samo na mašinskom nivou, u fazi izvršenja programa. Ovako postavljen koncept podrazumeva sledeće mogućnosti:

(1) Operacija koja se od strane kompilatora prihvati kao korektna, na nivou izvornog koda programa, može da bude potpuno nekorektna. Razmotrimo sledeći primer:

```
char c;
c  = 4;
```

Promenljiva *c* definisana je da pripada tipu *char*, što podrazumeva da joj se kao vrednosti dodeljuju znaci kao podaci. Međutim umesto korektnog dodeljivanja *c* = '4', promenljivoj *c* je dodeljena vrednost broja 4 kao konstante celobrojnog tipa. Kod jezika sa slabim tipovima podataka kompilator ne otkriva ovu grešku i informaciju o tipu koristi samo na mašinskom nivou kada promenljivoj *c* dodeljuje vrednost jednog bajta memorijske lokacije u kojoj je zapisana konstanta 4. Očigledno je da ovako postavljen koncept tipova može da dovede do veoma ozbiljnih grešaka u izvršavanju programa koje se ne otkrivaju u fazi kompilovanja programa.

(2) Koncept slabih tipova podrazumeva određeni automatizam u transformaciji tipova podataka u slučaju kada se elementi različitih tipova nalaze u jednom izrazu čija se vrednost dodeljuje promenljivoj određenog tipa. Razmotrimo sledeći primer:

```
float x, y;
int i, j, k;
i =  x;
k =  x-j ;
```

Promenljive *x* i *y* su realne (tipa *float*), a *i*, *j* i *k* celobrojne (tipa *int*). Naredbom `i=x;` vrši se dodeljivanje vrednosti tipa *float* promenljivoj celobrojnog tipa. Kod jezika sa slabim tipovima ovo dodeljivanje je dozvoljeno iako se pri tome *x* svodi na drugi format i pravi greška u predstavljanju njegove vrednosti. Kod ovako postavljenog koncepta tipova, da bi se napisao korektan program

potrebno je tačno poznavati mehanizme transformacije tipova. U drugoj naredbi iz primera (*k = x-j;*) od broja *x* koji je tipa float treba oduzeti broj *j*, tipa *int* i rezultat operacije dodeliti promenljivoj tipa *int*. Da bi smo bili sigurni u korektnost rezultata potrebno je da znamo redosled transformacija koje se pri tome izvršavaju, odnosno da li se prvo *x* prevodi u *int* i onda izvršava oduzimanje u skupu celih brojeva i vrednost rezultata dodeljuje promenljivoj tipa *int* ili se *j* prevodi u tip *float*, izvršava oduzimanje u skupu realnih brojeva, a zatim rezultat prevođi u tip *int* i dodeljuje promenljivoj *k*.

Koncept slabih tipova podataka dopušta puno slobode kod zapisivanja izraza u naredbama dodeljivanja; međutim cena te slobode je nejasan program sa skrivenim transformacijama, bez mogućnosti kontrole i korišćenja informacije o tipu u fazi kompilovanja programa.

# 2.3. Ekvivalentnost tipova

Šta ekvivalentnost tipova označava zavisi od programskog jezika. (Sledeća diskusija podrazumeva da je jezik statički tipiziran.) Jedna moguća definicija ekvivalentnosti tipova jeste **strukturna ekvivalentnost** (***structural equivalence***): $T1 \equiv T2$ ako i samo ako $T1$ i $T2$ imaju isti skup vrednosti. Strukturna ekvivalentnost se tako naziva zato što se ona može proveriti poređenjem struktura tipova $T1$ i $T2$. (Nepotrebno je, a u opštem slučaju i nemoguće, da se prebroje sve vrednosti ovih tipova.)

Kada kompilator jezika sa jakim tipovima podataka treba da obradi naredbu dodeljivanja oblika

x := izraz

on vrši dodeljivanje samo u slučaju ekvivalentnosti tipa promenljive sa leve strane dodeljivanja i rezultata izraza na desnoj strani naredbe, osim u slučaju kada je na desnoj strani celobrojni izraz a na levoj strani promenljiva nekog realnog tipa.

**Eksplicitnom ekvivalentnošću** tipova postiže se veća pouzdanost jezika. U ovom slučaju nisu potrebne posebne procedure po kojima bi se ispitivala strukturna ekvivalentnost. Međutim, kada je potrebno vrednost promenljive ili izraza dodeliti promenljivoj koja mu ne odgovara po tipu ovaj koncept zahteva korišćenje funkcija za transformisanje tipova.

# 2.4. Elementarni tipovi podataka

U okviru svakog programskog jezika, sistem tipova podataka zasniva se na skupu osnovnih tipova podataka nad kojima se dalje definišu izvedeni tipovi, podtipovi, strukturni tipovi i specifični apstraktni tipovi podataka. Skup osnovnih tipova podataka se obično svodi na tipove podataka za rad sa elementarnim numeričkim podacima (celi i realni brojevi), znakovnim podacima (pojedinačni znaci ASCII koda) i logičkim vrednostima (*true* i *false*).

## 2.4.1. Celobrojni tipovi (Integer ili int)

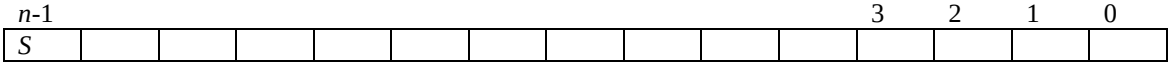
### Celobrojni tip podataka (tip INTEGER)

Podaci celobrojnog tipa pripadaju jednom lntervalu celih brojeva koji obuhvata pozitivne i negativne brojeve i koji se obično označava na sledeći način:

```
T := {minint, minint+1, ... , -1, 0, 1, ,.. , maxint-1, maxint } .
```

Ovde je najmanji broj označen sa *minint*, a najveći sa *maxint* (od engl. maximum integer i minimum integer) pri čemu ove veličine nisu fiksne već zavise od implementacije i prema tome variraju od računara do računara.

Od nekoliko načina binarnog kodiranja celih brojeva izdvojićemo metod potpunog komplementa koji se najčešće sreće u praksi. Kod ovog postupka brojevi se binarno predstavljaju pomoću sledeće *n*-bitne reči:



Bit najstarijeg razreda označen sa *S* (sign) predstavlja predznak broja. Ako je *S*=0 onda je je broj nenegativan, a ako je *S*=1 onda je broj negativan. Nula se označava sa nulama u svim bitovima:

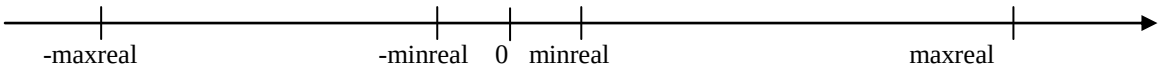
0000 0000 0000 0000

Obično postoji osnovni (standardni) tip INTEGER ili int, koji se zavisno od realizacije odnosi na određeni opseg celih brojeva. Nekad je to opseg koji odgovara formatu jedne polureči ili formatu jedne reči. U odnosu na ovaj osnovni celobrojni tip često postoji mogućnost definisanja i drugih celobrojnih tipova koji se odnose na neki kraći ili prošireni format.

## 2.4.2. Realni tip (float ili real)

Promenljive ovih tipova uzimaju za svoje vrednosti podskupove skupa realnih brojeva. U programskim jezicima postoji više vrsta podataka realnog tipa, koji se razlikuju po tačnosti predstavljanja podataka.

Realni tip podataka obuhvata jedan konačan podskup racionalnih brojeva ograničene veličine i tačnosti. Naravno, može se odmah postaviti pitanje zbog čega se koriste nazivi "realni tip" i "realni broj" za nešto što u opštem slučaju nije u stanju da obuhvati ni iracionalne brojeve ni beskonačne periodične racionalne brojeve. Ipak, to je terminologija koja je prihvaćena u praksi i opravdava se time što je (*float* ili *REAL*) tip podataka koji se najbliže približava pojmu realnog broja. Sa druge strane, lako je razumeti da su sve memorijske lokacije konačne dužine, pa stoga i brojni podaci koji se u njih smeštaju moraju biti konačne dužine i tako po prirodi stvari otpadaju iracionalni brojevi. Potrebe prakse nisu na ovaj način ni malo ugrožene jer se dovoljna tačnost rezultata može postići i sa veličinama konačne dužine. Tip *float* najčešće obuhvata brojne vrednosti iz sledećih podintervala brojne ose:




Ovde *minreal* označava najmanju apsolutnu vrednost veću od nule, koja se može predstaviti na raćunaru, a *maxreal* predstavlja najveću apsolutnu vrednost koja se može predstaviti na raćunaru. Realni brojevi iz intervala (-minreal, minreal) se zaokružuju i prikazuju kao 0, realni brojevi iz intervala (-∞, -maxreal) i (maxreal, +∞) ne mogu se predstaviti u memoriji računara, a -∞ i +∞ se kodiraju specijalnim kodovima.

## 2.4.3. Logički tip podataka

Logički tipovi podataka postoje kao osnovni tipovi podataka u svim novijim jezicima. Obično nose naziv LOGICAL (FORTRAN) ili BOOLEAN (Pascal, Ada). Obuhvataju samo dve vrednosti *true* i *false*, nad kojima su definisane osnovne logičke operaclje not, and, or i xor.

Takođe, važi i uređenost skupa vrednosti ovog tipa tako da je *false* < *true*.

Izrazi u kojima se primenjuju logičke promenljive i konstante nazivaju se logički izrazi. Ako se logičke konstante označavaju sa *false* i *true* onda podrazumevamo da svi izrazi moraju biti sačinjeni striktno od logičkih veličina. Ha primer, izraz z := ($x$>0) and ( ($y$=1) or ($y$<0) ) je korektan pri čemu se podrazumeva da su *x* i *y* celobrojne ili realne veličine, a *z* je velićina logičkog tipa. Podrazumeva se i da su neispravni mešoviti izrazi u kojima se koriste veličine *true* i *false* pomešane sa numeričkim konstantama i aritmetičkim operacijama. Na primer, nije definisan izraz 4*fa1se + 2*true + true, ali izraz 4*($x$>0) + 2*($y$>0) + ($z$>0) , koji je takođe besmislen kod logičkih konstanti *false* i *true*, u slučaju numeričkog kodiranja T={0,1} redovno ima i smisla i upotrebnu vrednost kao generator veličina 0,1,2,3,4,5,6,7.

## 2.4.4. Znakovni tipovi

Korisnik računara komunicira sa računarom preko ulaznih i izlaznih uređaja i tu je bitno da se pojavljuju podaci u formi koja je čitljiva za čoveka. To znači da se komuniciranje obavlja pomoću

znakova iz određene azbuke koja redovno obuhvata abecedno poređana velika i mala slova, cifre, specijalne znake i kontrolne znake. Pod specijalnim znacima se podrazumevaju svi oni znaci koji se javljaju na tastaturama i mogu odštampati, ali nisu ni slova ni cifre (na tastaturi sa kojom je pisan ovaj tekst specijalni znaci su ! # S % & *()_-+=:"; '<>?,/.). Pod kontrolnim znacima podrazumevaju se znaci koji se ne mogu odštampati (ili prikazati na ekranu terminala), već služe za upravljanje radom ulazno/izlaznog uređaja (na primer štampača). U ovu grupu spadaju specijalni znaci za pomeranje papira, znak koji izaziva zvučni signal na terminalu i drugi).

Da bi se znaci razlikovali od simboličkih naziva promenljivih obično se umeću između apostrofa (na nekim programskim jezicima umesto apostrofa se koriste znaci navoda). Tako se, na primer, podrazumeva da *A* predstavlja simbolički naziv promenljive, dok '*A*' predstavlja binarno kodirano prvo slovo abecede.

U praksi se primenjuje nekoliko metoda za binarno kodiranje znakova. Najpoznatiji metod je američki standard ASCII (American Standard Code for Information Interchange). Ovim smenama su neki manje važni znaci iz skupa ASCII znakova zamenjeni sa specifičnim jugoslovenskim znacima (na pr., umesto ASCII znakova \ | { } [ ] ~ @ i ^ u srpskoj varijanti se pojavljuju znaci Đ đ š ć š Ć č ž Ž i Č). Neki terminali imaju mogućnost podesivog izbora skupa znakova tako da korisnik može po potrebi izabratl američku ili jugoslovensku varijantu skupa ASCII znakova.

Prva 32 znaka u ASCII skupu su kontrolni znaci. Nekl od njlh imaju jedinstvenu interpretaciju kod svih uređaja, a kod nekih interpretacija se razlikuje od uređaja do uređaja. Ovde pominjemo sledeće:

| | | |
|---|---|---|
| BEL | (Bell) | = zvučni signal |
| LF | (Line Feed) | = prelazak u naredni red |
| FF | (Form Feed) | = prelazak na narednu stranu |
| CR | (Carriage /Return) | = povratak na početak reda |
| ESC | (Escape) | = prelazak u komandni režim |

Skup ASCII znakova je baziran na sedmobitnim znacima, pa prema tome obuhvata ukupno 128 znakova. Kako se radi o linearno uređenom skupu svaki znak ima svoj redni broj i ti brojevi su u opsegu od 0 do 127. Funkcija koja za svaki znak daje njegov redni broj (ordinal number) označava se sa *ord*. Argument ove funkcije je tipa CHARACTER, a vrednost funkcije je tipa CARDINAL. Za slučaj ASCII skupa imamo da važi sledeće:

ord('0') = 48, ord('*A*') = 65. ord('*a*')=97.

Inverzna funkcija funkciji *ord*, koja od rednog broja znaka formira znak, (character) je funkcija *chr*:

chr(48) = '0', chr(65) = '*A*', chr(97) = '*a*' .

Ako je *c* promenljiva tipa CHARACTER, a *n* promenljiva tipa CARDINAL onda važi
chr(ord(*c*) ) = *c* , ord(chr(*n*) ) = *n* .

# 2.5. Tipovi podataka u jeziku C

Programski jezik C se svrstava u jezike sa slabim tipovima podataka iako su eksplicitne definicije tipa obavezne. Mogu se koristiti samo unapred definisani tipovi podataka uz mogućnost da im se daju pogodna korisnička imena i na taj način poveća dokumentarnost programa.

Mehanizam tipa je opšti i odnosi se i na funkcije i na promenljive. Tipovi podataka u jeziku C mogu se globalno podeliti na osnovne i složene (struktuirane). Osnovni tipovi podataka su celobrojni (int), realni (float), znakovni (char), nabrojivi (enumerated) i prazan (void). Ovi tipovi podataka se koriste u građenju složenih tipova (nizova, struktura, unija, itd.).

U sledećoj tabeli su prikazani osnovni tipovi podataka ovog jezika sa napomenom o njihovoj uobičajenoj primeni.

| Tip | Memorija u bajtovima | Opseg | Namena |
|---|---|---|---|
| char | 1 | 0 do 255 | ASCII skup znakova i mali brojevi |

| | | | |
|---|---|---|---|
| signed char | 1 | -128 do 127 | ASCII skup znakova i veoma mali brojevi |
| enum | 2 | -32.768 do 32.767 | Uređeni skup vrednosti |
| int | 4 | -2.147.483.648 do 2.147.483.647 | Mali brojevi, kontrola petlji |
| unsigned int | 4 | 0 do 4.294.967.295 | Veliki brojevi i petlje |
| short int | 2 | -32.768 do 32.767 | Mali brojevi, kontrola petlji |
| long | 4 | -2.147.483.648 do 2.147.483.647 | Veliki brojevi |
| unsigned long | 4 | 0 do 4.294.967.295 | Astronomska rastojanja |
| float | 4 | $3,4*10^{-38}$ do $3,4*10^{38}$ | Naučne aplikacije (tačnost na 6 decimala) |
| double | 8 | $1,7*10^{-308}$ do $1,7*10^{308}$ | Naučne aplikacije (tačnost na 16 decimala) |
| long double | 10 | $3,4*10^{-4932}$ do $3,4*10^{4932}$ | Naučne aplikacije (tačnost na 19 decimala) |

U C-u postoji skup operatora za rad sa binarnim sadržajima koji su prikazani u tabeli koja sledi.

| Operator | Značenje |
|---|---|
| && | Logička I operacija (AND) |
| \|\| | Logička ILI operacija (OR) |
| ^ | Isključivo ILI (XOR) |
| >> | Pomeranje udesno |
| << | Pomeranje ulevo |
| ~ | Komplement |

U slučaju mešovitih izraza u kojima se pojavljuju različiti operatori takođe važe implicitna pravila kojima je definisan prioritet operacija. Nabrojaćemo osnovna:

- Unarni operatori (na pr. ++) su uvek višeg prioriteta u odnosu na sve binarne.
- Aritmetičke operacije su višeg prioriteta u odnosu na relacije poređenja.
- Operatori poredenja <= i >= (manje ili jednako i veće ili jednako) su višeg prioriteta u odnosu na jednako i nejednako.
- Poređenja su uvek višeg prioriteta u odnosu na operatore kojima se manipuliše bitovima.
- Operatori za manipulisanje bitovima su višeg prioriteta u odnosu na sve logičke operatore.
- Logičko *I* (&&) je višeg prioriteta u odnosu na logičko *ILI* ( | | ).

## Celobrojni tipovi u C

Celobrojni tipovi su brojački tipovi i javljaju se kao označeni ili neoznačeni.

Celobrojne vrednosti obuhvataju celobrojne konstante, celobrojne promenljive, izraze i funkcije.

Celobrojne konstante predstavljaju podskup skupa celih brojeva čiji opseg zavisi od deklaracije ali i od konkretne implementacije. U C jeziku celobrojne konstante se predstavljaju kao niske cifara. Ispred koje može da stoji znak '+' za pozitivne, a obavezan je znak '-' za negativne vrednosti.

Promenljivoj osnovnog celobrojnog tipa int obično se dodeljuje memorijski prostor koji odgovara "osnovnoj" jedinici memorije. Na taj način, opseg vrednosti tipa int u TURBO C na 16-bitnim računarima je [-32768, +32767] = $[-2^{15},2^{15}-1]$, a na 32-bitnim računarima je [-2147483648, +2147483647] = $[-2^{31},2^{31}-1]$.

Bitno je napomenuti da se mogu koristiti 3 brojna sistema, i to: dekadni (baza 10), oktalni (baza 8); heksadekadni (baza 16). Heksadecimalni brojevi počinju sa 0x ili 0X. Dozvoljene cifre su 0, 1,.., 9 i slova *a, b, c, d, e, f* (ili *A, B, C, D, E, F*). Oktalni brojevi počinju sa 0 a ostale cifre mogu biti 0, 1, ..7. Na primer, 012 je dekadno 10, 076 je dekadno 62. Takođe, 0x12 je dekadni broj 18, 0x2f je dekadno 47, 0XA3 je dekadno 163. Celobrojne konstante koje ne počinju sa 0 su oktalne.

Opseg celih brojeva se može menjati primenom kvalifikatora *long* i *short*. Kvalifikator *long* može da poveća opseg vrednosti celobrojnih promenljivih tipa *int*. Opseg vrednosti tipa *long int* (ili skraćeno *long*) je [-2147483648, +2147483647] = $[-2^{31},+2^{31}-1]$.

Tip *long int* (ili *long*) garantuje da promenljive tog tipa neće zauzimati manje memorijskog prostora od promenljivih tipa *int*.

Celobrojne konstante koje su prevelike da bi bile smeštene u prostor predviđen za konstante tipa *int* tretiraju se kao *long* konstante. Ove konstante se dobijaju dodavanjem znaka L (ili l) na kraj konstante, kao na primer 1265378L. Ako program ne koristi velike cele brojeve ne preporučuje se deklarisanje promenljivih sa *long*, jer se time usporava izvršenje programa.

Kvalifikator *short int* (ili skraćeno *short*) smanjuje opseg celobrojnih promenljivih. Opseg promenljivih tipa *short* uvek je $[-2^{15},2^{15}-1]$, tj. one se uvek smeštaju u 2 bajta. Upotrebom promenljivih ovog tipa ponekad se postiže ušteda memorijskog prostora. Međutim, upotreba ovih promenljivih može da uspori izvršavanje programa, jer se pre korišćenja u aritmetičkim izrazima ove promenljive transformišu u tip *int*.

Ako smo sigurni da su vrednosti celobronih promenljivih nenegativne, one se mogu deklarisati na jedan od sledećih načina:

unsigned int (skraćeno unsigned),
unsigned short int (skraćeno unsigned short),
unsigned long int (skraćeno unsigned long).

Time se interval pozitivnih vrednosti proširuje, jer bit za registrovanje znaka gubi to značenje.

Promenljive tipa *unsigned int* (skraćeno *unsigned*) mogu uzimati samo pozitivne celobrojne vrednosti. Promenljive tipa *unsigned* imaju rang $[0, 2^{\text{širina\_reči}-1}]$. Prema tome, na 16-bitnim računarima opseg promenljivih tipa *unsigned int* je $[0,65535]=[0,2^{16}-1]$, dok je na 32-bitnim računarima odgovarajući opseg jednak $[0,2^{32}-1]=[0,+4294967295]$.

Konstante tipa *long* se mogu cpecificirati eksplicitno dodajući sufiks L ili l posle broja. Na primer, 777L je konstanta tipa *long*. Slično *U* ili *u* se može koristiti za konstante tipa unsigned. Na primer, 3U je tipa unsigned, a 3UL je tipa unsigned long.

U slučaju greške *integer overflov*, program nastavlja da radi, ali sa nekorektnim rezultatom.

Promenljive celobrojnog tipa se deklarišu navođenjem imena promenljivih iza imena nekog celobrojnog tipa. Imena promenljivih se međusobno razdvajaju zarezima, a iza spiska se navodi ';'. U operatorima opisa je dozvoljeno izvršiti inicijalizaciju deklarisanih promenljivih.

**Primer.** Navedeno je nekoliko deklaracija promenljivih celobrojnih tipova.

```
long int x;
short int y;
unsigned int z,v,w;
```

Ključna reč *int* se može izostaviti u deklaracijama, pa se može pisati

```
long x;
short y, k=10;
unsigned z;
```

## Realni tipovi podataka u C

Promenljive ovih tipova uzimaju za svoje vrednosti podskupove skupa realnih brojeva. U jeziku C postoje tri vrste podataka realnog tipa, koji se razlikuju po tačnosti predstavljanja podataka: *float* (za jednostruku tačnost), *double* (za dvostruku tačnost) i *long double*. U TURBO C vrednosti tipa float se pamte u 4 bajta (32 bita), a double u 8 bajtova (64 bita). Efekat je da vrednosti tipa float zauzimaju 6 decimalnih mesta, a double 16 decimalnih mesta. Vrednosti tipa *long double* se smeštaju u 80 bitova.

U TURBO C, pozitivne vrednosti za *float* su iz opsega $[3.4*10^{-38}, 3.4*10^{38}]$, dok su pozitivne vrednosti za *double* iz opsega $[1.7*10^{-308},1.7*10^{+308}]$. Pozitivne vrednosti tipa *long double* uzimaju vrednosti iz opsega $[3.4*10^{-4932},1.1*10^{+4932}]$.

Za vrlo velike i vrlo male brojeve može se koristiti eksponencijalni zapis, koji se sastoji iz sledećih delova:

celobrojnog dela - niz cifara,
decimalne tačke,
razlomljenog dela - niz cifara,
znaka za eksponent e ili E,
eksponenta koji je zadat celobrojnom konstantom.

Primeri realnih brojeva su: 45., 1.2345, 1.3938e-11, 292e+3.

Promenljive realnog tipa deklarišu se navođenjem liste imena promenljivih iza imena tipa.

**Primer.** Deklaracije promenljivih realnih tipova:

```
float x,y;
double z;
float p=2.71e-34;
```

## Tip char

Tip char je jedan od fundamentalnih tipova podataka u jeziku C. Konstante i promenljive ovog tipa se koriste za reprezentaciju karaktera. Znakovni tip (tip *char*) definiše uređen skup osnovnih znakova jezika C. Takav skup obrazuje skup ASCII znakova. To znači da znakovnom tipu pripadaju i znaci koji nemaju grafičku interpretaciju.

Svaki karakter se smešta u računaru u jednom bajtu memorije. Ako je bajt izgrađen od 8 bitova, on može da pamti $2^8$=256 različitih vrednosti. Promenlijve i konstante tipa char uzimaju za svoje vrednosti karaktere odnosno cele brojeve dužine jednog bajta. To znači da se znak u memoriji registruje u jednom bajtu. Promenljive ovog tipa se deklarišu pomoću ključne reči *char*.

Ako ispred deklaracije *char* stoji rezervisana reč *signed*, tada se specificira interval kodnih vrednosti [-128,127]; ako je ispred *char* navedeno unsigned, tada se specificira kodni interval [0,255].

**Primer.** Iskazom

```
char ca, cb, cc;
```

promenljive *ca*, *cb*, *cc* deklarišu se kao promenljive tipa *char*. Karakter konstanta se piše između apostrofa, kao: '*a*', '*b*', '*c*'... U Turbo C se koriste ASCII kodovi karaktera, i oni predstavljaju njihovu numeričku vrednost.

Promenljive tipa char se mogu inicijalizovati na mestu deklarisanja. Na primer, možemo pisati

```
char c='A', s='A', x;
int i=1;
```

Funkcije printf() i scanf() koriste %c za format karaktera.

**Primer.**

```
printf("%c", 'a');
printf("%c %c %c", 'A', 'B', 'C');  /* ABC */
```

Takođe, konstante i promenljive tipa char se mogu tretirati kao mali integeri.

**Primer.**

```
printf("%d", 'a');  /* 97 */
printf("%c",97);  /* a */
```

Neke znakovne konstante se moraju specificirati kao "escape" sekvence, tj. moraju se navesti zajedno sa znakom \ (backslash). Escape sekvence se koriste pri kreiranju izlaznih izveštaja u cilju specificiranja upravljačkih znakova.

'\n' prelazak na novu liniju u ispisu;
'\t' horizontalni tab (pomera kursor za 5 ili 8 pozicija);
'\b' vraća kursor za jednu poziciju (povratnik, backspace);
'\f' *form feed* (pomera hartiju štampača na početak sledeće strane);
'\a' alarm;

'\" apostrof;
'\\' backslash.

Primer.
```
printf("\"ABC\"");  /* "ABC" */
```

U stringu je single quote običan karakter:
```
printf("'ABC'");  /* 'ABC' */
```

Karakter konstante se takođe mogu prikazati pomoću jednocifrene, dvocifrene ili trocifrene sekvence. Na primer '\007', ' \07' ili '\7' predstavlja bell (alarm).

Za ulaz i izlaz karaktera se mogu koristiti funkcije *getchar*() i *putchar*(). Ovi makroi su definisani u fajlu *stdio.h*. Za učitavanje karaktera sa tastature koristi se *getchar*(), dok se *putchar*() koristi za prikazivanje karaktera na ekran.

C obezbeđuje standardni fajl *ctype.h* koji sadrži skup makroa za testiranje karaktera i skup prototipova funkcija za konverziju karaktera. Oni postaju dostupni pomoću preprocesorske direktive

#include <ctype.h>

Makroi u sledećoj tabeli testiraju karaktere, i vraćaju vrednosti true (≠0) i false (=0).

| **makro** | **vrednost ≠ 0 se vraća za** |
|---|---|
| isalpha(c) | c je slovo |
| isupper(c) | c je veliko slovo |
| islower(c) | c je malo slovo |
| isdigit(c) | c je broj |
| isxdigit(c) | c je heksadecimalan broj |
| isspace(c) | c je blanko |
| isalumn(c) | c je slovo ili broj |
| ispunkt(c) | c je interpunkcijski znak |
| isprint(c) | c je terminalni karakter |
| iscntrl(c) | c je kontrolni karakter |

Takođe, u standardnoj biblioteci <ctype.h> uključene su i funkcije toupper(c) i tolower(c) za odgovarajuću konverziju karaktera. Ove funkcije menjaju vrednost argumenta c koja je smeštena u memoriji.

toupper(c) menja c iz malog u veliko slovo;
tolower(c) menja veliko slovo u malo;
toascii(c) menja c u ASCII kod;

## Konverzija tipova podataka i kast

Mnogi jezici visokog nivoa imaju strogu tipizaciju Takvi jezici nameću tokom prevođenja slaganje tipova podataka koji su upotrebljeni u aritmetičkim operacijama, operacijama dodeljivanja i pozivima funkcija. U jeziku C se mogu pisati operacije u kojima se koriste različiti tipovi podataka (mešovite operacije). Svaki aritmetički izraz poseduje vrednost i tip. Tip rezultata aritmetičkih izraza zavisi od tipova operanda. Ako su kod binarnih aritmetičkih operatora oba operanda istog tipa, tada je tip rezultata jednak tipu operanada. Kada to nije ispunjeno, vrši se automatska konverzija operanda nižeg hijerarhijskog nivoa u tip operanda višeg nivoa. Osnovni tipovi podataka imaju sledeći hijerarhijski nivo

char<int<long<float<double.

Korišćenjem rezervisanih reči *unsigned* i *long* povišava se hijerarhijski rang, tako da je detaljniji pregled hijerarhijskih nivoa sledeći:

char< int <unsigned <long < unsigned long <float < double < long double.

Na primer, ako su oba operanda tipa *int*, svaki aritmetički izraz koji je izgrađen od njih uzima vrednost tipa *int*. Ako je jedna od promenljivih tipa *short*, ona se automatski konvertuje u

odgovarajuću vrednost tipa *int*.

Automatska konverzija tipova se javlja i pri dodeljivanju vrednosti. Kod operatora dodeljivanja vrednosti, uvek se vrednost izraza sa desne strane konvertuje u vrednost prema tipu promenljive sa leve strane operatora dodeljivanja. Ako je promenljiva d tipa double, a izraz *i* tipa *int*, tada se pri dodeljivanju *d=i* vrednost promenljive *i* konvertuje u odgovarajuću vrednost tipa *double*. Ako je *i* promenljiva tipa *int*, a *d* je neki izraz tipa double, tada u izrazu *i=d*, prevodilac vrednost izraza *d* konvertuje u odgovarajuću vrednost tipa *int*, pri čemu se gube vrednosti decimalnih pozicija.

Osim već pomenute automatske konverzije tipova podataka može se izvršiti i eksplicitna konverzija tipova. Eksplicitna konverzija se postiže operatorom kast (*cast*), oblika (tip)<izraz>. Postoje situacije kada je konverzija tipova poželjna, iako se ne izvršava automatska konverzija. Takva situacija nastaje na primer pri deljenju dva cela broja. U takvim slučajevima programer mora eksplicitno da naznači potrebne konverzije. Takve konverzije su neophodne da bi se izraz tačo izračunao.

**Primer.** Ako je promenljiva *i* tipa *int*, tada se izrazom `(double)i` njena vrednost prevodi u odgovarajuću vrednost tipa double.

Operator kast je istog prioriteta i asocijativnosti kao i ostali unarni operatori. Kast se može primenjivati na izraze.

**Primer.**
(float)i+3 je ekvivalentno sa ((float)i)+3
(double)x=77 je ekvivalentno sa ((double)x)=77

Takođe, možemo pisati
x=(float)((int)y+1)
(double)(x=77)

Posle izraza x=(int)2.3+(int)4.2 vrednost za *x* postaje 6.

## Vrste konverzije tipova

- Konverzija tipa može biti:
  - standardna – ugrađena u jezik ili
  - korisnička – definiše je programer za svoje tipove.
- Standardne konverzije su, na primer:
- konverzije iz tipa *int* u tip *float*, ili iz tipa *char* u tip *int* i slično.
- Konverzija tipa može biti:
- implicitna – prevodilac je automatski vrši, ako je dozvoljena,
- eksplicitna – zahteva programer.
- Jedan način zahtevanja eksplicitne konverzije:
- pomoću C operatora *kast* (*cast*): (tip)izraz.
- Jezik C++ uvodi 4 specifična kast operatora.
- Postoji i drugi mehanizam konverzije (konverzioni konstruktor).

## Konstante

- Konstantni tip je izvedeni tip. Dobija se iz nekog osnovnog tipa pomoću specifikatora *const*.

Konstantni tip zadržava sve osobine osnovnog tipa, samo se podatak ne može menjati.
  - Primeri: `const float pi=3.14; const char plus='+';`
- Konstanta mora da se inicijalizuje pri definisanju.
- Prevodilac često ne odvaja memorijski prostor za konstantu.
- Konstante mogu da se koriste u konstantnim izrazima koje prevodilac treba da izračuna u toku prevođenja. Na primer, konstante mogu da se koriste u izrazima koji definišu dimenzije nizova.
- Umesto simboličkih konstanti koje se uvode sa *#define* preporuka je koristiti tipizirane konstante koje se uvode sa *const*.
- Dosledno korišćenje konstanti u programu obezbeđuje podršku prevodioca u sprečavanju grešaka.

## Sizeof operator

Unarni operator `sizeof()` daje za rezultat broj bajtova potrebnih za smeštanje svog argumenta. Vrednost izraza `sizeof(obj)` se izračunava za vreme kompilovanja. Argument može biti ime promenljive, ime tipa ili izraz. Ako je *obj* ime promenljive, tada je vrednost izraza `sizeof(obj)` broj bajtova potrebnih za registrovanje te promenljive u memoriji. Ako je operand *obj* ime tipa, tada je vrednost izraza `sizeof(obj)` dužina tog tipa, odnosno broj bajtova potrebnih za registrovanje elemenata tog tipa. Ako je argument neki tip, ime tog tipa se mora navesti između zagrada. Naredba *sizeof* se koristi kada se generiše kôd koji zavisi od veličine tipa.

## Osnovne aritmetičke operacije

Osnovne aritmetčke operacije su:
+ sabiranje,
- oduzimanje,
* množenje,
/ deljenje,

Ako su oba operanda operacije deljenja / celi brojevi tada se iz realnog broja koji predstavlja njihov količnik odbacuje decimalna tačka i razlomljeni deo.

Na primer, 15/2=7, 2/4=0, -7/2=-3, 15**%**2=1, 7./4=1.74.

Operacija - može biti i unarna, i tada se koristi za promenu znaka svog argumenta (unarni minus).

Sve operacije poseduju prioritet i asocijativnost (redosled), kojima se determiniše postupak evaluacije izraza. Ako u izrazu postoje operacije različitog nivoa prioriteta, operacije se izvršavaju po opadajućem nivou prioriteta. Prioritet se može promeniti upotrebom zagrada. Asocijativnost odlučuje o redosledu izvršavanja operacija istog prioriteta.

Levoasocijativne operacije se izvršavaju s leva na desno a desnoasocijativne s desna na levo.

Najviši priorirtet ima operacija promene znaka, zatim levoasocijativni multiplikativni operatori *, / i %, dok su najnižeg nivoa prioriteta levoasocijativni aditivni operatori + i -.

Operacija inkrementiranja (uvećanja) ++ i operacija dekrementiranja (umanjenja) -- su unarne operacije sa asocijativnošću s desna na levo. U tabeli prioriteta operatori zauzimaju sledeće mesto:
najvišeg prioriteta su unarni operatori - (promena znaka); ++, --;
nižeg prioriteta su multiplikativni *, /, %; dok su najmanjeg prioriteta aditivni operatori +, -.

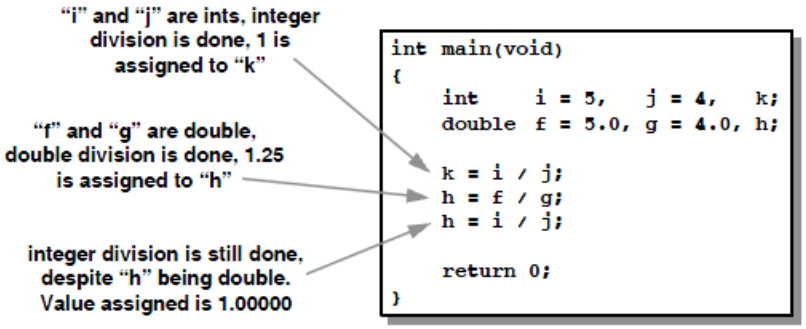
Operacije ++ i -- se mogu pisati ispred argumenta (prefiksno) ili iza argumenta (postfiksno). Mogu se primenjivati nad promenljivima a ne mogu na konstantama ili izrazima.

**Primer.** Može se pisati ++i, i++, −−cnt, cnt-- a ne može 777++, ++(a*b-1).

Svaki od izraza ++i, i++, --i, i-- ima određenu vrednost. Izrazi ++i, i++ uzrokuju inkrementiranje vrednosti promenljive i u memoriji za 1, dok izrazi --i, i-- uzrokuju dekrementiranje njene vrednosti za 1. Izrazom ++i vrednost za i se inkrementira pre nego što se upotrebi, dok se sa i++ vrednost za i inkrementira posle upotrebe te vrednosti. Slična pravila važe za izraze −−i, i−−, samo što se radi o dekrementiranju te vrednosti.

Operatori ++ i -- razlikuju se od operatora +, −, *, /, % po tome što menjaju vrednosti varijable u memoriji, tj. operatori ++ i −− poseduju bočni efekat.

**Primer.** Kompajler koristi tipove operanada kako bi odredio rezultat izračunavanja.




## Bibliotečke aritmetičke funkcije

Deo C jezika jesu standardne biblioteke koje sadrže često korišćene funkcije. Biblioteka math.h sadrži deklaraciju funkcija iz matematičke biblioteke. Sadržaj ove datoteke se uključuje u program naredbom

#include<math.h>.

Neke od funkcija iz ove biblioteke su date u nastavku.

Poznate su funkcije sin(x), cos(x), tan(x), exp(x), log(x), log10(x), sqrt(x), fabs(x)=|x|. Sledi opis najvažnijih funkcija iz ove bibilioteke.

`ceil(x)`, rezultat je najmanja celobrojna vrednost ne manja od *x* (plafon od *x*).

Rezultat funkcije `floor(x)` je najveća celobrojna vrednost ne veća od *x* (pod od *x*).

`pow(x,y)`=x^y. Ako je *x*=0 mora da bude *y*>0. Ako je *x*<0 tada *y* mora da bude ceo broj.

`asin(x)` vrednost funkcije arcsin(x), x∈[-1,1].

`acos(x)` vrednost funkcije arccos(x), x∈[-1,1].

`atan(x)` vrednost funkcije arctg(x), x∈[-π/2,π /2].

`atan2(x,y)` vrednost funkcije arctg(x/y), x∈[-π,π]. Ne može istovremeno da bude x=y=0.

`sinh(x)` vrednost funkcije sh(x).

`cosh(x)` vrednost funkcije ch(x).

`tanh(x)` vrednost funkcije th(x).

`modf(x,&y)` vrednost funkcije je razlomljeni deo realnog broja *x* sa predznakom tog broja. U argumentu y, kao bočni efekat, daje se celobrojni deo broja *x* sa predzankom tog broja. Argumenti *x* i *y* su tipa *double*.

`fmod(x,y)` vrednost funkcije je ostatak realnog deljenja *x*/*y* sa predznakom argumenta *x*. Standard ne precizira rezultat u slučaju *y*=0. Argumenti *x* i *y* su tipa *double*. Na primer, fmod(4.7,2.3)=0.1.

U biblioteci <stdlib.h> nalaze se funkcije različite namene. Navedene su neke od njih.

`abs(n)` apsolutna vrednost, gde su vrednost argumenta i funkcije tipa int,

`labs(n)` apsolutna vrednost, gde su vrednost argumenta i funkcije tipa long,

`rand()` vrednost funkcije je pseudoslučajni broj iz intervala [0,RAND_max], gde RAND_max simbolička konstanta čija vrednost zavisi od računara i nije manja od 32767,

`srand(n)` postavlja početnu vrednost sekvence pseudoslučajnih brojeva koju generiše `rand()`. Podrazumevana početna vrednost je 1. Tip argumenta je *unsigned int*. Funkcija ne daje rezultat.

`atof(s)` ova funkcija vrši konverziju realnog broja iz niza ASCII cifara (karaktera) oblika "±cc....cc... E±ee" u binarni ekvivalent. Argument *s* je string a rezultat je tipa *double*. Pre konverzije se brišu početne praznine. Konverzija se završava kod prvog znaka koji ne može da bude deo broja.

`atoi(s)` ova funkcija vrši konverziju celog broja iz niza ASCII cifara (karaktera) oblika "±cc..." u binarni ekvivalent. Argument *s* je string a rezultat je tipa *int*. Početne praznine se ignorišu. Konverzija se završava kod prvog znaka koji ne može da bude deo broja.

`atol(s)` ova funkcija vrši konverziju celog broja iz niza ASCII cifara (karaktera) oblika "±cc..." u binarni ekvivalent tipa *long*. Argument *s* je string a rezultat je tipa *long*. Početne praznine se ignorišu. Konverzija se završava kod prvog znaka koji ne može da bude deo broja.

## Operacija dodeljivanja u C

U C jeziku se operator = tretira kao operator dodeljivanja. Njegov prioritet je manji od prioriteta do sada razmatranih operacija, a njegova asocijativnost je ``s desna na levo''. Izraz dodeljivanja vrednosti je oblika

<promenljiva> = <izraz>;

na čijoj je desnoj strani proizvoljan izraz. Kompletan izraz je završen sa ; (semicolon). Vrednost izraza na desnoj strani se dodeljuje promenljivoj sa leve strane. Ako su tipovi promenljive i izraza različiti, vrši se konverzija vrednosti izraza u odgovarajuću vrednost saglasno tipu promenljive. Vrednost izraza dodeljivanja jednaka je vrednosti izraza sa njegove desne strane.

**Primer.** Sledeća sekvenca izraza

```
y=2; z=3;   x=y+z;
```

može se efikasnije zapisati u obliku

```
x=(y=2)+(z=3);
```

**Primer.** Zbog desne asocijativnosti operatora = je izraz x=y=z=0 ekvivalentan izrazu

```
x=(y=(z=0)).
```

**Primer.** Operator y=x++; je ekvivalentan sledećoj sekvenci operatora: y=x; x=x+1;
Operator y=--x je ekvivalentan sledećim operatorima x=x−1; y=x;

Posle izvršavanja operatora

```
x=y=1; z=(x+(++y))*3;
```

dobijaju se sledeće vrednosti promenljivih:

```
x=1, y=2, z=(1+2)*3=9.
```

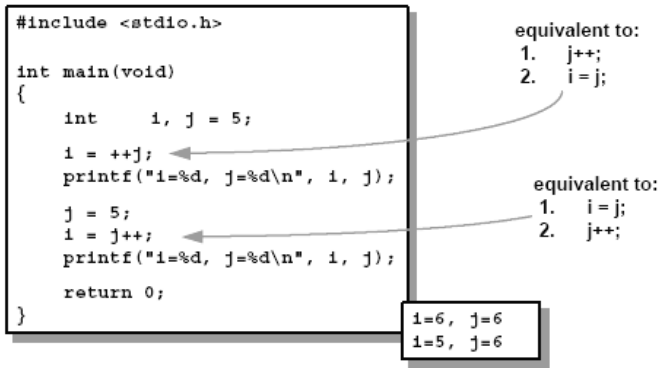
Operatori

```
x=y=1; z=(x+(y++))*3;
x=y=1; t=(x+y++)*3;
```

daju sledeće vrednosti promenljivih:

```
x=1, y=2, z=(1+1)*3=6, t=(1+1)*3=6.
```

**Primer.**




Operatori složenog dodeljivanja su:

=, +=, -=, *=, /=, %=, ||=, &=,^=, |=

Svi ovi operatori imaju isti prioritet i asocijativnost s desna na levo. Ako su *l* i *r* proizvoljni izrazi, tada je izraz `l<op>=r` jednak `l=l<op> r`.

Na primer,

x+=2; je ekvivalentno sa x=x+2;
x%=2; je ekvivalentno sa x=x%2;

**Primer.** Šta se ispisuje posle izvršenja sledećeg programa?

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

```
void main()
{ int x,y,z;
  x=y=1;
  z=(x+y++)*(y==1);
  printf("%d, %d, %d\n",x,y,z);
  int i=1,j;
  i=i++*++i;
  printf("%d\n",i);
  i=j=2;
  printf("%d\n",++j+i++);
  printf("%d, %d\n",i,j);
}
```

**Primer.** Šta se ispisuje posle izvršenja sledećeg programa?

```
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
void main()
{ int i,j=6;
  if(j=0) i=5;
  else i=1;
  printf("%d\n%d\n",i,j);
}
```

### Operacije poređenja i logičke operacije

U jeziku C postoje samo numerički tipovi. Ne postoji čak ni logički tip podataka. Za predstavljanje logičkih vrednosti koriste se celobrojni podaci tipa int. Vrednost 0 se tumači kao logička neistina (false), a bilo koja vrednost različita od nule tumači se kao logička istina (true).

## Operacije poređenja

Operacije poređenja su:
<, <=, >, >=, ==, !=.
Rezultat izvršenja ovih operacija je 1 ako je ispunjeno poređenje, a inače je 0. Ove operacije se izvršavaju s leva na desno. Operacije poređenja su nižeg prioriteta od aritmetičkih operacija.
Unutar operacija poređenja, operacije <, <=, >, >= su višeg prioriteta od operacija == i !=.

Na primer, x>y+3 je ekvivalentno sa x>(y+3).

### Logičke operacije

Postoje tri logičke operacije:

- `!` je operacija negacije,
- `&&` je konjunkcija, i
- `||` predstavlja opreaciju disjunkcije.

Rezultat primene ovih operacija je 0 ili 1.

Operacija negacije ! je unarna, i daje rezultat 1 ako je vrednost operanda 0, a vrednost 0 ako je vrednost operanda 1.

Operacija konjunkcije && je binarna, i daje 1 ako su oba operanda različita od 0, a 0 u suprotnom. Ako je levi operand jednak 0 pri izvršenju operacije konjunkcije desni operand se ignoriše.

Operacija disjunkcije || je binarna, i daje 1 ako je bar jedan operand različit od 0, a 0 inače. Ako je vrednost levog operanda jednaka 1 pri izvršenju operacije disjunkcije desni operand se ignoriše.

Najviši prioritet ima operacija negacije; sledećeg nivoa prioriteta je operator konjunkcije, dok je najnižeg nivoa prioriteta operator disjunkcije.

Prema tome,

`p||q && r` je isto sa `p||(q && r)`,
`x<=y && r` je isto sa `(x<=y)&& r`.

**Primer.** Napisati operatore dodeljivanja kojima se realizuje sledeće:

a) Promenljivoj *p* se dodeljuje vrednost 1 ako se od odsečaka *x, y, z* može konstruisati trougao, a inače 0.

```
p=((x+y>z) && (x+z>y) && (y+z>x));
```

b) Promenljivoj p se dodeljuje vrednost 1 ako se pravougaonik sa stranicama *a* i *b* može ceo smestiti u pravougaonik sa stranicama *c* i *d*, a inače 0.

```
p=((a<c)&& (b<d) || (a<d)&& (b<c));
```

# 2.6. Diskretni tipovi podataka u programskim jezicima

Grupa diskretnih tipova obuhvata sve tipove podataka sa konačnim, diskretnim skupom vrednosti. U takvom skupu vrednosti je definisana relacija poretka. Diskretnost i uređenost skupa vrednosti omogućava definisanje intervala i upotrebu nekih opštih zajedničkih funkcija i atributa u ovoj grupi tipova. U diskretne tipove podataka svrstavaju se svi celobrojni tipovi podataka, kao i svi drugi elementarni tipovi sa diskretnim skupom vrednosti kao što su znakovni i logički tipovi. Posebno značajni u grupi diskretnih tipova su tipovi nabrajanja (enumerated types) i intervalni tipovi nekih jezika.

## 2.6.1. Nabrojivi tipovi u programskim jezicima

U programskom jeziku C se u okviru definicija tipova nabrajanja eksplicitno koristi reč *enum*. Gore datim tipovima podataka u ovom jeziku odgovaraju sledeće definicije:

```
enum BOJE {crvena,bela,zelena,plava};
enum DANI  {ponedeljak,utorak,sreda,cetvrtak,petak,subota,nedelja};
enum MESECI   {jan,feb,mar,apr,maj,jun,jul,avg,sep,okt,nov,dec};
enum STATUS  {ON,OFF};
enum PRAVCI  {sever,jug,istok,zapad};
```

# 2.8. Upotreba typedef iskaza u C

Ključna reč *typedef* dozvoljava dodeljivanje alternativnih imena postojećim tipovima podataka, kao i definisanje novih tipova podataka.

**Primer.** Iskazom

```
typedef int celi;
```

definiše se ime *celi* kao ekvivalentno ključnoj reči *int*. Sada se moše pisati

```
celi i, j, n;
```

Izraz

```
typedef double vektor[20];
```

definiše tip *vektor* kao niz od 20 elemenata tipa double.

Analogno, tip *matrica* kao dvodimenzionalni niz elemenata tipa double može se definisati na sledeći način:

```
typedef double matrica[10][10];
```

Nakon ovih definicija tipova se imena vektor i matrica mogu koristiti kao tipovi podataka.

# 3. ULAZ I IZLAZ PODATAKA

U programskim jezicima postoji veći broj funkcija za unošenje i izdavanje podataka. Pomoću naredbi ulaza i izlaza program dobija neophodne podatke i prikazuje dobijene rezultate. Ove funkcije se koriste za standardni ulaz i izlaz podataka preko tastature i monitora. Postoje varijante ovih funkcija koje se mogu koristiti za pristup datotekama.

## 3.1. Ulaz i izlaz podataka u jeziku C

### 3.1.1. Funkcije printf() i scanf()

Funkcija *printf*() se koristi za formatizovani izlaz podataka, a *scanf*() za formatizovano učitavanje podataka. Ove funkcije se nalaze u standarnoj biblioteci *stdio.h*. Datoteka *stdio.h* sadrži podake neophodne za pravilno funkcionisanje ulazno-izlaznih funkcija. Njen sadržaj se uključuje u program naredbom

#include<stdio.h>.

Funkcija *printf*() se poziva izrazom oblika

printf(Upravljacki_string [, Arg1,Arg2,...]);

Lista argumenata može da se izostavi. Tada je jedini argument funkcije *printf*() upravljački string koji se ispisuje na ekran. U opštem slučaju, funkcija *printf*() se primenjuje na listu argumenata koja se sastoji iz dva dela. Prvi argument funkcije *printf*() je kontrolni ili konverzioni string (upravljački string), a drugi je lista izlaznih podataka čije se vrednosti ispisuju. Konverzioni string određuje format za štampanje liste izlaznih podataka. Najprostija specifikacija formata počinje karakterom '%', a završava se konverzionim karakterom (formatom ili konverzionom specifikacijom).

U sledećoj tabeli su prikazani konverzioni karakteri

| **konverzioni karakter** | **konverzija izlaznog niza** |
|---|---|
| `c` | karakter char |
| `d` | ceo broj int |
| `u` | ceo dekadni broj bez znaka |
| `o` | oktalni broj int |
| `x, X` | heksadekadni broj int |
| `ld` | dug ceo broj long int |
| `lo` | dug oktalni broj long int |
| `lx` | dug heksadekadni broj long int |
| `f` | fiksni zarez za float |
| `lf` | double |
| `e, E` | pokretni zarez za float i double |
| `g, G` | kraći zapis za f ili e |
| `s` | string |
| `p` | vrednost pointera |

Konverzija *g* i *G* su jednake konverzijama *e* i *E* ukoliko je eksponent prikazanog broja manji od -4 ili je veći ili jednak preciznosti obuhvaćene specifikacijom *d*.

Opšti oblik konverzione specifikacije je:

%[+|-|_][sirina][.tacnost]konverzioni_karakter

gde:

- *konverzioni_karakter* definiše konverziju.

- Znak - ukazuje da se argument poravnava na levoj strani zadate širine polja.

- Znakom + se kod numeričkih konverzija označava da ispred pozitivnih vrednosti treba da se napiše predznak +.

- Znak razmaka '_' označava da je predznak praznina umesto +.

Bez ovih parametara konverzije, pozitivne vrednosti se prikazuju bez predznaka.

- Parametar *sirina* zadaje **minimalnu** širinu polja. Polje se proširuje za potreban broj pozicija ako izlazna vrednost zahteva više pozicija nego što je specificirano parametrom sirina. Ako specifikacija širine polja počinje nulom, tada se polje umesto prazninama popunjava nulama u neiskorišćenim pozicijama ako je poravnanje po desnoj margini.

- Parametar *tacnost* se koristi za realne brojeve (*float* ili *double*) i određuje broj decimala iza decimalne tačke.

**Primer.** (a) Posle izraza

```
printf("Danas je sreda \b\b\b\b\b petak \n ");
```

na ekranu se ispisuje tekst

```
Danas je petak.
```

(b) `printf("Vrednost za x = %f\n", x);`

(c) `printf("n = %d x = %f %f^ %d=%lf\n", n,x,x,n,pow(x,n));`

Funkcija *scanf*() ima za prvi argument kontrolni string koji sadrži formate koji odgovaraju ulaznim veličinama. Drugi argument jeste lista adresa ulaznih veličina.

scanf(Upravljacki_string,Adresa1[,Adresa2,...]);

Upravljački string sadrži konverzione specifikacije koje odgovaraju ulaznim veličinama. Lista adresa sadrži adrese promenljivih u koje se smeštaju učitane i konvertovane vrednosti. Operator adrese se predstavlja simbolom &. Adresa promenljive se formira tako što se ispred imena promenljive piše operator adresiranja &.

Na primer, u izrazu `scanf("%d", &x);` format `%d` uzrokuje da se ulazni karakteri interpretiraju kao ceo broj i da se učitana vrednost smešta u adresu promenljive *x*.

Neka su deklarisane celobrojne promenljive *i*, *j*, *k*. Tada se tri celobrojne vrednosti mogu dodeliti ovim promenljivim preko tastature pomoću operatora

```
scanf("%d%d%d", &i,&j,&k);
```

Za razdvajanje ulaznih polja u funkciji *scanf*() koriste se sledeći znakovi: praznina (blank), tabulator (*tab*) i prelaz u novi red (*enter*) (tzv. beli znaci). Izuzetak se čini u slučaju ako neki od ulaznih znakova odgovara konverzionom karakteru %*c*, kojim se učitava sledeći znak čak i kada je on nevidljiv.

Pri ulaznoj konverziji opšte pravilo je da podatak čini niz znakova između dva bela znaka. Odavde sledi da jednim pozivom funkcije *scanf* mogu da se učitavaju podaci iz više redova. Važi i obrnuto, veći broj poziva funkcije scanf može da u čitava uzastopne podatke iz istog reda.

Znaci koji se pojavljuju u upravljačkom stringu a nisu konverzioni moraju se pri učitavanju pojaviti na određenom mestu i ne učitavaju se. Na primer, ako se unese ulazni red oblika

1:23:456

tada se operatorom

scanf("%d:%d:%d", &i,&j,&k);

celobrojnim promenljivim *i*, *j* i *k* dodeljuju vrednosti 1, 23 i 456, redom.

Opšti oblik konverzione specifikacije je:

%[*][sirina]konverzioni_karakter

gde je:

- znak * označava da se obeleženo ulazno polje preskače i ne dodeljuje odgovarajućoj promenljivoj.

- *sirina* zadaje **maksimalnu** širinu polja. Ako je širina polja veća od navedene u konverzionoj specifikaciji, koristi se onoliko znakova koliko je zadato maksimalnom širinom.

- *konverzioni_karakter* je jedan od konverzionih znakova iz sledeće tabele:

| **konverzioni karakter** | **tip argumenta** |
|---|---|
| `c` | pokazivač na char |
| `h` | pokazivač na short |
| `d` | pokazivač na int |
| `ld, D` | pokazivač na long |
| `o` | pokazivač na int |
| `lo, O` | pokazivač na long |
| `x` | pokazivač na int |
| `lx, X` | pokazivač na long |
| `f` | pokazivač na float |
| `lf, F` | pokazivač na double |
| `e` | pokazivač na float |
| `le, E` | pokazivač na double |
| `s` | pokazivač na string |

**Primer.** (i) Neka je *i* celobrojna i *x* realna promenljiva.
Ako je ulazna linija oblika

57 769 98.

tada se operatorom

```
scanf("%2d%*d%f",&i,&x);
```

promenljivim *i* i *x* dodeljuju redom vrednosti 57 i 98.0.

(ii) Ako su *i, j, k* celobrojne promenljive i ako se unese ulazna linija 123 456 789, posle primene operatora

```
scanf("%2d%3d%2d",&i,&j,&k);
```

promenljive *i, j, k* uzimaju vrednosti 12, 3, 45.

**Primer.** Jednostavan C program koji demonstrira komentare i pokazuje nekoliko promenljivih i njihove deklaracije.

```
#include <stdio.h>
void main()
{  int i, j;     /* Ove 3 linije deklarišu 4 promenljive */
   char c;
   float x;
   i = 4;        /* i, j imaju pridružene celobrojne vrednosti */
   j = i + 7;
   c = 'A';      /* Karakteri su okruženi apostrofima */
   x = 9.087;    /* x zahteva neku realnu vrednost */
   x = x * 4.5; /* Promena vrednosti u x */
  /* Prikaz vrednosti promenljivih na ekran */
   printf("%d %d %c %f", i, j, c, x);
}
```

**Primer.** Dekadni, oktalmi i heksadekadni sistem.

```
#include <stdio.h>
int main(void)
{ int dec = 20, oct = 020, hex = 0x20;
  printf("dec=%d, oct=%d, hex=%d\n", dec, oct, hex);
  printf("dec=%d, oct=%o, hex=%x\n", dec, oct, hex);
  return 0;
}
```

**Primer.** Realni brojevi u naredbi *printf*.

```
#include <stdio.h>
#include <float.h>
int main(void)
{ double f = 3.1416, g = 1.2e-5, h = 5000000000.0;
  printf("f=%lf\tg=%lf\th=%lf\n", f, g, h);
  printf("f=%le\tg=%le\th=%le\n", f, g, h);
  printf("f=%lg\tg=%lg\th=%lg\n", f, g, h);
  printf("f=%7.2lf\tg=%.2le\th=%.4lg\n", f, g, h);
  return 0;
}
```

```
f=3.141600 g=0.000012 h=5000000000.000000
f=3.141600e+00 g=1.200000e-05 h=5.000000e+09
f=3.1416 g=1.2e-05 h=5e+09
f= 3.14 g=1.20e-05 h=5e+09
```

## 3.1.2. Direktive pretprocesora u C

Simbolička konstanta je konstanta kojoj je pridružen identifikator. U programu se kasnije koristi identifikator, a ne vrednost konstante. Simboličke konstante u jeziku C se definišu korišćenjem direktiva pretprocesora. Ovakva definicija počinje pretprocesorskom direktivom *#define* (sve pretprocesorske direktive počinju znakom #). Iza direktive pretprocesora se navodi simboličko ime konstante a zatim konstantna vrednost koja predstavlja vrednost te konstante. Vrednost konstante se može zadati na sledeći način:
- pomoću imena već definisane konstante;
- konstantom (celobrojnom, realnom, znakovnom, logičkom);
- konstantnim izrazom.

Iza ove direktive ne koristi se znak ';' jer pretprocesorske direktive ne pripadaju jeziku C. Pretprocesorske direktive se moraju navesti pre prvog korišćenja imena konstante u programu. U praksi se direktive najčešće pišu na početku programa.

Direktive *#define* mora da se piše u posebnom redu. Jednom naredbom može da se definiše samo jedna konstanta. Zajedno sa nekom ovakvom direktivom ne smeju da budu druge naredbe. Jedino je dozvoljen komentar na kraju reda.

Identifikatori simboličkih konstanti pišu se velikim slovima da bi se razlikovali od identifikatora promenljivih koji se obično pišu malim slovima. Ovo pravilo nije nametnuto sintaksom jezika C već stilom koji se ustalio u praksi.

**Primer.** Definisanje simboličkih konstanti.

```
#define PI 3.141592
#define N 100
#define NMIN -N
#define PIX2 (PI*2)
#define LIMT 5
#define BELL '\007'
#define PORUKA "Zdravo"
```

Mogućnost da se konstanta definiše pomoću pretprocesorskih direktiva omogućuje da se vrednost konstante jednostavno zapiše i promeni na jednom mestu. Korišćenje konstanti u programu navođenjem njihovih imena umesto odgovarajuće vrednosti povećava čitljivost programa.

Takođe, može da se uvede promenljiva kojoj se dodeljuje vrednost konstante.

**Primer.** Umesto izraza

```
#define POREZ 15
```

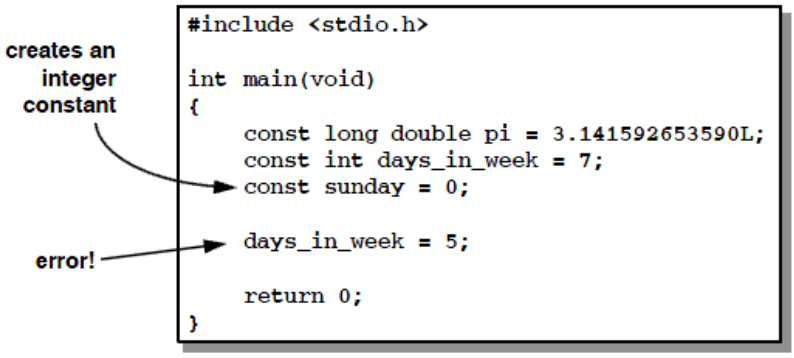
može se pisati

```
float POREZ=15
```

Metod koji koristi promenljive je neefikasan, jer se pri svakom korišćenju vrednosti promenljive vrši obraćanje memoriji za izračunavanje njene vrednosti, čime se usporava izvršenje programa. Ovde

se radi o smeni u "vreme izvršavanja", za razliku od pretprocesorske direktive kojom se vrši smena "u toku kompilacije".

Imenovane konstante se mogu kreirati koristeći naredbu *const*.




## Primeri u jeziku C

**Primer.** Zbir dva broja u jeziku C.

```
#include <stdio.h>
main()
 { int x, y;
    printf("Unesi dva cela broja");
    scanf("%d %d", &x, &y);
    printf("Njihov zbir je %5d\n",x+y);
 }
```

**Primer.** Proizvod dva broja tipa long.

```
#include <stdio.h>
main()
 { long x=2345678, y=67890;
    printf("Njihov proizvod je %ld\n",x*y);
 }
```

**Primer.** Zadavanje i prikazivanje karaktera.

```
#include <stdio.h>
main()
 { char c='A';
   printf("%c%d\n",c,c);
 }
```

**Primer.** Rad sa određenim brojem decimala.

```
#include <stdio.h>
void main()
{float x=23.45678888, y=7.8976789;
 printf("Njihov proizvod je:%f,%0.16f,%0.20f\n", x*y,x*y,x*y);
}
```

**Primeri.** Napredno korišćenje #define.

**Primer .**

```
#include <stdio.h>
#define veci(X,Y) X>Y
#define kvad(X) X*X
#define kvadrat(X) ((X)*(X))
void main()
{printf("To su %d %d %d %d\n", veci(3,2),kvad(5),kvad(5+5),kvadrat(5+5));}
```

**Primer.**

```
#include <stdio.h>
#define MAX   23
#define Ako   if
#define Onda
#define Inace else
void main()
 { int n=24;
    Ako (MAX > n) Onda printf("veci\n");
    Inace printf("manji\n");
 }
```

# 4. OSNOVNE UPRAVLJAČKE STRUKTURE

Svaki program sadrži naredbe dodeljivanja kojima se izračunavaju vrednosti određenih izraza i pamte (dodeljuju) kao vrednosti odgovarajućih promenljivih. Ipak, veoma retko se bilo koji program sastoji samo od jedne sekvence naredbi dodeljivanja kao i naredbi ulaza/izlaza. Mnogo se češće zahteva postojanje više različitih tokova u zavisnosti od određenih uslova ili mogućnost višestrukog izvršavanja dela programa. Naredbe koje omogućavaju definisanje toka programa nazivaju se upravlajčke naredbe.

Upravlajčke naredbe u prvom imperativnom programskom jeziku FORTRAN bile su izvedene iz osnovnih mašinskih naredbi. To je period kada metodologija programiranja tek počinje da se razvija tako da još nije definisan neki osnovni skup upravljačkih naredbi. Početkom sedamdesetih Wirt definiše metodologiju strukturnog programiranja i programski jezik Pascal sa skupom upravljačkih struktura kojima se implementiraju osnovne algoritamske strukture. Ovaj koncept je široko prihvaćen, tako da danas viši programski jezici iz grupe proceduralnih jezika obično raspolažu skupom upravljačkih struktura kojima se upravlja tokom izvršenja programa u skladu sa konceptima strukturnog programiranja. Iako se od jezika do jezika ovaj skup upravljačkih struktura donekle razlikuje, može se reći da su ipak među njima u ovom domenu male suštinske razlike. Smatra se da je skup naredbi kojima se upravlja tokom programa dovoljan ako obezbeđuje sledeće tri osnovne upravlajčke strukture:

(1) Strukturu selekcije, koja omogućava izbor jedne od dve mogućnosti. U Pascal-u se obično implementira izrazom oblika:

**If** B **then** St **else** Sf ;

Struktura selekcije određena je *if-then* i *if-then-else* izrazima.

(2) Strukturu višestruke selekcije, koja omogućava izbor jedne između više ponuđenih grana. U Pascal-u se koristi *case* izraz oblika:

```
case X of
    xa  :  Sa;
    xb  :  Sb;
end;
```

(3) Strukturu iteracije, koja omogućava višestruko izvršavanje nekog bloka naredbi. U jeziku Pascal se koristi *while do* petlja oblika:

**while** B **do** S**;**

Upravljačke strukture jednog proceduralnog jezika bi trebalo da ispunjavaju sledeće osobine:

- smisao naredbi mora da bude jasan i jednoznačan;

- sintaksa naredbe treba da bude tako postavljena da dozvoljava hijerarhijsko ugrađivanje drugih naredbi, što mora da bude jednoznačno definisano i jasno iz samog teksta programa;
- potrebno je da postoji mogućnost lake modifikacije naredbi.

# 4.1. načini predstavljanja algoritama

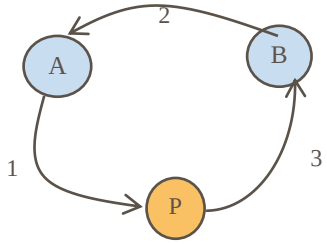
- Tekstualni
- Grafički (pomoću blok šema)
- Pseudo kodom
- Strukturogramom

***Tekstualni način***

- Koriste se precizne rečenice govornog jezika
  - Koristi se za lica koja se prvi put sreću sa pojmom algoritma
  - Dobra osobina: razumljivost za širi krug ljudi
  - Loše: nepreciznost koja proističe iz nepreciznosti samog jezika

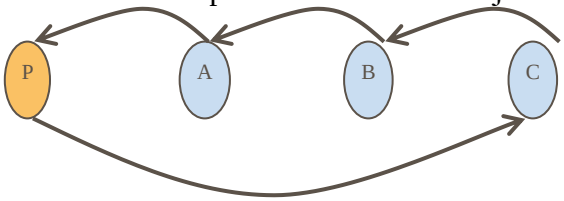
**Primer 1.** Zameniti sadržaje dve memorijske lokacije, A i B

- Rešenje – potrebna je treća, pomoćna, lokacija
  1. sadržaj lokacije A zapamtiti u pomoćnu lokaciju, P
  2. sadržaj lokacije B zapamtiti u lokaciju A
  3. sadržaj lokacije P zapamtiti u lokaciju B
  4. kraj




**Primer 2**

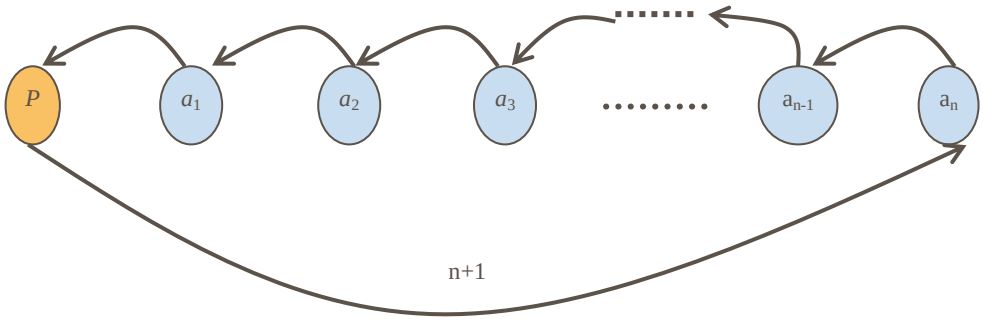
- Ciklički pomeriti u levo sadržaje lokacija A, B i C



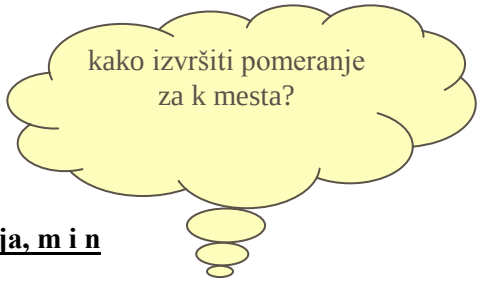

- iz A u P
- iz B u A
- iz C u B
- iz P u C
- kraj

**Primer 3.** Ciklički pomeriti u levo sadržaje lokacija $a_1$, $a_2$, ..., $a_n$.

1 2 3 $n$




- iz $a_1$ u P
- iz $a_i$ u $a_{i-1}$, za $i$=2,...,$n$
- iz P u $a_n$
- kraj

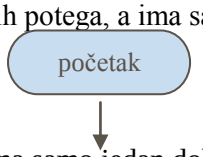



**Euklidov algoritam za nalaženje NZD dva prirodna broja, m i n**

1. Podeliti *m* sa *n* i ostatak zapamtiti u *r*;
2. Ako je *r* jednako 0, NZD je *n* i kraj, inače preći na korak 3;
3. Zameniti *m* sa *n*, *n* sa *r*, i preći na korak 1;

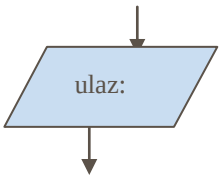
***Grafičko predstavljanje algoritma***

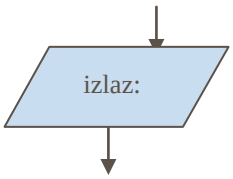
- Koriste se odredjeni grafički simboli za predstavljanje pojedinih aktivnosti u algoritmu
  - Ideja je pozajmljena iz teorije grafova
  - Algoritam se predstavlja usmerenim grafom
    - čvorovi grafa predstavljaju aktivnosti koje se obavljaju u algoritmu
    - potezi ukazuju na sledeću aktivnost koja treba da se obavi
- Polazni čvor u usmerenom grafu koji predstavlja algoritam nema dolaznih potega, a ima samo jedan izlazni poteg (granu)



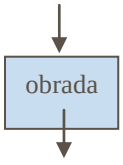

- Krajnji čvor u grafu koji predstavlja algoritam nema izlaznih potega, a ima samo jedan dolazni poteg


- Blok oblika romboida koristi se za označavanje ulaznih i izlaznih aktivnosti

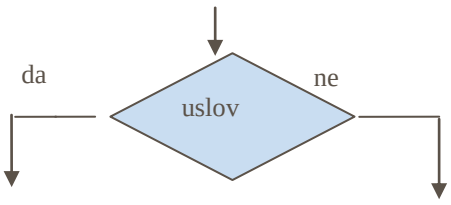


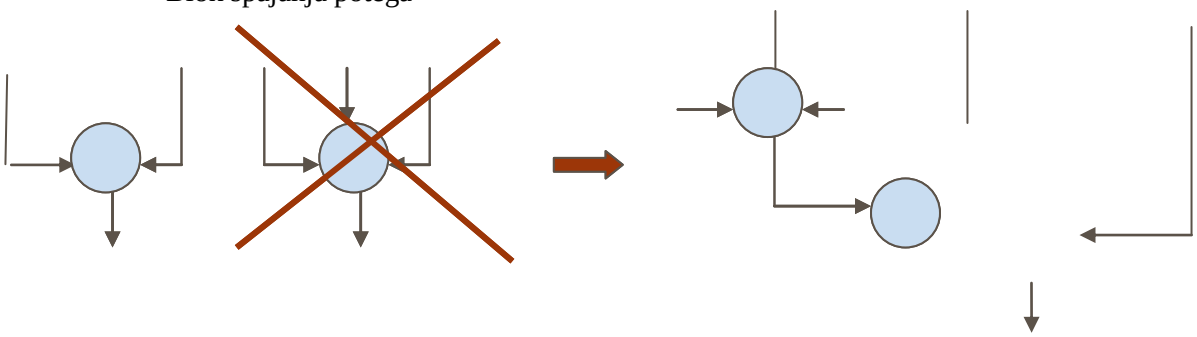

- Blok obrade je pravougaonog oblika




- Blok odluke



- Blok spajanja potega

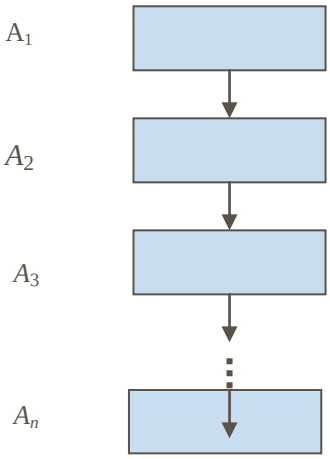


## Osnovne algoritamske strukture

- Kombinovanjem gradivnih blokova dobijaju se osnovne algoritamske strukture
  - sekvenca
  - alternacija (selekcija)
  - petlja (ciklus)
- Pomoću osnovnih algoritamskih struktura može se predstaviti svaki algoritam

### *Sekvenca*

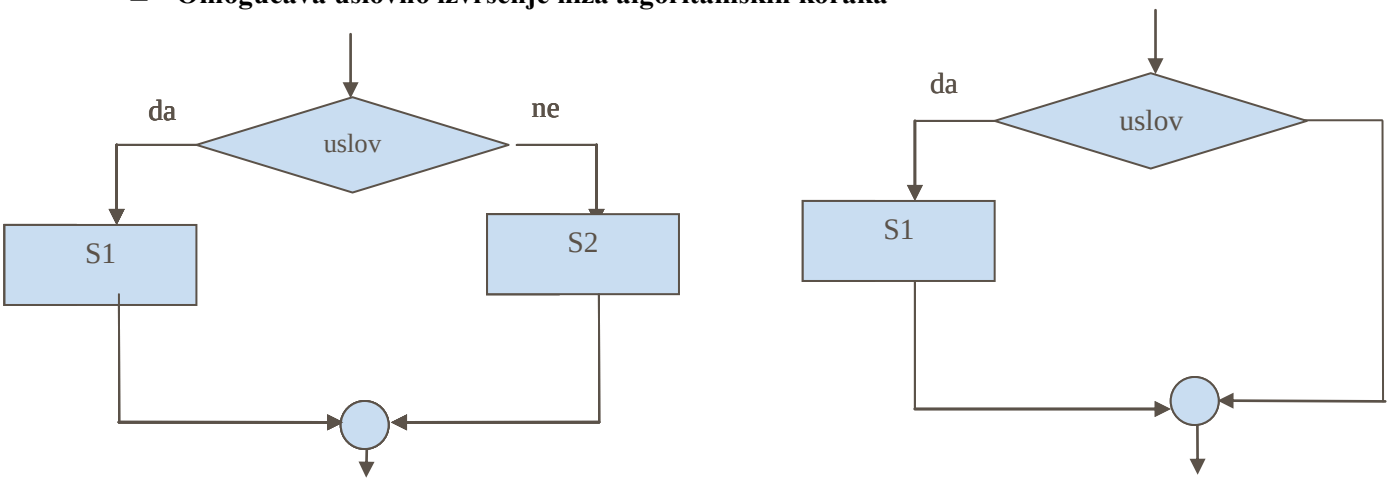
- linijska struktura koja se dobija kaskadnim povezivanjem blokova obrade




- Algoritamski koraci se izvršavaju redom, jedan za drugim
- Algoritamski korak $A_i$, i=2,...,n ne može da otpočne sa izvršenjem dok se korak $A_{i-1}$ ne završi
- sekvenca predstavlja niz naredbi dodeljivanja (:=)
- oblik naredbe:
  *promenljiva*:=*vrednost*
  *a:=b*
  *n:=n+1*

### *Alternacija (selekcija)*

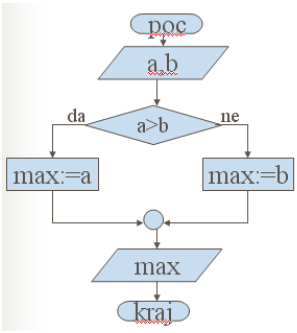
- **Omogućava uslovno izvršenje niza algoritamskih koraka**

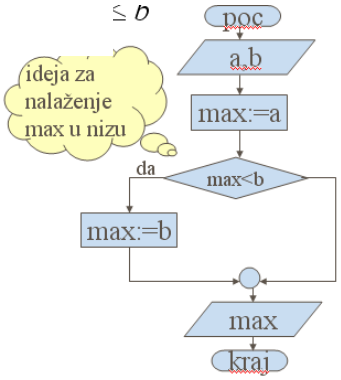



- Blokovi označeni sa S1 i S2 mogu sadržati bilo koju kombinaciju osnovnih algoritamskih struktura.

**Primer 1.** Nacrtati dijagram toka algoritama kojim se odredjuje veći od dva zadata broja korišćenjem formule

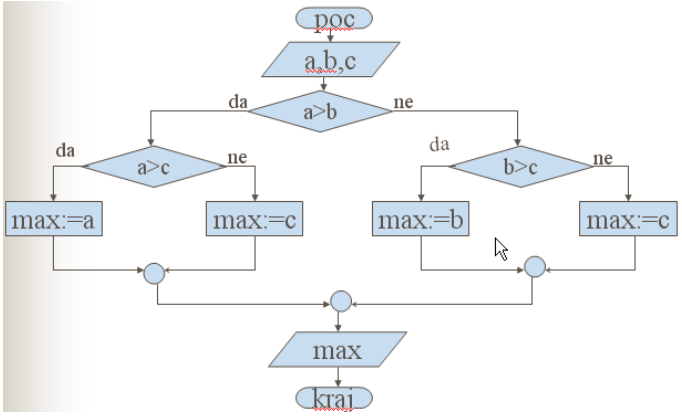
$$\max\{a,b\} = \begin{cases} a, & a > b \\ b, & a \le b \end{cases}$$



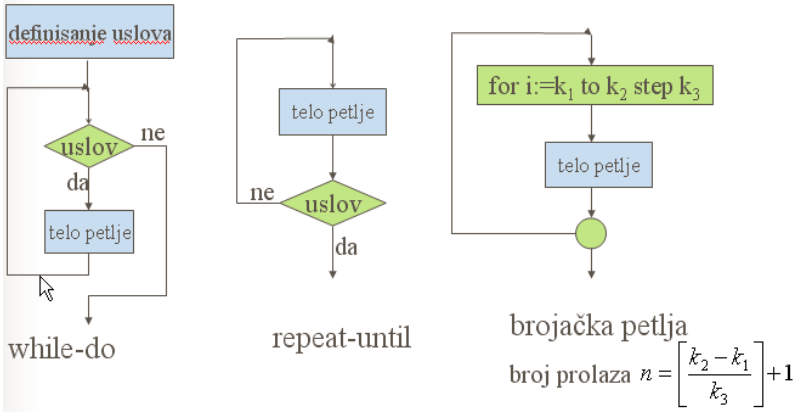

Naći maksimum od tri zadata broja a, b i c

$$\max\{a,b,c\} = \max\{\max\{a,b\},c\}$$



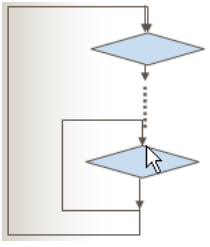

***Petlja (ciklus)***

- Omogćava da se algoritamski koraci ponavljaju više puta.



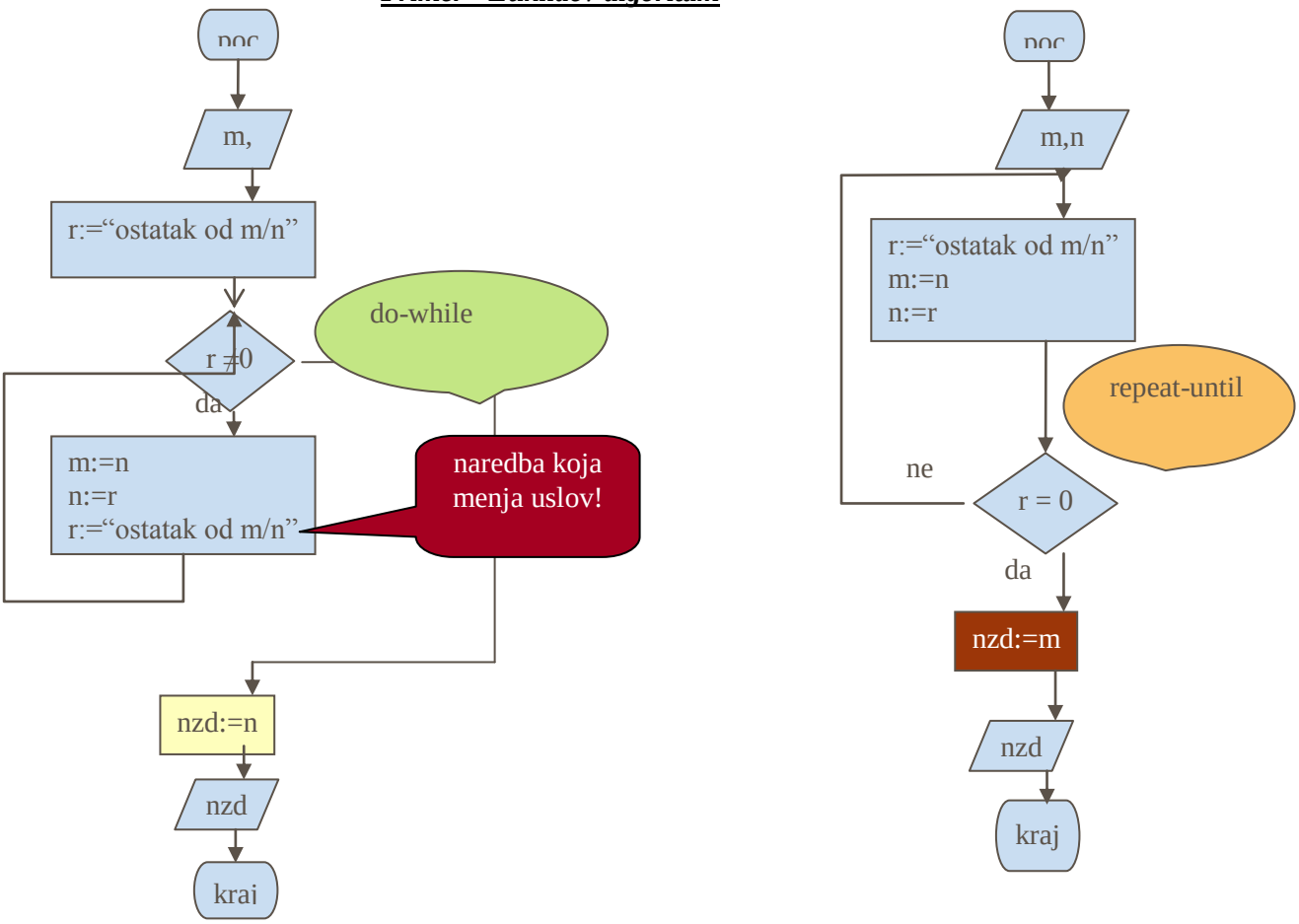

***Pravila***

- Ako petlja počne unutar *then* bloka ili *else* bloka, u tom bloku se mora i završiti!
- Dozvoljene su paralelne (ugnježdjene) petlje.
- su dozvoljene petlje koje se seku!



Paralelne petlje

***Primer - Euklidov algoritam***




**Predstavljanje algoritma pomoću pseudo koda**

- Koristi se tekstualni način dopunjen formalizmom
  - svaka od osnovnih algorotamskih struktura se predstavlja na tačno definisani način:
    - sekvenca se predstavlja kao niz naredbi dodeljivanja odvojenih simbolom ;

```
a:=5;
b:=a*b;
c:=b-a;
```

- ***Alternacija***

```
if (uslov) then
   niz_naredbi
else
   niz_naredbi
endif;
```

ili

```
if (uslov) then
   niz_naredbi
endif;
```

```
if (a>b) then
   max:=a
else
  max:=b
endif;
```

ili

```
max:=a;
if (max<a) then
max:=b
endif;
```

***Alternacija, primer2***

- max(a,b,c)

```
if (a>b) then
    if (a>c) then
        max:=a
    else
        max:=c
    endif;
else
    if (b>c) then
        max:=b
    else
        max:=c
    endif;
endif;
```

***Petlje – pseudo kod***

```
while (uslov) do
        niz_ naredbi
enddo;
```

**Primer:**

```
r:="ostatak od m/n"
while (r≠0) do
        m:=n;
        n:=r;
        r:="ostatak od m/n";
enddo;
nzd:=n;
```

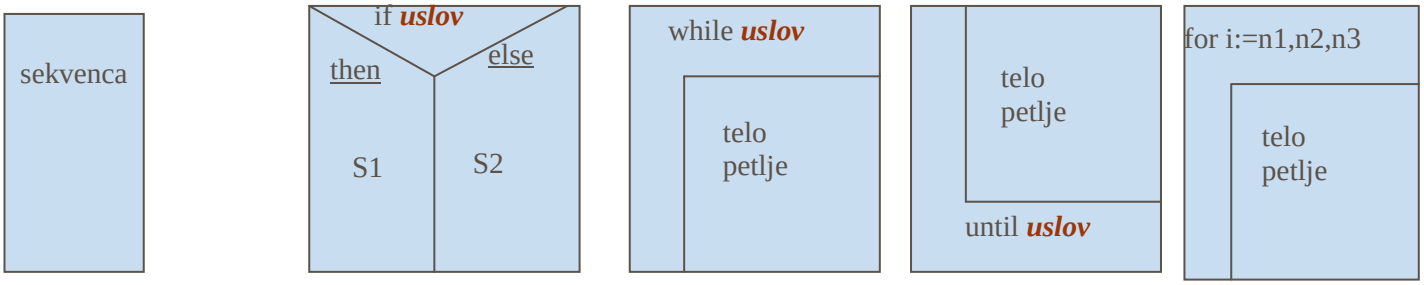
```
repeat
        niz_naredbi
until (uslov);
```

**Primer:**

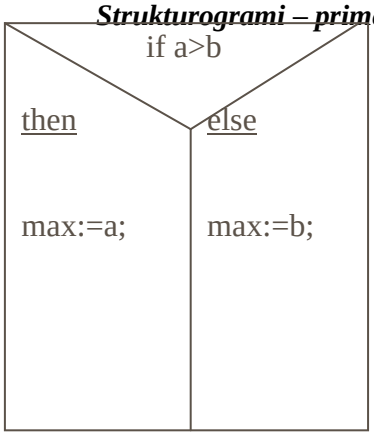
```
repeat
        r:="ostatak od m/n";
        m=n;
        n:=r;
until (r=0);
nzd:=m;
```

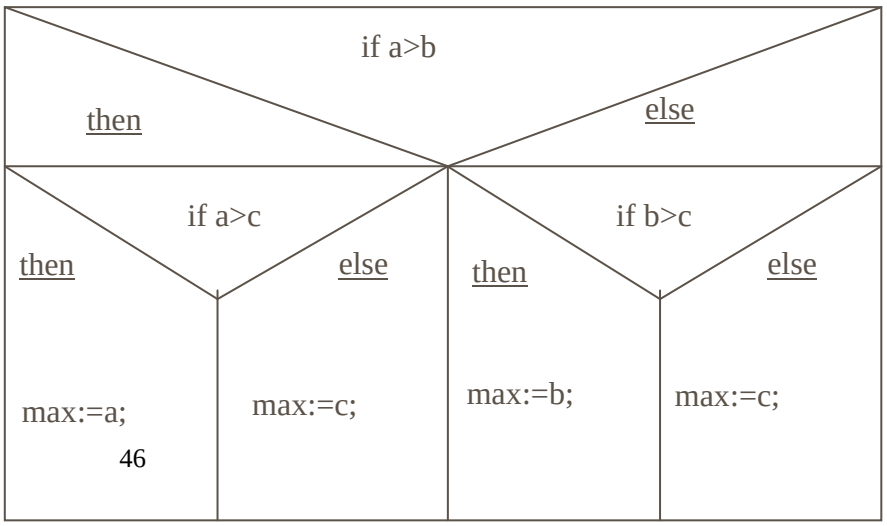
***Strukturogrami***

- Kombinacija grafičkog i pseudo koda;
  - Koriste se kao prikladna dokumentacija za već završene programe.
  - Program se piše tako da se popunjavaju određene geometrijske slike




***Strukturogrami – primer***



# 4.2. Sekvenca naredbi i blok

Sekvenca naredbi je jedan od osnovnih koncepata u strukturnim jezicima, uveden radi lakše implementacije drugih upravljačkih struktura. Prvi put se javlja u jeziku Algol 60, u kome se naziva i složena naredba, a definisan je kao niz naredbi ograničen zagradama *begin* i *end*:

```
begin
    naredba_1;
    …….
    naredba_n
end
```

Sekvenca naredbi se u Algol-u 60 može pojaviti svuda gde je sintaksom dozvoljena naredba. Time ovaj koncept postaje jako sredstvo apstrakcije naredbe.

U Algol-u 60 se prvi put javlja i koncept bloka kao uopštenje koncepta sekvence naredbi. Blok je po definiciji upravljačka struktura koja sadrži opise lokalnih promenljivih i sekvencu naredbi. Kao i sekvenca, zatvara se zagradama *begin* i *end*.

```
begin
    deklaracija_1;
    …….
    deklaracija_m;

    naredba_1;
    …….
    naredba_n
end
```

## 4.2.1. Globalne i lokalne promenljive

Globalna promenljiva (***global variable***) je takva promenljiva koja je deklarisana za upotrebu kroz ceo program. Trajanje globalne varijable je celokupno vreme izvršenja programa: varijabla se kreira kada program startuje, a uništava odmah po završetku programa.

Lokalna promenljiva (***local variable***) je takva promenljiva koja je deklarisana unutar nekog bloka, za upotrebu samo unutar tog bloka. Trajanje lokalne varijable je aktivacija bloka koji sadrži deklaraciju te varijable: varijabla se kreira na ulasku u blok, i uništava se na izlasku iz bloka.

Blok (***block***) je programska konstrukcija koja uključuje lokalne deklaracije. U svim programskim jezicima, telo neke procedure jeste blok. Neki jezici takođe imaju blok komande, kao na primer ‘‘{ . . . }’’ u C, C++, ili JAVA, ili ‘‘**declare** . . . **begin** . . . **end**;’’ u ADA. Aktivacija (***activation***) nekog bloka je vremenski interval u kome se taj blok izvršava. Partikularno, aktivacija procedure je vreme interval između poziva i vraćanja vrednosti. U toku jednog izvršenja programa blok može biti aktiviran nekoliko puta, tako da lokalna promenljiva može imati nekoliko trajanja.

Promenljive opisane u jednom bloku su lokalne promenljive tog bloka, a globalne za sve blokove sadržane u njemu. Ukoliko se u nekom bloku predefinišu promenljive već definisane u spoljašnjem bloku, u unutrašnjem bloku važe te nove definicije kao lokalne definicije bloka. Van bloka prestaje dejstvo lokalnih definicija unutrašnjeg bloka. Razmotrimo sledeći primer, koji je konstruisan sa idejom da ilustruje dejstvo lokalnih i globalnih definicija.

**Primer.** Globalne i lokalne promenljive.

```
A: begin real a; ... Pa;
     B: begin real b; ... Pb end;
     C: begin real c;  .. Pc;
                D: begin real d; . .. Pd end;
                E: begin real e;  ... Pe end;
        end;
      F: begin real f;  .. Pf;
                G; begin real g; ... Pg end;
        end;
```

**end**;

U ovom primeru postoji sedam blokova (označeni oznakama *A*:, *B*:, *C*:, *D*:, *E*:, *F*: i *G*:) i u svakom od ovih blokova opisana je po jedna promenljiva. Sledeća tabela daje podatke o svim blokovima u programu. Slovo *L* označava da je promenljiva lokalna, a *G* da je globalna.

| | **Blok** | | | | | | |
|---|---|---|---|---|---|---|---|
| **Promenljiva** | A | B | C | D | E | F | G |
| a | L | G | G | G | G | G | G |
| b | | L | | | | | |
| c | | | L | G | G | | |
| d | | | | L | | | |
| e | | | | | L | | |
| f | | | | | | L | G |
| g | | | | | | | L |

U sledećem primeru dat je ilustrovan pojam bloka kao i pojam lokalnih i globalnih promenljivih.

**Primer.** Doseg i sakrivanje u C++.

```
int x=0;           // globalno x
void f () {
  int y=x,         // lokalno y, globalno x;
      x=y;         // lokalno x, sakriva globalno x
  x=1;             // pristup lokalnom x
  ::x=5;           // pristup globalnom x
  { int x;         // lokalno x, sakriva prethodno x
    x=2;           // pristup drugom lokalnom x
  }
  x=3;             // pristup prvom lokalnom x
}
int *p=&x;         // uzimanje adrese globalnog x


#include<stdio.h>
int x=0;           // globalno x
void main ()
{  int y=x,         // lokalno y, globalno x;
       x=y;         // lokalno x, sakriva globalno x
  printf("x = %d\n",x);
  x=1;             // pristup lokalnom x
  printf("x = %d\n",x);
  ::x=5;           // pristup globalnom x
  printf("x = %d\n",x);
  { int x;         // lokalno x, sakriva prethodno x
    x=2;           // pristup drugom lokalnom x
   printf("x = %d\n",x);
  }
 // x=3;             // pristup prvom lokalnom x
  printf("x = %d\n",x);
  int *p=&x;
  printf("x = %d\n",*p);
}
```

**Primer.** Blok u jeziku C.

```
#include<stdio.h>
void main()
 { float a,b,c,d;
   a=1.0; b=2.0; c=3.0; d=4.0;
   printf("Spoljasnji blok a=%f b=%f c=%f d=%f\n",a,b,c,d);
   { float c,d,e,f;
     c=a+10; d=b+20; e=c+30; f=d+40;
```

```
      printf("Unutrasnji blok a=%f b=%f c=%f d=%f e=%f f=%f\n",
                    a,b,c,d,e,f);
    }
    printf("Spoljasnji blok a=%f b=%f c=%f d=%f\n", a,b,c,d);
}
```

U ovom primeru postoje dva bloka: prvi u kome se definišu promenljive *a*, *b*, *c* i *d* i drugi u kome su promenljive *c* i *d* predefinisane i promenljive *e* i *f* prvi put definisane. Kako je drugi blok unutar prvog dejstvo ovih definicija je sledeće:

- *a*, *b*, *c* i *d* su lokalne promenljive prvog bloka.
- *c*, *d*, *e* i *f* su lokalne promenljive drugog bloka.
- *a* i *b* važe i u drugom bloku, ali kao globalne promenljive.
- *c* i *d* su opisane i u prvom i u drugom bloku pa se u svakom od ovih blokova mogu koristiti nezavisno od njihove namene u drugom bloku.
- *e* i *f* se mogu koristiti samo u drugom bloku, dok su u prvom nedefinisane.

Kao posledicu ovako postavljenih definicija promenljivih naredbom za štampanje u umetnutom bloku štampaju se brojevi 1.0, 2.0, 11.0, 22.0, 41.0 i 62.0, dok naredba za štampanje u spoljašnjem bloku daje: 1.0, 2.0, 3.0 i 4.0.

Kako dejstvo lokalnih promenljivih prestaje po izlasku iz bloka, kod ponovnog ulaska u isti blok one su nedefinisane i potrebno je da im se ponovo dodele neke inicijalne vrednosti.

Sekvenca naredbi je u istom ovom obliku zastupljena i u Pascal-u, dok je struktura bloka izostavljena. U C postoje oba koncepta ali su *begin* i *end* zamenjeni zagradama { i }. To ilistruje sledeći primer.

Kod novijih jezika kod kojih je zastupljen koncept ključnih reči kojima se otvaraju i zatvaraju delovi upravlajčke strukture, sekvenca naredbi ne postoji kao odvojena struktura već je sastavni deo drugih upravljačkih struktura.

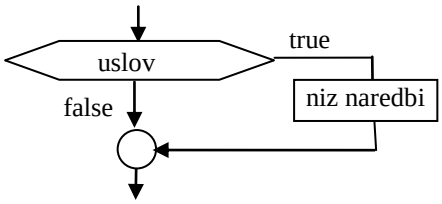
# 4.3. Struktura selekcije

Struktura selekcije omogućava definisanje više mogućih tokova programa i jedno je od osnovnih sredstava za pisanje fleksibilnih programa. Strukture selekcije dele se u dve osnovne grupe. Jednoj pripadaju takozvane *if*-selekcije, a drugoj višestruka selekcija.

If-selekcije se obično javljaju kao *if-then* i *if-then-else* struktura, pri čemu prva omogućava izbor posebnog toka u zavisnosti od određenog uslova, a druga izbor jedne ili druge alternative u zavisnosti od toga da li je određeni uslov ispunjen.

## 4.3.1. If-then struktura

Na sledećoj slici prikazan je dijagram toka koji ilustruje grafički prikaz ove strukture na nivou algoritma.




*If-then* struktura se obično implementira kao poseban, jednostavniji slučaj *if-then-else* strukture. Jedino kod nekih starijih jezika (BASIC, FOTRAN) ona postoji kao zasebna upravljačka struktura.

U FORTRAN-u se *if-then* struktura naziva logičkom IF naredbom i ima sledeću formu:

IF (logički izraz) naredba

Naredba koja sledi iza logičkog izraza izvršava se samo ako je vrednost logičkog izraza jednaka *TRUE*, inače se tok programa prenosi na sledeću naredbu.

Jedan od problema u primeni *if* naredbe jeste implementacija slučaja kada se izvršava veći broj naredbi u jednoj ili *then* ili *else* grani. U programskim jezicima koji ne podržavaju koncept sekvence naredbi, ova struktura se implementira koristeći *GOTO* naredbu.

Uobičajeni način korišćenja ove naredbe u FORTRAN-u IV ilustruje sledeći primer.

**Primer.** Logička *IF* naredba u FORTRAN-u. U naredbi je sa *Uslov* označena logička promenljiva.

```
      IF (.NOT. Uslov) GO TO 20
         I=1
         J=2
         K=3
20     CONTINUE
```

U ovom primeru, ako je vrednost promenljive *Uslov* jednaka *FALSE* program se nastavlja od naredbe sa oznakom 20. Za vrednost *TRUE* iste promenljive program se nastavlja naredbom I=1 koja sledi neposredno iza naredbe *IF*. Upotreba *GO TO* naredbe je bila neophodna s obzirom na nepostojanje sekvence naredbi u FORTRANu IV.

Uvođenjem koncepta sekvence naredbi, if-then naredba u Algol-u dobija drugačiju sintaksu, onakvu kakvu poznajemo u savremenim proceduralnim jezicima. U opštem slučaju njena struktura je definisana na sledeći način:

```
if (logički  izraz) then
   begin
      naredba_1;
      ...
      naredba_n
   end;
```

Sekvenca naredbi, obuhvaćena zagradama zagradama ili između *begin* i *end***,** sastavni je deo *if* naredbe. Ta sekvenca naredbi se izvršava za vrednost *true* logičkog izraza. Za vrednost *false* sekvenca naredbi se preskače i program nastavlja sa prvom naredbom koja sledi iza zagrade *end*.

**Primer.** *If-then* naredba u Algol-u 60.

```
if (uslov) then
  begin
    i:=1;
    j:=2;
    k:=3
   end;
```

If-then struktura se javlja u različitim oblicima u mnogim jezicima čiji su koreni u Algol-u 60, uključujući i FORTRAN 77 i FORTRAN 90.

U FORTRAN-u se može koristiti IF-THEN struktura, prema sledećoj sintaksi:

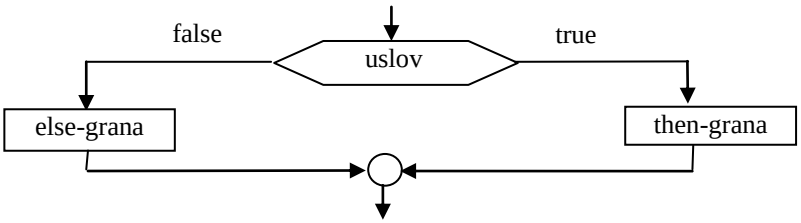
```
IF(uslov)
   naredba
END IF
```

Uočimo da se u ovom jeziku koriste ključne reči kojima se otvaraju i zatvaraju pojedini delovi strukture.

U jeziku Pascal je *if-then* struktura preuzeta iz Algol-a. U jeziku C se umesto reči *begin* i *end* koriste zagrade { i }, redom.

## 4.3.2. If-then-else struktura

Dijagram toka *if-then-else* strukture koji ilustruje smisao ove strukture na nivou algoritma prikazan je na slici.




Ova upravljačka struktura se prvi put javlja u Algol-u 60 i to u obliku:

```
if (logički_izraz)
  then naredba
  else naredba;
```

Naredba koja sledi iza ključne reči *then* predstavlja *then* granu programa, a naredba iza else predstavlja njenu *else* granu. Kada je vrednost logičkog izraza *true* izvršava se *then* grana, a kada je vrednost *false* - *else* grana programa.

Ova upravljačka struktura postoji u svim proceduralnim jezicima sa određenim varijacijama u sintaksi. U Pascal-u je prihvaćena sintaksa iz Algol-a. Sledeći primer ilustruje mogućnosti njene primene.

**Primer.**

```
if (broj = 0) then
    res  := 0
else
    res  := 1;
```

Ovako definisana *if-then-else* struktura ima niz nedostataka koji posebno dolaze do izražaja kada se *if* naredbe ugrađuju jedna u drugu. Na primer u Pascal-u je moguće napisati sledeću sekvencu naredbi:

```
if (sum = 0) then
  if (broj = 0)
     then res := 0
     else res := 1;
```

Ova sekvenca naredbi može da bude protumačena na dva različita načina u zavisnosti od toga da li *else* grana pripada prvoj ili drugoj *if* naredbi. U jezicima C, Pascal, kao i u više drugih jezika koji koriste isti koncept za razrešavanje ovakvih situacija: primenjuje se semantičko pravilo da se *else* grana uparuje sa najbližom neuparenom *then* granom. Očigledno je da u ovakvim slučajevima može da dođe do pogrešnog tumačenja pojedinih segmenata programa. U Algol-u se ovaj problem razrešava na sintaksnom nivou. Naime, u Algol-u nije dozvoljeno ubacivanje *if* naredbi u *then* granu već se nadovezivanje *if* naredbi može vršiti samo po *else* grani. Kada je neophodno da se po *then* grani vrši dalje grananje obavezna je upotreba zagrada kojima se naredba koja se ubacuje transformiše u takozvane proste naredbe koje se jedino mogu naći u *then* grani. Prethodni primer bi, u Algol-u, bio napisan kao u sledećem primeru.

**Primer.** Umetanje *if* naredbi u Algol-u.

```
if sum = 0 then
 begin
   if broj = 0
      then res := 0
      else res := 1
 end;
```

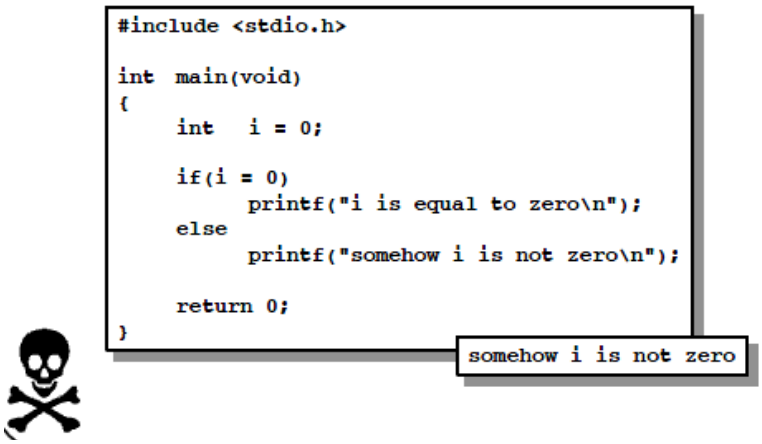
Ako u prethodnom primeru *else* grana treba da bude sastavni deo prve *if* naredbe dati deo programa treba da bude napisan kao u primeru koji sledi.

**Primer**. Umetanje *if* naredbi u Algol-u (druga verzija).

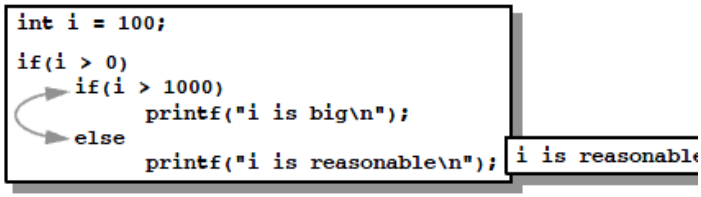
```
if sum = 0 then
  begin
     if broj = 0
        then  res   := 0
```

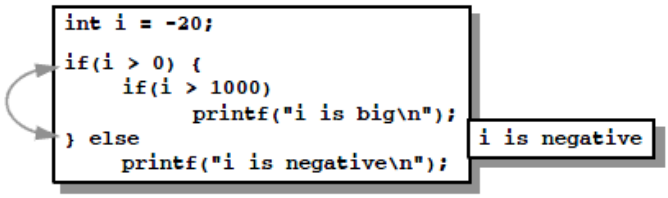
```
    end
else res  := 1;
```

**Napomena**: Jedna od najčešćih grešaka je da se test za jednakost "==" zameni sa dodeljivanjem "=".




**Primer**. *Else* se pridružuje najbližem *if*.




## Primeri u C.

**Primer.** Napisati program kojim se dati brojevi *x*, *y*, *z* udvostručuju ako je *x*>= *y*>= *z*, a u protivnom menjaju znak.

```
main()
{ float x,y,z;
   printf("Zadati x,y,z:\n");     scanf("%f%f%f",&x,&y,&z);
   if((x>=y) && (y>=z))     { x*=2; y*=2; z*=2; }
   else                     { x=-x; y=-y; z=-z; }
   printf("x=%f y=%f z=%f", x,y,z);
}
```

Kod *else-if* iskaza je važno da rezervisana reč *else* odgovara prethodnom slobodnom *if*, ukoliko to nije drugačije određeno velikim zagradama.

**Primer.** Izračunati

$$y = \begin{cases} -5, & x<0, \\ x+2, & 0<=x<1, \\ 3x-1, & 1<=x<5, \\ 2x, & x>=5. \end{cases}$$

```
void main()
{ float x,y;
   scanf("%f",&x);
```

```
    if(x<0) y=-5;
    else if(x<1) y=x+2;
    else if(x<5) y=3*x-1;
    else y=2*x;
    printf("y= %f\n", y);
 }
```

**Primer.** Urediti tri zadata realna broja u poredak $x < y < z$.

```
#include<stdio.h>
main()
 { float x,y,z,p;
     scanf("%f%f%f",&x,&y,&z);
     if(x>y)    { p=x; x=y; y=p; }
     if(x>z)    { p=x; x=z; z=p; }
     if(y>z)    { p=y; y=z; z=p; }
     printf("x= %f y= %f z= %f\n",x,y,z);
  }
```

**Primer.** Data su tri realna broja u poretku $x < y < z$. Umetnuti realni broj $t$ tako da među njima bude odnos $x < y < z < t$.

```
void main()
 { float x,y,z,t,p;
    scanf("%f%f%f%f",&x,&y,&z,&t);
    if(t<x)      { p=t; t=z; z=y; y=x; x=p; }
    if(t<y)      { p=t; t=z; z=y; y=p; }
    if(t<z)      { p=t; t=z; z=p; }
    printf("x= %f y= %f z= %f t= %f\n",x,y,z,t);
 }
```

**Primer.** Diskutovati rešenje sistema linearnih jednačina

$$a_1 x + b_1 y = c_1$$
$$a_2 x + b_2 y = c_2.$$

```
void main()
 { float a1,b1,c1,a2,b2,c2,d,dx,dy,x,y;
   printf("Koeficijenti prve jednacine?"); scanf("%f%f%f",&a1,&b1,&c1);
   printf("Koeficijenti druge jednacine?");
   scanf("%f%f%f", &a2,&b2,&c2);
   d=a1*b2-a2*b1;  dx=c1*b2-c2*b1;  dy=a1*c2-a2*c1;
   if(d!=0){x=dx/d; y=dy/d; printf("x=%f y=%f\n",x,y);}
   else if(d==0 && dx==0 && dy==0)
    { printf("Sistem je neodređen ");
      printf("resenje je oblika X,(%f-%fX)/%f\n",c1,a1,b1);
    }
   else printf("Sistem je nemoguc\n");
 }
```

**Primer.** Rešavanje jednačine $ax^2+bx+c=0$ u jezicima Pascal i C.

```
program Kvadratna_Jednacina;

var a,b,c,D,x1,x2,x,Re,Im:real;

begin
  readln(a,b,c);
  if a<>0 then begin
    D:=sqr(b)-4*a*c;
    if(D>0) then begin
      x1:=(-b+sqrt(D))/(2*a); x2:=(-b-sqrt(D))/(2*a);
      writeln('x1= ',x1:0:3,' x2= ',x2:0:3);
    end
    else if D=0 then begin
```

```
      x1:=-b/(2*a);    writeln('x1= ',x1:0:3,' x2= ',x1:0:3);
    end
    else begin
      Re:=-b/(2*a); Im:=sqrt(-D)/(2*a);
      write(Re:0:3,'+i*',Im:0:3);  writeln('  ',Re:0:3,'-i*',Im:0:3);
    end
  end
  else if b<>0 then begin
    x1:=-c/b; writeln(x1:0:3);
  end
  else if(c<>0) then   writeln('Jednacina nema resenja')
  else writeln('Identitet');
end.
```

```
#include<stdio.h>
#include<math.h>
void main()
{ float a,b,c,D,x1,x2,Re,Im;
  scanf("%f%f%f",&a,&b,&c);
  if(a)
  { D=b*b-4*a*c;
    if(D>0)
      { x1=(-b+sqrt(D))/(2*a); x2=(-b-sqrt(D))/(2*a);
        printf("x1= %f x2= %f\n",x1,x2);
      }
      else if(D==0)
      { x1=-b/(2*a); printf("x1= %f x2= %f\n",x1,x1); }
      else
      { Re=-b/(2*a); Im=sqrt(-D)/(2*a);
        printf("%f+i*%f, %f-i*%f\n",Re,Im,Re,Im);
      }
  }
  else if(b)
  { x1=-c/b;   printf("%f\n",x1);  }
  else if(c)   printf("Jednacina nema resenja\n");
  else printf("Identitet\n");
}
```

**Primer.** U gradu *A* se nalazi zaliha goriva od *V* ($0 < V < 2000000000$) litara, od koje kamion-cisterna treba da dostavi što je moguće veću količinu u grad *B*. Od grada *A* do gada *B* ima tačno *d* ($0 < d \leq 2000$) kilometara. Cisterna troši litar na jedan kilometar, a može da primi ukupno *C* ($0 < C \leq 5000$) litara za prevoz i potrošnju.

Napisati program koji za date *V*, *d*, *C*, ispisuje koliko najviše goriva može da se dostavi iz *A* u *B*, i koliko PRI TOME najviše može ostati u *A*. Cisterna može ostati u gradu koji daje povoljniji ishod.

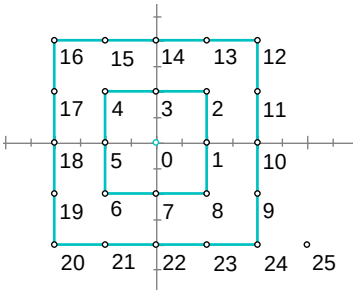
Test primer:

```
2000 100 1000                              1700  0
```

```
#include <stdio.h>
void main ()
 { long V, d, C, A, B;
   printf("Unesi kolicinu u gradu A: "); scanf ("%ld",&V);
   printf("Unesi rastojanje između gradova: "); scanf ("%ld",&d);
   printf("Unesi kapacitet cisterne: "); scanf ("%ld",&C);
   A=V; B=0;
   if((C>d) &&(V>=C))
    { if(C>2*d)
       { A=V%C;   B=(C-2*d)*(V/C);
         if(A<=d) B=B+d;
```

```
          else
           { if(A-d>d){B=B+A-d; A=0;}
             else B=B+d;
           }
        }
       else      {A=V-C; B=C-d;}
     }
   printf ("A= %ld B= %ld\n", A, B);
 }
```

**Primer.** Ispisana je sledeća spirala brojeva oko zamišljenog koordinatnog sistema:




Za proizvoljan prirodan broj *n* odrediti njegove koordinate u tom koordinatnom sistemu.

**(PASCAL)**

```
PROGRAM spirala(input,output);
  VAR n, r, x, y :integer;
  BEGIN
     read(n);    r := trunc((sqrt(n) + 1)/2);
     IF n<=4*sqr(r) - 2 * r THEN
        BEGIN
           x:=r;     y:=n - (4 * sqr(r) - 3 * r)
        END
     ELSE IF n <=sqr(r) THEN
        BEGIN
           y:=r;     x := (4*sqr(r) -r) - n
        END
    ELSE IF n<=4*sqr(r) + 2 * r THEN
        BEGIN
           x:=-r;    y:=(4*sqr(r) + r) - n
        END
    ELSE BEGIN
           y:=-r;    x:= n - (4*sqr(r) + 3 * r)
         END;
   WRITE ('n = ',n,'x = ', x,'y = ', y)
 END.
```

**(C)**

```
#include<stdio.h>
#include<math.h>
main()
 { int n,r,x,y;
    scanf("%d", &n);     r=(sqrt(n)+1)/2;
    if(n<=4*r*r-2*r)        { x=r; y=n-(4*r*r-3*r); }
    else if(n<=4*r*r)       { y=r; x=4*r*r-r-n; }
    else if(n<=4*r*r+2*r)       { x=-r; y=4*r*r+r-n; }
    else       { y=-r; x=n-(4*r*r+3*r); }
    printf("\n Za n=%d koordinate su x=%d y=%d\n",n,x,y);
 }
```

### 4.3.3. Operator uslovnog prelaza u C

Operator uslovnog prelaza ?: je neobičan. Pre svega, to je ternarni operator, jer uzima tri izraza za svoje argumente. Način na koji se on primenjuje je takođe neobičan:

exp1 ? exp2 : exp3;

Prvi argument je pre znaka pitanja '?'. Drugi argument je između '?' i ':', a treći posle ':'. Semantika ovakvog izraza je sledeća. Prvo se izračunava izraz *exp*1. Ako je njegova vrednost različita od 0 (tj. ako je jednaka *true*), tada se evaluira izraz *exp*2, i njegova vrednost je vrednost celog izraza. Ako je vrednost izraza *exp*1 nula (*false*), tada se evaluira izraz *exp*3, i njegova vrednost se vraća kao rezultat. Na taj način, ovaj izraz predstavlja zamenu za *if-then-else* naredbu.

**Primer.** Umesto izraza

```
if(y<z)  x=y;
else  x=z;
```

kojim se izračunava x=min{y,z}, može se pisati

```
x=(y < z) ? y:z;
```

Zagrade se mogu izostaviti.

Prioritet operatora uslovnog prelaza je veći od prioriteta operatora dodeljivanja, a njegova asocijativnost je "s desna na levo".

**Primer.** Vrednost izraza

```
(6>2)?1:2
```

jednaka je 1, dok je vrednost izraza

```
(6<2)?1:2
```

jednaka 2.

Ako je izraz posle ':' takođe uslovni izraz, dobijamo else-if operator.

**Primeri.**

1. Vrednost izraza

$$s = \begin{cases} -1, & x < 0, \\ x * x, & x >= 0 \end{cases}$$

može se izračunati pomoću

```
s=(x < 0) ? -1 : x*x;
```

2. Vrednost *s* = sgn(*broj*) se može izračunati pomoću izraza

```
s=(broj < 0) ? -1 : ((broj==0) ? 0 : 1);
```

3. Bez korišćenja operatora *if* napisati program koji za zadate *x*, *y* izračunava

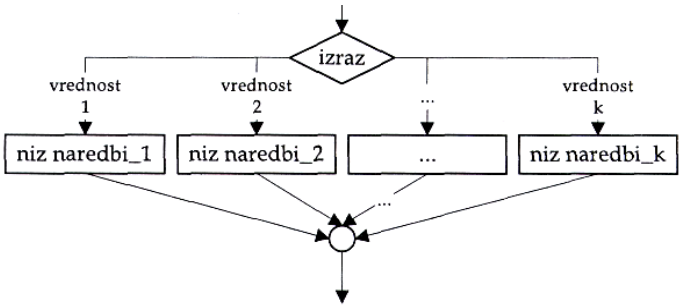
$$z = \begin{cases} \min\{x,y\}, & y >= 0, \\ \max\{x^2,y^2\}, & y<0. \end{cases}$$

```
z=(y>=0) ? ((x<y) ? x:y) : ((x*x>y*y) ? x*x : y*y);
```

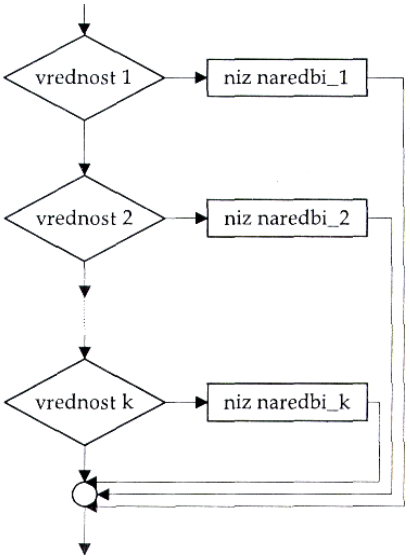
## 4.4. Struktura višestruke selekcije

Struktura višestruke selekcije omogućava izbor jedne od više mogućih naredbi ili grupa naredbi u zavisnosti od vrednosti određenog uslova. Struktura višestruke selekcije se koristi kao zamena za komplikovane upravljačke strukture sledećeg oblika, u kojima se funkcija *if* mnogo puta ponavlja.

Graf na sledećoj slici ilustruje ovu upravljačku strukturu na nivou algoritma.




Kako se svaka višestruka selekcija može simulirati običnim *if-then-else* grananjem, na sledećoj slici prikazan je dijagram toka koji odgovara višestrukom grananju.




Ova struktura koja je u svom razvoju pretrpela mnogo izmena do jednog modernog oblika koji se danas u manje-više sličnim varijantama javlja u svim programskim jezicima.

## 4.4.1. Višestruka selekcija u C

U programskom jeziku C naredba za višestruko grananje ima dosta zastarelu koncepciju. To je *switch* naredba koja u opštem slučaju ima sledeći oblik:

```
switch(izraz)
 { case vrednost11: [case vrednost12: … ]
     operator11
     ......
     break;
   case vrednost21: [case vrednost22: …]
     operator21
     ......
     break;
   ......
   case vrednostn1: [case vrednostn2: …]
     operatorn1
     ......
     break;
   default:
     operator01
     ......
     break;
 }
 sledeća naredba;
```

Semantika ove naredbe je sledeća: izračunava se vrednost izraza *izraz*, koji se naziva *selektor*. Vrednost izraza *izraz* mora da bude celobrojna (ili znakovna, koja se automatski konvertuje u odgovarajuću celobrojnu vrednost). Dobijena vrednost se upoređuje sa vrednostima *vrednost*11, *vrednost*12, ..., koji moraju da budu celobrojne konstante ili celobrojni konstantni izrazi. Ove vrednosti se mogu posmatrati kao obeležja za određenu grupu operatora. Ako je pronađeno obeležje čija je vrednost jednaka vrednosti izraza *izraz*, izračunavaju se operatori koji odgovaraju toj vrednosti. Ključna reč *break* predstavlja kraj jedne *case* grane, odnosno kraj grupe operatora sadržanih u nekoj *case* grani. Ako nije pronađena vrednost u *case* granama jednaka vrednosti izraza *izraz*, izračunavaju se iskazi u *default* grani; ako je izostavljena alternativa *default*, neće se izvršiti ni jedna od alternativa operatora *switch*.

Izraz prema kome se vrši selekcija i konstante u pojedinim granama treba da budu `integer` tipa. Naredbe u granama mogu da budu obične ili složene naredbe i blokovi.

Iako na prvi pogled ova naredba liči na ono što imamo u Pascal-u, radi se o nestrukturnoj naredbi koja se izvršava tako što se izborom grane određuje samo mesto odakle naredba počinje, a iza toga se nastavlja njen sekvencijalni tok. To konkretno znači da se naredba u *default* grani izvršava uvek kao i to da se naredba u *k*-toj grani izvršava uvek kada je selektovana neka od prethodnih grana. Da bi se prekinulo upravljanje pod dejstvom ove naredbe kada se izvrši selektovana grana, odnosno da bi se postigla semantika *case* strukture u Pascal-u, potrebno je da se na kraju svake grane upravljanje eksplicitno prenese na kraj cele *switch* naredbe. U te svrhe u C-u se koristi naredba *break* koja predstavlja izlazak iz *switch* naredbe. Kôdom iz primera u C-u je realizovano višestruko grananje iz prethodnog primera gde je korišćena *case* struktura iz Pascal-a.

**Primer.** Program kojim se simulira digitron. Učitavaju se dva operanda i aritmetički operator. Izračunati vrednost unetog izraza.

```
void main()
 { float op1, op2;
    char op;
    printf("Unesite izraz \n");
    scanf("%f %c %f", &op1, &op, &op2);
    switch(op)
    { case '+':   printf("%f\n", op1+op2);     break;
       case '-':  printf("%f\n", op1-op2);     break;
       case '*':  printf("%f\n", op1*op2);     break;
       case '/':  printf("%f\n", op1/op2);     break;
       default:   printf("Nepoznat operator\n");
    }
 }
```

Operator *break* se koristi u okviru *switch* operatora (kasnije i u *while*, *do*, *for* operatorima) da bi se obezbedio izlaz neposredno na naredbu iza *switch* strukture. Ukoliko se iza neke grupe operatora u *switch* operatoru ispusti *break* operator, tada se u slučaju izbora te grupe operatora izvršavaju i sve preostale alternative do pojave *break* operatora ili do kraja *switch* operatora.

**Primer.** Posmatrajmo sledeći program.

```
 void main()
 { int ocena;
    scanf("%d",&ocena);
    switch(ocena)
      { case 5: printf("Odlican\n"); break;
        case 4: printf("Vrlo dobar\n");
        case 3: printf("Dobar\n");
        case 2: printf("Dovoljan\n"); break;
        case 1: printf("Nedovoljan\n"); break;
        default: printf("Nekorektna ocena\n:");
      }
 }
```

Ukoliko je *ocena* = 4 ispisuje se sledeći rezultat:

```
Vrlo Dobar
Dobar
Dovoljan
```

Operator *switch* se može realizovati pomoću većeg broja *if* operatora. Međutim, programi napisani pomoću *switch* operatora su pregledniji.

**Primer.** Za zadati redni broj meseca ispisati broj dana u tom mesecu.

```
void main()
{ int mesec;
   char ch;   /* Da li je godina prestupna */
   printf("Unesi redni broj meseca: ");
   scanf("%d", &mesec);
   switch(mesec)
     { case 1:case 3:case 5:case 7:case 8:case 10:case 12:
                printf("31 dan\n");
           break;
        case 4: case 6: case 9: case 11:
                printf("30 dana\n);
           break;
        case 2: printf("Da li je godina prestupna? ");
           scanf("%c%c",&ch,&ch);
           /* prvo ucitavanje je fiktivno, uzima enter */
           if((ch=='D') || (ch=='d')) printf("29 dana\n);
           else printf("28 dana\n");
           break;
        default: printf("Nekorektan redni broj\n");
     }
}
```

**Primer.** Odrediti sutrašnji datum.

```
void main()
 {unsigned int d,m,g,prestupna;
  printf("Unesi datum u obliku ddmmgg:\t");
  scanf("%2d%2d%4d",&d,&m,&g);
  if(d<1 || d>31 || m<1 || m>12)
    { printf("Nepravilan datum\n"); exit(0); }
   prestupna=((g%4==0) && (g%100!=0)) ||
             ((g%100==0) && (g%400==0));
  if(m==2 && d>28+prestupna)
    { printf("Nepravilan datum\n"); exit(0); }
  switch(d)
   { case 31:
      switch(m)
       { case 1:case 3:case 5:case 7:case 8:case 10:  d=1; m++; break;
          case 12:  d=1; m=1; g++; break;
          default: { printf("%2d. mesec ne moze imati 31 dan\n",m);
                     exit(0);
                   }
       }
     break;
     case 28: if(!prestupna && (m==2)) { d=1; m=3; }
              else d=29;
              break;
     case 29: if(prestupna && (m==2)) { d=1; m=3; }
              else d=30;
              break;
     case 30: switch(m)
```

```
                { case 4: case 6: case 9: case 11:
                    d=1; m++;
                    break;
                  case 2:printf("Februar ne moze imati 30 ana\n");
                         exit(0);
                  default:d=31;
                }
                break;
     default:d++;
   }
  printf("Sledeci dan ima datum:\t%2d\t%2d\t%4d\n",d,m,g);
}
```

Operator *switch* se može koristiti umesto *if* operatora. Na primer, umesto operatora

```
if(a>b) max=a;
else max=b;
```

može se pisati

```
switch(a>b)
{   case 0: max=b; break;
     case 1: max=a;
}
```

Očigledno da ovo nije praktično.

# 4.5. Programske petlje

U svakom programskom jeziku obično postoje upravljačke strukture kojima može da se definiše višestruko ponavljanje određene sekvence naredbi na istim ili različitim podacima. To su programske petlje ili strukture iteracije. Ciklusi omogućavaju da se skupovi srodnih podataka koji se odnose na isti pojam obrađuju na isti način.

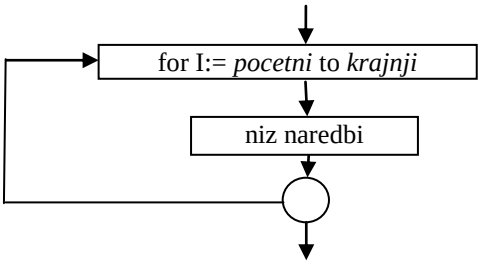
Prema jednoj podeli, programske petlje se mogu podeliti na:
- brojačke i
- iterativne.

Kod brojačkih petlji se unapred može reći koliko puta se ponavlja telo petlje. Kod iterativnih programskih ciklusa, nije unapred poznato koliko puta se izvršava telo petlje. Kod ovakvih ciklusa, kriterijum za njihov završetak je ispunjenje određenog uslova.

Prema drugoj podeli, programske petlje se dele na:
- petlje sa preduslovom i
- petlje sa postuslovom.

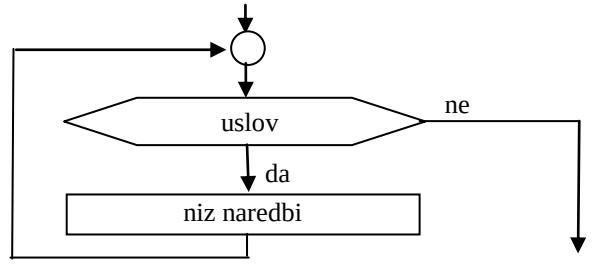
Kod petlji sa preduslovom, izlazni kriterijum se proverava pre nego što se izvrši telo petlje, dok se kod petlji sa postuslovom prvo izvrši telo petlje pa se zatim proverava izlazni kriterijum.

U savremenim programskim jezicima skoro bez izuzetaka postoje naredbe za definisanje petlji sa unapred određenim brojem prolaza. To su tzv. brojačke petlje kod kojih postoji brojač (indeks petlje) kojim se upravlja izvršavanjem petlje. Struktura brojačke petlje u Pascal-u je prikazana na sledećoj slici.

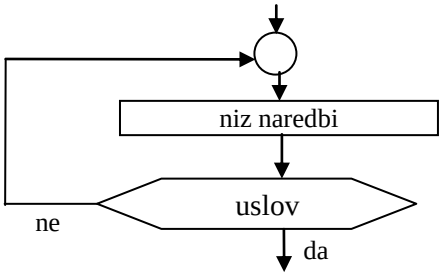



U ovoj petlji se naredbe u telu petlje izvršavaju za sve vrednosti brojačke promenljive *I* između vrednosti izraza *pocetni* i vrednosti izraza *krajnji*.

Sa prihvatanjem strukturnog programiranja u novijim jezicima se pojavljuje i logički kontrolisana petlja kojom se realizuje *while* (*while-do*) struktura. Ova struktura predviđa ponavljanje izvršavanja tela petlje sve dok uslov u zaglavlju petlje ima vrednost *true*. Kod ovakvih petlji nije unapred poznat broj izvršavanja tela petlje. Za ove petlje se često koristi i termin *pretest* petlje, jer se ispitivanje uslova vrši pre svakog izvršavanja petlje.




Strukturnim programiranjem definisana je i *until* struktura (*do-while* struktura) koja predviđa post-test uslova, odnosno ispitivanje uslova posle izvršenja tela petlje. *Repeat-until* struktura je prikazana na sledećoj slici.




Poseban oblik petlji sa postuslovom je *do-while* naredba u C. Ovakva struktura sa postuslovom se završava kada uslov nije ispunjen.

## 4.5.1. Programske petlje u C

### While naredba u C

Ovom naredbom se realizuju programski ciklusi sa nepoznatim brojem ponavljanja, zavisno od određenog uslova. Operator *while* se koristi prema sledećoj sintaksi:

```
while(izraz)
    operator
```

Efekat ove naredbe je da se telo *while* ciklusa, označeno sa *operator* izvršava dok je vrednost izraza izraz jednaka *true* (nenula). Kada vrednost izraza izraz postane *false* (nula), kontrola se prenosi na sledeću naredbu. Telo ciklusa, označeno sa operator, izvršava se nula ili više puta. Operator može biti prost ili složen. Svaki korak koji se sastoji iz provere uslova i izvršenja operatora naziva se "iteracija". Ako odmah po ulasku u ciklus *izraz* ima vrednost "netačno" (nula), operator se neće izvršiti.

**Primer.** Maksimum *n* unetih brojeva.

```
main()
 { int i, n;
   float max, x;
   printf("Koliko brojeva zelite? ");   scanf("%d", &n);
   while (n<=0)
   {  printf("Greska, zahtevan je ceo pozitivan broj.\n");
      printf("Unesite novu vrednost: ");    scanf("%d", &n);
   }
   printf("\n Unesite %d realnih brojeva: ", n);
   scanf("%f", &x);   max = x;   i=1;
   while (i<=n)
   { scanf("%f", &x);  if(max<x) max = x;  ++i;  }
   printf("\n Maksimalni od unetih brojeva: %g\n",max);
 }
```

## Primeri u jeziku C

**Primer.** Izračunati $\sqrt{a}$ prema iterativnoj formuli

$$x_0 = a/2;\ x_{i+1} = x_i - (x_i^2 - a)/(2x_i) = x_i + 1/2*(a/x_i - x_i)\ ,\ i = 0, 1, \ldots$$

Iterativni postupak prekinuti kada se ispuni uslov $|x_{i+1} - x_i| < 10^{-5}$.

```
#include <stdio.h>
main()
 { float a,x0,x1;
    scanf("%f",&a)
    x0=a/2;   x1=(x0*x0-a)/(2*x0);
    while(fabs(x1-x0)<1E-5) { x0=x1; x1=(x0*x0-a)/(2*x0); }
    printf("%f\n",x1);
 }
```

**Primer.** Izračunati aritmetičku sredinu niza brojeva različitih od nule. Broj elemenata u nizu nije unpred poznat.

```
void main()
 { int brojac=0;
   float suma=0, stopcode=0, broj;
    printf("Daj niz znakova zavrsen sa %d\n",stopcode);
    scanf("%f",&broj);
    while(broj != stopcode)
      { suma+=broj; brojac++;   scanf("%f",&broj);       }
    printf("Srednja vrednost= %f\n",suma/brojac);
 }
```

**Primer.** Dejstvo operatora ++ i - - u ciklusima.

```
#include <stdio.h>
void main()
 { int i=0;
    printf("Prva petlja :\n");
    while(i<5)  { i++; printf("%5d",i); }    /* 1  2  3  4  5*/
    printf("\nDruga petlja  :\n");
    i=0;
    while(i++<5) printf("%5d",i); /* 1  2  3  4  5*/
    i=0;
    printf("\nTreca petlja :\n");
    while(++i<5) printf("%5d",i); /* 1  2  3  4*/
 }
```

**Primer.** Suma celih brojeva unetih preko tastature.

```
#include <stdio.h>
main()
 { int x,sum=0;
    while(scanf("%d",&x)==1)   sum+=x;
    printf("Ukupan zbir je  %d\n",sum);
 }
```

**Primer.** Ispitati da li je zadati prirodan broj *n* prost.

```
#include <math.h>
void main()
 { int i,n,prost;
    scanf("%d",&n);
    prost=(n%2!=0) || (n==2);   i=3;
    while(prost && i<=sqrt(n))
     { prost = n%i!=0; i+=2; }
```

```
    if(prost) printf("%d je prost\n",n);
    else printf("%d nije prost\n",n);
 }
```

**Primer.** Dati su tegovi mase $3^k$ kg, $k$ = 0,1,2,... i to iz svake vrste po jedan. Napisati program koji određuje koje tegove bi trebalo postaviti levo a koje desno da bi se izmerio predmet težine $A$ kg koji se nalazi na levom tasu.

```
#include <stdio.h>
void main()
{ int dteg,lteg,s,a;
  printf("Unesite tezinu\n");
  scanf("%d",&a);
  s=1;
  while (a>0)
  { dteg=a%3;
    if (dteg==2){ dteg=0; lteg=1;   }
    else lteg=0;
    printf("Teg od %dkg staviti levo: %d, desno: %d\n",s,lteg,dteg);
    s*=3;
    a=(a+lteg)/3;
  }
}
```

```
#include <stdio.h>

void main() {
  int t, p = 0;
  scanf("%d", &t);
  while (t > 1) {
      if (t % 3 == 1)
         {   printf("3^%d ide na desnu\n", p);  t--;   }
      else if (t % 3 == 2)
         {   printf("3^%d ide na levu\n", p);  t++;   }
      else {  p++; t /= 3;    }
  }
  printf("3^%d ide na desnu\n", p);
}
```

**Primer.** Izračunati

$$\sum_{k=0}^{\infty}(-1)^k \frac{x^{2k}}{(2k)!},\ x<\pi/2$$

.

Sumiranje prekinuti kada je apsolutna vrednost poslednjeg dodatog člana manja od zadate tačnosti $e$.

Koristi se

$$a_k = \frac{-x^2}{2k(2k-1)} a_{k-1}$$

```
void main()
 { double c,a,e,x;
    int k=0;
    scanf("%lf%lf",&x, &e);
    while(fabs(a)>=e)
      { k++; a=-(x*x/(2*k*(2*k-1)))*a; c+=a; }
    printf("cos(%.2lf)=%.7lf",x,c);
 }
```

**Primer.** Heksadekadni broj prevesti u dekadni. Unošenje heksadekadnog broja prekinuti kada se unese karakter koji ne pripada skupu karaktera '0', ... , '9', '*A*', ... , '*F*'.

```
#include <stdio.h>
void main()
{int broj,u,k;
   char  cif;
   broj=0;u=1;
   printf("unesi heksadekadni broj\n"); scanf("%c",&cif);
   u=((cif<='F')&&(cif>='0'));
   while (u)
    { if ((cif>='A')&&(cif<='F')) k=cif-55;    else k=cif-48;
      broj=broj*16+k;   scanf("%c",&cif); u=((cif<='F')&&(cif>='0'));
    }
   printf("%d\n",broj);
}
```

**Primer.** Napisati program koji simulira rad džepnog kalkulatora. Program učitava niz brojnih vrednosti razdvojenih znakovima aritmetičkih operacija +, -, *, / kao i izračunavanje vrednosti izraza koji je definisan na ovaj način. Aritmetičke operacije se izvršavaju s leva na desno, bez poštovanja njihovog prioriteta.

```
#include <stdio.h>
void main()
 { double result, num;
   char op;
   printf("\n\n");  scanf("%lf" , &num); scanf("%c" , &op);
   result = num ;
   while (op != '=')
      { scanf("%lf" , &num) ;
        switch (op)
          {  case '+' : result += num ; break;
             case '-' : result -= num ; break;
             case '*' : result *= num ; break;
             case '/' : result /= num ; break;
          }
        scanf("%c" , &op);
      }
      printf("Rezultat je %.10lf." , result) ;
  }
```

**4.** Dati su tegovi mase $3^k$ kg, $k = 0,1,2,...$ i to iz svake vrste po jedan. Napisati program koji određuje koje tegove bi trebalo postaviti levo a koje desno da bi se izmerio predmet težine $A$ kg koji se nalazi na levom tasu.

```
#include <stdio.h>
void main()
{
      int dteg,lteg,s,a;
      printf("Unesite tezinu\n");
      scanf("%d",&a);
      s=1;
      while (a>0)
      {     dteg=a%3;
            if (dteg==2)
            {
                  dteg=0;
                  lteg=1;
            }
            else lteg=0;
            printf("Teg od %dkg staviti levo: %d, desno:
                              %d\n",s,lteg,dteg);
            s*=3;
            a=(a+lteg)/3;
```

```
      }
}
```

## Primena while ciklusa u obradi teksta u C

Često puta se pri radu sa tekstovima umesto funkcija *scanf*() i *printf*() koriste bibliotečke funkcije *getchar*() i *putchar*(). Pre korišćenja ovih funkcija mora da se navede pretprocesorska direktiva `#include<stdio.h>`. Ove funkcije se koriste za ulaz i izlaz samo jednog karaktera. Funkcija *putchar*() ima jedan znakovni argument. Pri njenom pozivu potrebno je da se unutar zagrada navede znakovna konstanta, znakovna promenljiva ili funkcija čija je vrednost znak. Ovaj znak se prikazuje na ekranu. Funkcija *getchar*() nema argumenata. Rezultat njenog poziva je znak preuzet iz tastaturnog bafera. Ovom funkcijom se ulazni znak unosi u program.

**Primer.** Izračunati koliko je u zadatom tekstu uneto znakova '*a*', '*b*', zajedno '*c*' i '*C*', kao i broj preostalih karaktera.

```
#include <stdio.h>
void main()
 { int c, a_count=0, b_count=0, cC_count=0, other_count=0;
    while((c=getchar()) != EOF)
      switch (c)
      { case 'a':   ++a_count;    break;
        case 'b':  ++b_count;       break;
        case 'c': case 'C':   ++cC_count;      break;
        default:    ++other_count;
      }

      printf("\n%9s%5d\n%9s%5d\n%9s%5d\n%9s%5d\n%9s%5d\n",
          "a_count:", a_count, "b_count:", b_count,
          "cC_count:", cC_count, "other:", other_count,
          "Total:", a_count+b_count+cC_count+other_count);
 }
```

**Primer.** Unos teksta je završen prelazom u novi red. Prebrojati koliko je u unetom tekstu znakova uzvika, znakova pitanja a koliko tačaka.

```
#include <stdio.h>
void main()
 { char c;
    int uzv,upt,izj;
    uzv=upt=izj=0;
    while((c=getch())!='\n')
        switch(c)
          { case '!':++uzv; break;
            case '?':++upt; break;
            case '.':++izj; break;
          }
    printf("\n");  printf("Uzvicnih ima %d\n",uzv);
    printf("Upitnih ima %d\n",upt); printf("Izjavnih ima %d\n",izj);
  }
```

Kod organizacije while ciklusa mora se voditi računa o tome da telo ciklusa menja parametre koji se koriste kao preduslov za ciklus, tako da posle određenog broja iteracija postane "netačan". U protivnom, ciklus će biti beskonačan.

## Do-while naredba u C

*Do-while* naredba je varijanta while naredbe. Razlikuje se od while naredbe po tome što je uslov na kraju ciklusa:

```
do
      operator
while(izraz);
```

sledeća naredba;

Telo ciklusa je označeno sa *operator*, i izvršava se jedanput ili više puta, sve dok izraz ne dobije vrednost 0 (*false*). Tada se kontrola prenosi na sledeću naredbu.

## Primeri

**Primer.** Ispis celog broja s desna na levo.

```
void main()
 { long broj;
    printf("Unesite ceo broj"); scanf("%ld",&broj);
    printf("Permutovani broj");
    do  { printf("%d",broj%10);  broj /= 10; }  while(broj);
    printf("\n");
 }
```

**Primer.** Izračunati približno vrednost broja $\pi = 3.14159$ koristeći formulu

$$\pi/4 = 1-1/3+1/5-1/7+\ldots$$

Sumiranje prekinuti kada apsolutna vrednost člana koji se dodaje bude manja od zadate vrednosti *eps*.

```
void main()
{ int znak=1;
   float clan,suma,eps,i=1.0;
   scanf("%f",&eps);   clan=suma=1.0;
   do
     { clan=znak/(2*i+1); suma+=clan; znak=-znak; i++; }
   while(1/(2*i+1)>=eps);
   printf("Broj Pi=%f\n",4*suma);
}
```

## For naredba u C

U leziku C, naredba *for* je povezana sa *while* naredbom. Preciznije, konstrukcija

```
for(pocetak; uslov; korekcija)
     operator
sledeća naredba
```

ekvivalentna je sa

```
pocetak;
while(uslov)
  { operator
    korekcija
  }
```

pod uslovom da uslov nije prazan.

Operator koji predstavlja telo ciklusa može biti prost ili složen.

*For* ciklus oblika

```
for(; uslov ;) operator
```

ekvivalentan je *while* ciklusu

```
while(uslov) operator
```

Takođe, ekvivalentni su sledeći beskonačni ciklusi:

```
for(;;) operator
```

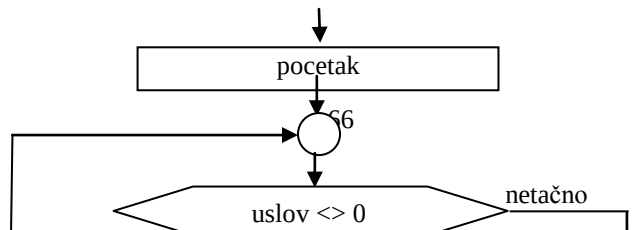
i

```
while(1) operator.
```

For naredba oblika

```
for  (pocetak;   uslov;   korekcija) operator
```

može se predstaviti sledećim dijagramom toka



Izrazi u zaglavlju petlje mogu da budu i izostavljeni. Na primer, petlja bez drugog izraza izvršava se kao da je njegova vrednost *true*, što znači kao beskonačna petlja. Ako su izostavljeni prvi i treći izraz, telo ciklusa je prazno.

U jeziku C se za definisanje petlje sa unapred definisanim brojem izvršenja tela petlje može koristiti *for* naredba. Međutim, for naredba u C je mnogo fleksibilnija od odgovarajućih naredbi u drugim jezicima. Naime, izrazi u zaglavlju petlje mogu da objedine više naredbi. To znači da se u okviru petlje može istovremeno definisati više uslova upravljanja vezanih za različite promenljive koje čak mogu da budu i različitih tipova. Razmotrimo sledeći primer.

## Primeri

**Primer.** Suma prvih *n* prirodnih brojeva se može izračunati na više različitih načina.

```
1. sum=0; for(i=1; i<=n; ++i)  sum += i;
2. sum=0; i=1; for(; i<=n;) sum += i++;
3. for(sum=0, i=1; i<=n; sum += i, i++);
```

**Primer.** Faktorijel prirodnog broja urađen na dva slična načina.

```
void main()
 { int i,n;
    long fak=1;
    printf("\n Unesite broj"); scanf("%d",&n);
    for(i=1; i<=n; ++i) fak *=i;
    printf("%d! = %ld \n",n,fak);
 }
```

```
 void main()
 { int i,n;
    long fak=1;
    printf("\n Unesite broj"); scanf(" %d",&n);
    for(fak=1,i=1; i<=n; fak*=i,++i);
    printf("%d! = %ld \n",n,fak);
 }
```

**Primer.** Izračunati 1! + 2! +…+ *n*!.

```
void main()
 { int i,n; long fakt,s;
    scanf("%d",&n);
    for(s=0, fakt=1, i=1; i<=n; fakt*=i, s+=fakt, i++);
    printf("Suma=%ld\n",s);
 }
```

Isti izraz se može izračunati na sledeći način

```
void main()
 { int n; long s;
```

```
    scanf("%d",&n);
    for(s=0; n>0; s=s*n+n--);
    printf("Suma=%ld\n",s);
 }
```

**Primer.** Izračunati sve trocifrene *Armstrongove* brojeve.

```
void main()
 { int a,b,c, i;
    for(i=100;i<=999;i++)
     { a=i%10;    b=i%100/10;    c=i/100;
        if(a*a*a+b*b*b+c*c*c==i)  printf("%d\n",i);
     }
 }
```

**Primer.** Program za određivanje savršenih brojeva do zadatog prirodnog broja. *Savršen* broj je jednak sumi svojih delitelja.

```
void main()
 { int i,m,n,del,s;
    scanf("%d",&m)
    for(i=2;i<=m;i++)
     { n=i; s=1;
        for(del=2; del<=i/2; del++) if(n%del==0) s+=del;
        if(i==s) printf("%d\",i);
     }
 }
```

**Primer.** Tablica ASCII kodova.

```
void main()
 { signed char c;  unsigned char d;
   for(c=-128; c<=127; c++) printf("c = %d %c\n",c,c);
   for(d=0; d<=255; d++) printf("d = %d %c\n",d,d);
 }
```

**Primer. (Pascal)** Izračunati

$$\cos(x) \approx \sum_{k=0}^{\infty} \frac{x^{2k}}{(2k)!}$$

sa tačnošću ε. Sumiranje prekinuti kada bude ispunjen uslov

$$|(x^{2k})((2k)!)| < \varepsilon .$$

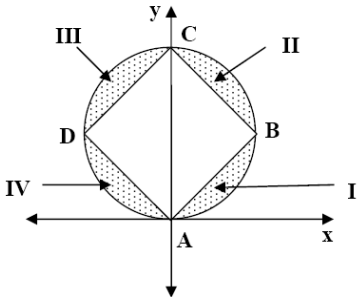
```
program suma;
  var x,eps,a,s:real;
      k:integer;
  begin
    writeln('Unesite promenljivu x i tacnost  '); readln(x,eps);
    k:=0; a:=1; s:=a;
    while abs(a)>=eps do
      begin
        a:=sqr(x)/((2*k+1)*(2*k+2))*a;   s:=s+a;   k:=k+1;
      end;
    writeln('Suma =  ',s:0:6);
  end.
```

**Primer.** U ravni je dato $N$ tačaka svojim koordinatama $(x, y)$. Napisati program na programskom jeziku C, koji na svom izlazu daje ukupan broj tačaka koje pripadaju oblastima I, II, III odnosno IV, i broj tačaka koje ne pripadju ni jednoj od naznačenih oblasti. Oblasti su definisane kružnicom i pravama kao na slici, a njihove jednačine glase:

$$K: \; x^2 + (y-1)^2 = 1$$

AB: $y = x$, BC: $y = -x + 2$, CD: $y = x + 2$, DA: $y = -x$.

Broj tačaka kao i njihove koordinate uneti sa tastature na početku programa.



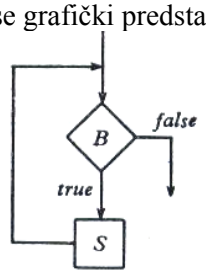

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
void main()
{ int n, i;
  float x, y;
  // Brojaci tacaka u odgovarajucim oblastima.
  int br1 = 0, br2 = 0, br3 = 0, br4 = 0, brOstalo = 0 ;
  // Promenljie koje definisu polozaje tacaka
  int unutarK, ispodAB, iznadBC, iznadCD, ispodDA;
  printf("Unesite broj tacaka\n"); scanf("%d", &n);
  for (i = 0; i < n; i++)
  { printf("Unesite kordinate tacke %d\n", i + 1); scanf("%f %f", &x, &y);
    unutarK = (x*x + (y-1)*(y-1)) < 1;
    ispodAB = y < x;
    iznadBC = y > -x + 2;
    iznadCD = y > x + 2;
    ispodDA = y < -x;
    if (unutarK)
      { // Tacka je unutar kruznice
      if (ispodAB)  br1++;
      else if (iznadBC) br2++;
      else if (iznadCD) br3++;
      else if (ispodDA) br4++;
      }
  }
  brOstalo = n - br1 - br2 - br3 - br4;
  printf("Brojevi tacaka u oblsatima I, II, III, IV respektivno iznose:
              %d %d %d %d\n",    br1, br2, br3, br4);
  printf("Broj tacaka koje ne pripadaju ni jednoj oblasti iznosi: %d\n",
brOstalo);
}
```

# 4.6. Formalizacija repetitivnih iskaza

Neka je *B* logički izraz, a S iskaz. Tada:

```
while B do S                                                    (1)
```

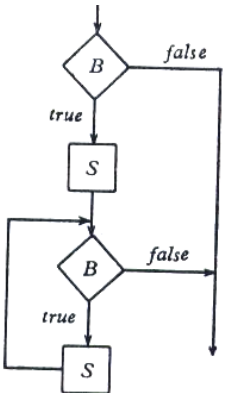
označava ponavljanje radnje definisane iskazom *S* sve dok izračunavanje izraza *B* daje vrednost *true*. Ako je vrednost izraza *B* jednaka *false* na samom početku procesa obavljanja radnje definisane sa (1), iskaz *S* neće biti uopšte izvršen. Iskaz (1) se grafički predstavlja na sledeći način:




(2)

Efekat iskaza `while B do S` može se rigorozno definisati tako što se kaže da je on ekvivalentan efektu iskaza:

```
if B then
   begin
      S;
      while B do S                                    (3)
   end.
```




(4)

Ovakva definicija znači da dijagrami (2) i (4) opisuju ekvivalentne procese izvršenja (radnje).

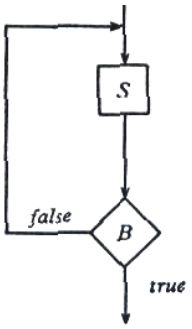
Primer `while B do S` iskaza dat je u programu (5), koji izračunava sumu $h(n) = 1+1/2+ \ldots +1/n$.

```
var n : integer; h : real;
h :=0;
while n>0 do                                          (5)
  begin
    h := h+1/n; n := n−1
  end.
```

Drugi oblik repetitivnog iskaza je:

```
repeat S until B,                                     (6)
```

gde je *S* niz iskaza a *B* logički izraz. Iskaz (6) definiše sledeću radnju: izvršava se iskaz *S*, a zatim se izračunava izraz *B.* Ako je rezultat izračunavanja izraza *B* jednak *false*, ponovo se izvršava iskaz *S*, i tako dalje. Proces ponavljanja radnje definisane sa *S* završava se kada izračunavanje izraza *B* da *true.* Struktura ovako definisanog repetitivnog iskaza predstavlja se sledećim dijagramom:



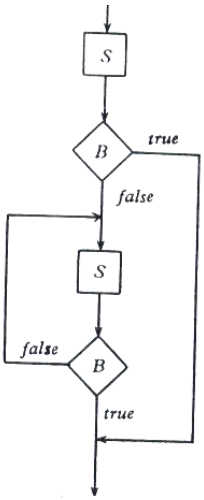

(7)

Poređenjem (2) i (7) zaključujemo da se u (7) iskaz *S* mora izvršiti bar jednom, dok se u (2) on ne mora uopšte izvršiti. Kako se u prvom slučaju *B* izračunava pre svakog izvršenja iskaza *S*, a u drugom nakon svakog izvršenja iskaza *S*, razlozi efikasnosti nalažu da *B* bude u što je moguće prostijoj formi.

Rigorozna definicija radnje određene iskazom `repeat S until B` daje se na taj način što se kaže da je ona ekvivalentna radnji koju određuje iskaz:

```
begin
  S;
  if ¬B then
     repeat S until B
```

```
end.                                                    (8)
```




(9)

Ovo znači da dijagrami (7) i (9) opisuju ekvivalentne radnje. Kako se u (2) i (7) pojavljuju zatvorene putanje ili petlje, oba ova iskaza se često i nazivaju petljama. U njima se *S* se naziva telom petlje.

Primer `repeat` iskaza dat je u programu (10), koji izračunava sumu $h(n)=1+1/2+\ldots+1/n$:

```
var n: integer; h: real;
h:=0;
repeat
   h:= h+1/n;  n:=n-1                                   (10)
until not(n>0).
```

Fundamentalni problem vezan za repetitivnu kompoziciju iskaza ilustrovaćemo primerima (5) i (10). Proces izvršenja iskaza (10) je konačan samo ako u početku važi $n>0$. Za $n<0$ (10) predstavlja beskonačnu petlju. S druge strane, program (5) se ponaša korektno, čak i ako je $n<0$, tj. proces njegovog izvršavanja je konačan. Vidimo, dakle, da pogrešno koncipiran repetitivni iskaz može opisivati radnju beskonačnog vremenskog trajanja.

# 4.7. Nasilni prekidi ciklusa

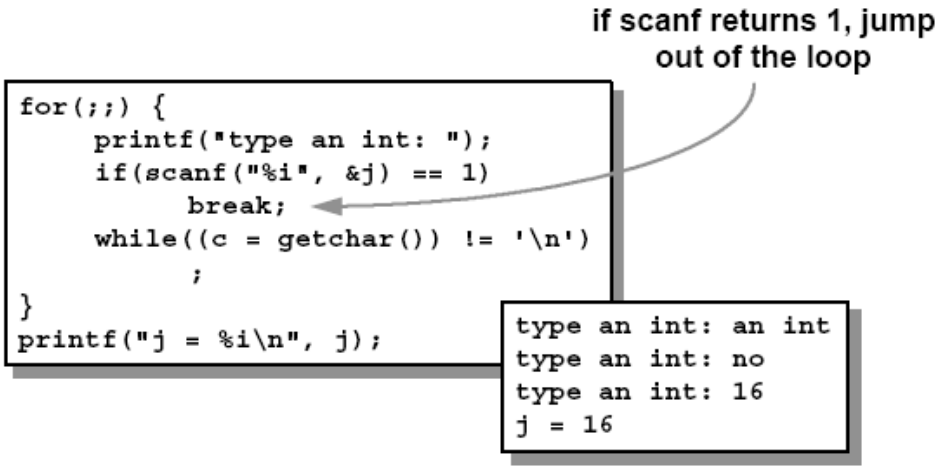
Petlje razmatrane u prethodnim primerima imaju zajedničku karakteristiku da se iz petlje izlazi ili na kraju, kada se završi sekvenca naredbi koja čini telo petlje, ili na samom početku pre nego i započne ta sekvenca naredbi, zavisno od vrste petlje i ispunjenja uslova. U takvim slučajevima, petlja se završava na osnovu izlaznog kriterijuma, koji je postavljen kao preduslov ili postuslov. Međutim, u određenim slučajevima pogodno je da programer sam odredi mesto izlaska iz petlje. U novijim programskim jezicima, obično postoji jednostavan mehanizam za uslovni ili bezuslovni izlazak iz petlje. U programskim jezicima C i Pascal takva naredba se naziva *break*.

U programskom jeziku C za bezuslovni izlazak iz najbliže petlje koristi se naredba *break*. Tada je njeno značenjem slično kao i kod *switch* naredbe.

**Primer**. Naredba *break* za bezuslovni izlazak iz petlje u C.

```
sum=0;
while (sum < 2000)
{  scanf("%d", &podatak);
   if(podatak < 0) break;
   sum = sum + podatak;
}
```

U svakom prolazu kroz petlju u ovom primeru učitava se po jedan podatak i dodaje sumi. Zaglavljem petlje je definisano da se sumiranje izvršava sve dok se sumiranjem ne dostigne vrednost 2000. Međutim, ukoliko se učita negativni podatak, petlja se prekida i program se nastavlja sa prvom naredbom iza petlje.

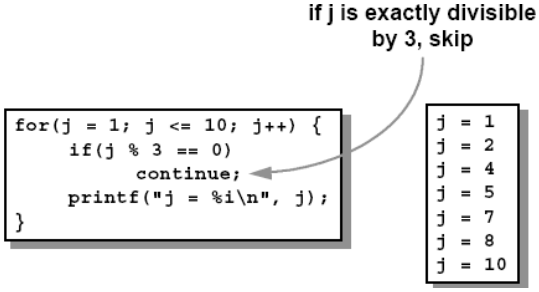



U jeziku C postoji i naredba *continue* koja se takođe koristi za upravljanje tokom izvršenja petlje. Ovom naredbom se samo preskače ostatak petlje i program nastavlja novim prolazom kroz petlju počevši od prve naredbe petlje. Efekat te naredbe opisan je u narednom primeru.

**Primer.** Efekat naredbe *continue* u C.

```
sum=0;
while (sum < 2000)
 { scanf("%d", &podatak);
   if (podatak < 0) continue;
   sum = sum + podatak;
 }
```

U ovom slučaju, kada se učita negativni podatak preskače se naredba za modifikovanje sume i program nastavlja ponovnim prolaskom kroz petlju. To znači da će u ovom slučaju petlja uvek da se završi kada se dostigne vrednost *sum* >= 2000, što u prethodnom primeru nije uvek bio slučaj.




# 4.8. Naredbe za bezuslovno grananje

Za bezuslovno grananje u programu koristi se *GOTO* naredba. U suštini, to je jedna od najmoćnijih naredbi koja u sprezi sa naredbom selekcije omogućava da se implementiraju i najsloženiji algoritmi. Problem je jedino u tome, što tako definisan kôd može da bude veoma nepregledan, a takvo programiranje podložno greškama.

Po konceptu strukturnog programiranja *GOTO* naredba je izbačena iz upotrebe, jer se nastoji da se program definiše umetanjem struktura jednu u drugu i da pri tome sve strukture imaju jedan ulaz i jedan izlaz. Postoji nekoliko jezika u kojima *GOTO* naredba uopšte ne postoji, na primer Modula 2. Međutim u mnogim jezicima koji podržavaju koncept strukturnog programiranja naredba *GOTO* je zadržana i ostaje pogodno sredstvo za rešavanje mnogih problema.

## 4.8.1. Oznake (labele)

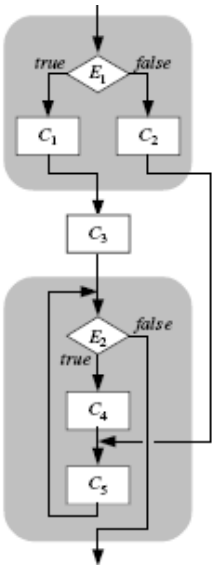
Skok (***jump***) prenosi kontrolu na specificiranu tačku u programu. Tipična forma bezuslovnog skoka je izraz ‘‘**goto** *L*;’’, čime se kontrola toka programa direktno prenosi na tačku označenu sa *L*, što se naziva obeležje (***label***).

**Primer.** Sledeći fragment programa (u C-like jeziku) sadrži bezuslovni skok:

```
if (E1)
  C1
else {
  C2
  goto X;
}
C3
while (E2) {
  C4
  X: C5
}
```

Ovde obeležje *X* označava partikularnu tačku programa, preciznije, početak komande *C*5. Na taj način, skok ‘‘**goto** X;’’ prenosi kontrolu na početak komande *C*5.

Sledeća slika prikazuje dijagram toka koji odgovara ovom fragmentu programa. Napomenimo da jedna komanda *if* i komanda *while* imaju prepoznatljive poddijagrame, koji su označeni. Fragment programa, kao celina ima jedan ulaz i jedan izlaz, ali komanda *if* ima dva izlaza, dok *while* komanda ima dva ulaza.




Neograničeni skokovi omogućavaju komande koje imaju višestruke ulaze i višestruke izlaze. Oni imaju tendenciju da produkuju ‘‘špagetti’’ kôd, tako nazvan zato što je njihov dijagram toka zamršen.

‘‘Špagetti’’ kôd ima tendenciju da bude teško razumljiv. Većina glavnih programskih jezika podržava skokove, ali su realno zastareli u modenim jezicima. Čak i u jezicima koji podržavaju skokove, njihova upotreba je ograničena restrikcijom dometa svakog obeležja: skok ‘‘**goto** *L*;’’ je legalan samo unutar dosega *L*. U C, na primer, doseg svakog obeležja je najmanje obuhvaćen blok komandom (‘‘{ . . . }’’). Prema tome, mogući su skokovi unutar blok komande, ili iz jedne blok komande izvan neke ograđene blok komande; međutim, nemoguć je skok unutar neke blok komande od spolja, niti iz jednog tela funkcije u drugo.

C skokovi nisu ograničeni da spreče ‘‘špagetti’’ kodovanje.

Skok unutar blok komande je relativno jednostavan, dok je skok izvan blok naredbe komplikovaniji. Takav skok mora da uništi lokalne promenljive bloka pre nego što se transfer prenese na destinaciju skoka.

**Primer**. Skok izvan blok naredbe. Uočimo sledeći (veštački) C kod:

```
#include<stdio.h>
void main()
{ char stop = '.',ch='!';
  do { char ch;
```

```
        ch = getchar();
        if (ch == EOF) goto X;
        putchar(ch);
   } while (ch != stop);
   printf("done");
   X: printf("%c\n",ch);
}
```

Skok ‘‘**goto** X;’’ prenosi kontrolu izvan blok komande {. . .}. Prema tome, on takođe uništava lokalnu promenljivu *ch* iz bloka. Po izlasku iz ciklusa, prikazuje se vrednost ‘!’ kao vrednost promenljive *ch*.

Skok izvan tela procedure je još komplikovaniji. Takav skok mora da uništi lokalne promenljive i da terminira aktivaciju procedure pre prenosa kontrole na takvu destinaciju. Velike komplikacije nastaju kada je destinacija skoka unutar tela rekurzivne procedure: koje rekurzivne aktivacije se terminiraju kada se skok izvrši?

Prema tome, površno su skokovi veoma jednostavni, dok u suštini oni uvode neželjenu kompleksnost u semantiku jezika visokog nivoa.

Za označavanje naredbi, na koje se pod dejstvom GOTO naredbe prenosi upravljanje, u programskim jezicima se koristi koncept oznaka (labela). Iskaz *Goto* je prost iskaz koji ukazuje na to da dalju obradu treba nastaviti u drugom delu programskog teksta, to jest sa mesta na kome je labela.

U nekim jezicima (C-u, Algol-u 60 na primer) koriste se identifikatori kao oznake. U drugim (FORTRAN i Pascal na primer) to su neoznačene celobrojne konstante. U Adi su oznake takođe identifikatori, ali obuhvaćeni zagradama << i >>, dok se za razdvajanje oznaka od naredbi na koje se odnose u mnogim jezicima koristi dvotačka (:). Evo nekih primera:

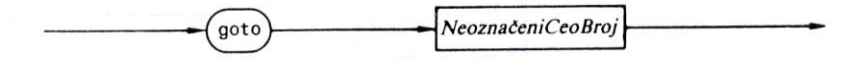
Primer naredbe GOTO za skok na označenu naredbu u Adi.

```
                goto  KRAJ;
                …
<<KRAJ>>  SUM   := SUM + PODATAK;
```

Isti primer napisan u C-u bio bi sledećeg oblika:

```
          goto  kraj;
          …
kraj:     sum := sum + podatak;
```

U svim programskim jezicima obično su definisana određena pravila koja ograničavaju korišćenje naredbe *GOTO*. Na primer, u Pascal-u oznake moraju eksplicitno da budu opisane u odeljku deklaracije naredbi. One se ne mogu prenositi kao argumenti potprograma niti modifikovati. Kao deo *GOTO* naredbe može da bude samo neoznačena konstanta, ali ne i izraz ili promenljiva koja dobija vrednost oznake.




Svaka labela poseduje sledeće osobine:

1. mora se pojaviti u odeljku deklaracija labela, pre svog pojavljivanja u bloku,
2. mora prethoditi jednom i samo jednom iskazu koji se pojavljuje u iskaznom delu bloka.
3. ima oblast važenja u celokupnom tekstu tog bloka, isključujući sve ugnježdene blokove koji ponovo deklarišu tu labelu.

**Primer.** U beskonačnom ciklusu unositi dva realna broja i računati njihov proizvod.

```
program mnozi;
  uses crt;
  label 1;
  var a,b:real;
  begin
   1:
    write(' Prvi broj? '); readln(a); write(' Drugi broj? '); readln(b);
    writeln(' Njihov proizvod je '); writeln(a*b:20:0);
```

```
    goto 1;
  end.
```

**Primer.** Primeri upotrebe *Goto* naredbe (programski deo):

```
  label 1; { blok A }
  ...
  procedure B; { blok B }
     label 3, 5;
     begin
            goto 3 ;
        3 : Writeln('Dobar dan');
        5 : if P then
               begin S; goto 5 end ; { while P do S }
            goto 1 ;
            { ovo uzrokuje prevremen zavrsetak aktivacije procedure B}
            Writeln('Doviđenja')
      end; { blok B }
   begin
         B;
     1: Writeln('program')
         { "goto 3" nije dozvoljen u bloku A )
   end { blok A }
```

Skokovi izvana u strukturirani iskaz nisu dozvoljeni. Stoga su sledeći primeri nepravilni.

Primeri sintaksno nepravilnih programa:

```
a) for I := 1 to 10 do
     begin
         S1;
      3: S2
     end ;
   goto 3

b) if B then goto 3;
   ...
   if B1 then 3 : S

c) procedure P;
     procedure Q;
        begin ...
            3 :S
        end ;
   begin ...
      goto 3
   end.
```

Iskaz *Goto* treba sačuvati za neuobičajene ill retke situacije gde je potrebno razbiti prirodnu strukturu algoritma. Dobro je pravilo da bi trebalo izbegavati upotrebu skokova pri izražavanju pravilnih ponavljanja i uslovnih izvršavanja iskaza, jer takvi skokovi uništavaju odraz strukture obrade u tekstualnim (statičkim) strukturama programa. Štaviše, nedostatak slaganja između tekstualne i obradne (tj. statičke i dinamičke) strukture poguban je po jasnoću progama i čini proveru ispravnosti programa mnogo težom. Postojanje iskaza *Goto* u Pascal programu često ukazuje na to da programer još nije naučio "misliti" u Pascalu (jer je u nekim drugim programskim jezicima *Goto* neophodna konstrukcija).

# 5. POTPROGRAMI

U programiranju se često javlja potreba da se neki deo programa više puta ponovi u jednom programu. Takođe, često puta se ista programska celina javlja u više različitih programa. Ovakva nepotrebna ponavljanja se izbegavaju pomoću potprograma.

Potprogrami se uvek koriste sa idejom da se izbegne ponavljanje istih sekvenci naredbi u slučaju kada u programu za to postoji potreba. Međutim, potprogrami su mnogo značajniji kao sredstvo za struktuiranje programa. Pri razradi programa metodom dekompozicije odozgo naniže, svaka pojedinačna aktivnost visokog nivoa može da se zameni pozivom potprograma. U svakoj sledećoj etapi ide se sve niže i niže u tom procesu dekompozicije sve dok se rešenje kompleksnog programa ne razbije na jednostavne procedure koje se konačno implementiraju. Kako napredujemo u veštini računarskog programiranja, tako programe pišemo u nizu koraka preciziranja. Pri svakom koraku razbijamo naš zadatak u više podzadataka, definišući tako više parcijalnih programa. Koncepti procedure i funkcije omogućavaju da izložimo podzadatke kao konkretne potprograme.

Može se reći da su potprogrami osnovno sredstvo apstrakcije u programiranju. Potprogramom se definiše skup izlaznih veličina u funkciji ulaznih veličina. Realizacija tih zavisnosti definisana je u potprogramu.

Potprogrami se definišu kao programske celine koje se zatim po potrebi pozivaju i realizuju, bilo u okviru istog programa u kome su i definisani, bilo u drugim programima.

Istaknimo neke opšte karakteristike potprograma kakvi se najčešće sreću u savremenim programskim jezicima.

(1) Svaki potprogram ima jednu ulaznu tačku. Izuzetak od ovog pravila postoji jedino u FORTRAN-u, gde je moguće definisati više različitih ulaza u potprogram.

(2) Program koji poziva potprogram prekida svoju aktivnost sve dok se ne završi pozvani potprogram. To znači da se u jednom trenutku izvršava samo jedan potprogram.

(3) Po završetku pozvanog potprograma upravljanje se vraća programu koji ga je pozvao, i to na naredbu koja sledi neposredno iza poziva potprograma.

Potprogram karakterišu sledeći osnovni elementi:

- ime potprograma,
- lista imena i tipova argumenata (fiktivni argumenti),
- lokalni parametri potprograma,
- telo potprograma,
- sredina (okruženje) u kojoj potprogram definisan.

**Ime potprograma** je uvedena reč, odnosno identifikator, kojom se potprogram imenuje i kojim se vrši njegovo pozivanje.

**Lista fiktivnih parametara:** u nekim programskim jezicima ova lista može da sadrži samo imena fiktivnih argumenata, čime je određen njihov broj i redosled navođenja, Kod novijih programskih jezika u listi argumenata se navode i atributi o tipu argumenata kao i o načinu prenošenja u potprogram. Fiktivnim argumentima definiše se skup ulazno-izlaznih veličina potprograma. Kod poziva potprograma se fiktivni argumenti (parametri) zamenjuju stvarnim argumentima. Fiktivni i stvarni argumenti moraju da se slažu po tipu, dimenzijama i redosledu navođenja u listi argumenata.

**Lokalni parametri.** U samoj proceduri mogu da budu definisani lokalni elementi procedure i to promenljive, konstante, oznake, a u nekim jezicima i drugi potprogrami.

**Telo potprograma** čini sekvenca naredbi koja se izvršava nakon poziva potprograma. U telu potprograma se realizuje kôd potprograma.

**Okolinu (okruženje) potprograma** predstavljaju vrednosti globalnih promenljivih okruženja u kome je potprogram definisan i koje se po mehanizmu globalnih promenljivih mogu koristiti u potprogramu.

U programskim jezicima se obično sreću dva tipa potprograma:
- funkcijski potprogrami (**funkcije**),
- potprogrami opšteg tipa (**procedure**).

# 5.1. Funkcije

Funkcijski potprogrami po konceptu odgovaraju matematičkim funkcijama. Lista argumenata u deklaraciji ovih potprograma se obično sastoji se samo od ulaznih argumenata, na osnovu kojih se u telu potprograma izračunava vrednost koja se prenosi u pozivajuću programsku jedinicu. Ulazne vrednosti se u funkciji ne menjaju.

U jezicima Pascal i FORTRAN se za prenošenje vrednosti iz potprograma u pozivajuću jedinicu koristi promenljiva koja se identifikuje sa imenom funkcije. U telu takvih funkcija mora da postoji bar jedna naredba dodele kojom se rezultat dodeljuje imenu funkcije. Kaže se da funkcija prenosi vrednost u glavni program preko svog imena.

U novijim programskim jezicima obično postoji naredba *return* kojom se eksplicitno navodi vrednost koja se iz potprograma prenosi u glavni program. Naredba return se koristi u jeziku C. U jeziku C se ne koristi reč function, jer su svi potprogrami funkcije. Ako funkcija vraća jedan rezultat naredbom *return* ispred njenog imena se piše tip rezultata, inače se piše službena reč *void*.

Funkcijski potprogrami se pozivaju slično matematičkim funkcijama, tako što se u okviru izraza navede ime funkcije sa imenima stvarnih parametara umesto fiktivnih. Koristeći gore navedene primere, suma prvih 999 celih brojeva se može izračunati pozivom funkcije SUM na sledeći način:

```
SUMA := SUM(999)        (Pascal)
```
ili
```
SUMA = SUM(999)         (C, FORTRAN).
```

Funkcija se koristi u slučaju kada se vraća jedna veličina u rezultujuću programsku jedinicu.

## 5.1.1. Poziv i definicija funkcija u C

Program se sastoji iz jedne ili više funkcija, pri čemu se jedna od njih naziva *main*(). Kada programska kontrola naiđe na ime funkcije, tada se poziva ta funkcija. To znači da se programska kontrola prenosi na pozvanu funkciju. Kada se funkcija završi upravljanje se prenosi u pozivajuću programsku jedinicu. Izvršenje programa počinje sa *main*().

Funkcije se dele u dve grupe: funkcije koje vraćaju vrednost u pozivajuću programsku jedinicu i funkcije koje ne vraćaju vrednost.

Za svaku funckiju moraju se definisati sledeći elementi:
- tip vrednosti koju funkcija vraća,
- ime funkcije,
- lista formalnih parametara,
- deklaracija lokalnih promenljivih,
- telo funkcije.

Opšta forma definicije funkcija je:

```
tip ime(lista_parametara)
 {   deklaracije lokalnih promenljivih
     operator1
```

```
    …
    operatorN
}
```

Deo definicije koji se nalazi pre prve zagrade '{' naziva se zaglavlje funkcijske definicije, a sve između velikih zagrada se naziva telo funkcijske definicije.

Tip funkcije zavisi od tipa vrednosti koja se vraća. Takođe, koristi se službena reč *void* ako funkcija ne vraća vrednost. Ako je tip rezultata izostavljen podrazumeva se *int*.

Parametri su po sintaksi identifikatori, i mogu se koristiti u telu funkcije (formalni parametri). Navođenje formalnih parametara nije obavezno, što znači da funkcija može da bude bez formalnih parametara.

Poziv funkcije koja daje rezultat realizuje se izrazom

ime(spisak_stvarnih_parametara)

koji predstavlja element drugih izraza. U slučaju da funkcija ne vraća rezultat, dopisuje znak `;', čime poziv funkcije postaje operator:

ime(spisak_stvarnih_parametara);

Između formalnih i stvarnih parametara mora da se uspostavi određena korespodencija po broju, tipu i redosledu.

Kada se formalni parametri definišu u zaglavlju funkcije, tada se za svaki parametar posebno navodi tip. Tada je zaglavlje funkcije sledećeg oblika:

tip ime(tip $param_1$,...,tip $param_n$)

U programskom jeziku C, formalni parametri se mogu deklaristati ispod zaglavlja funkcije. Tada se svi parametri istog tipa mogu deklarisati odjednom, koristeći jedinstveno ime tipa.

## Return naredba

Među operatorima u telu funkcije može se nalaziti operator *return*, koji predstavlja operator povratka u pozivajuću funkciju. Ako se operator *return* ne navede, funkcija se završava izvršenjem poslednjeg operatora u svom telu, i ne vraća rezultat.

Naredba *return* se koristi iz dva razloga:

- Kada se izvrši naredba return, kontrola toka programa se vraća u pozivajuću jedinicu.
- Osim toga, ako neki izraz sledi iza ključne reči *return*, tj. ako se naredba *return* koristi u obliku `return izraz` ili `return(izraz)`, tada se vrednost izraza *izraz* vraća u pozivajuću programsku jedinicu. Ova vrednost mora da bude u skladu sa tipom funkcije u zaglavlju funkcijske definicije.

Naredba *return* u jeziku C ima jednu od sledeće dve forme:

```
return;
```

ili

```
return(izraz);   odnosno   return izraz;
```

Na primer, može se pisati

```
return(3);
return(a+b);
```

Dobra je praksa da se izraz čija se vrednost vraća piše između zagrada.

Opšti oblik definicije funkcije:

```
<povratni_tip|void> ime(<tip> par1, <tip> par2, …, <tip> parN)
{ <lokalne promenljive>
   <blok instrukcija>
   Return <povratna vrednost> // Izostavlja se ako je funkcija tipa void
}
```

**Primer.** Uzmimo kao primer funkciju za izračunavanje sume prvih *n* prirodnih brojeva. Radi poređenja isti potprogram napisan je u različitim programskim jezicima:

**Pascal**

```
function SUM (n: integer) : integer;
  var   i, POM : integer; { Opis lokalnih promenljivih }
 begin  { Telo funkcije }
   POM := 0:
   for i := 1 to n do POM := POM + i;
   SUM := POM {Vrednost koja se prenosi u glavni program}
end;
```

**FORTRAN**

```
      INTEGER FUNCTION SUM(N)
        INTEGER N
        DO 10 I = 1, N
10          SUM = SUM + I
        END

      PROGRAM SUMA
      INTEGER S,N
      READ(*,10)N
10    FORMAT(I2)
      S=SUM(N)
      WRITE(*,20)S
20    FORMAT(I4)
```

**C**

```
#include<stdio.h>
int SUM (int n)
{  int POM = 0;
   for(int i = 1; i <= n; i++)POM = POM + i ;
   return POM;
}
void main()
{ int n;
  scanf("%d",&n);     printf("%d\n",SUM(n));
}
```

## Prototip funkcije

U tradicionalnom stilu C jezika, funkcija se može koristiti pre nego što je definisana. Definicija funkcije može da sledi u istom ili u nekom drugom fajlu ili iz biblioteke. Nepoznavanje broja argumenata odnosno tipa funkcije može da uzrokuje greške. Ovo se može sprečiti prototipom funkcije, kojim se eksplicitno ukazuje tip i broj parametara koje funkcija zahteva i tip rezultujućeg izraza. Opšta forma prototipa funkcije je sledeća:

```
tip ime (lista_tipova_parametara);
```

U ovom izrazu je lista_tipova_parametara lista tipova odvojenih zarezima. Kada je funkcija pozvana, argumenti se konvertuju, ako je moguće, u navedene tipove.

Na primer, sintaksno su ispravni sledeći prototipovi:

```
void poruka1(int);
int min(int, int);
```

Takođe, prototip funkcije može da uključi i imena parametara, čime se može obezbediti dodatna dokumentacija za čitaoca.

Na primer, možemo pisati

```
double max(duble x, double y);
```

## Primeri

**Primer.** Izračunati minimum dva cela broja. Funkcija *min*(*x*,*y*) vraća rezultat tipa *int*, te se eksplicitno navođenje tipa rezultata može izostaviti. U pozivajućoj funkciji (u ovom slučaju je to funkcija *main*()) naveden je prototip funkcije *min*.

```
#include<stdio.h>
void main()
   { int min(int,int);
     int j,k,m;
     printf("Unesi dva cela broja: ");scanf("%d%d",&j,&k);
     m=min(j,k); printf("\n%d je minimum za %d i %d\n\n", m,j,k);
   }
min(int x,int y)
  { if (x<y) return(x);    else return(y);   }
```

**Primer.** Minimum i maksimum slučajnih brojeva.

```
#include<stdio.h>
#include<stdlib.h>
void  main()
   { int n;
     void slucajni(int);
     printf("Koliko slucajnih brojeva?"); scanf("%d",&n);
     slucajni(n);
   }

void slucajni(int k)
  { int i,r,mini,maxi;
     int min(int, int), max(int, int);
     srand(10);
     r=mini=maxi=rand(); printf("%d\n",r);
     for(i=1;i<k;++i)
       { r=rand();printf("%d\n",r);
         mini=min(r,mini);maxi=max(r, maxi);
       }
     printf("Minimum: %7d\n Maximum:%7d\n", mini,maxi);
 }
int min(int i, int j)
  { if(i<j) return(i);   else return(j); }
int max(int i, int j)
  { if(i>j) return(i);   else return(j); }
```

**Primer.** Izračunavanje broja kombinacija *m*-te klase od *n* elemenata po formuli

$$\binom{n}{m} = \frac{n!}{m!(n-m)!}$$

**Pascal**

```
program fakt1;

  var m,n:integer;
      bk:longint;

  function fakt(k:integer):longint;
    var i:integer;     p:longint;
    begin
      p:=1;
      for i:=2 to k do p:=p*i;
      fakt:=p;
    end;

  begin
    readln(n,m);
    bk:=fakt(n) div fakt(m) div fakt(n-m);
    writeln('Broj kombinacija je ',bk);
```

```
  end.
```

**C**

```
#include<stdio.h>
void main()
   { int m,n;   long  bk;
      long fakt(int);
      scanf("%d%d",&n,&m);
      bk=fakt(n)/fakt(m)/fakt(n-m);
      printf("Broj kombinacija je %ld\n",bk);
    }

long fakt(int k)
  { int i;     long p=1;
     for(i=2; i<=k; i++) p*=i;
     return(p);
  }
```

Izračunavanje vrednosti *n*! se može izdvojiti u posebnom potprogramu. Tada se prethodno napisani potprogrami mogu napisati na sledeći način.

**Pascal (II način)**

```
program fakt2;

var m,n:integer;
    bk:longint;

function komb(p,q:integer):longint;
  function fakt(k:integer):longint;
    var i:integer;
        p:longint;
    begin  {fakt}
      p:=1;
      for i:=2 to k do p:=p*i;
      fakt:=p;
    end;
  begin {komb}
    komb:= fakt(p) div fakt(q) div fakt(p-q)
  end;

begin
  readln(n,m);
  bk:=komb(n,m);   writeln('Broj kombinacija je ',bk);
end.
```

**C (II način)**

```
#include<stdio.h>
void main()
   { int m,n;   long  bk;
     long komb(int, int);
      scanf("%d%d",&n,&m);
      bk=komb(n,m);  printf("Broj kombinacija je %ld\n",bk);
    }

long komb(int p, int q)
{ long fakt(int);
  return fakt(p)/fakt(q)/fakt(p-q);
}

long fakt(int k)
  { int i;     long p=1;
     for(i=2; i<=k; i++) p*=i;
     return(p);
  }
```

**Primer.** Izračunati

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots + (-1)^{n-1}\frac{x^{2n-1}}{(2n-1)!} + \ldots$$

Sumiranje prekinuti kada se ispuni uslov $|a/S| <= \varepsilon$, gde je $\varepsilon$ zadata tačnost, $S$ predstavlja tekuću vrednost sume, dok $a$ predstavlja vrednost člana koji se dodaje.

```
#include<stdio.h>
void main()
  { double x,eps;
    double sinus();     scanf("%lf%lf",&x,&eps);
    printf("Sinus je %lf\n", sinus(x,eps));
   }

 /* apsolutna vrednost */
double abs(x)  { return (x>0)?x:-x; }

/* sinus */
double sinus(double x, double eps)
 { double sum, clan;    int k;    clan=x; k=0;   sum=clan;
    while(abs(clan)>eps*abs(sum))
       { k+=1;   clan*=-(x*x)/(2*k*(2*k+1));   sum+=clan;  }
    return(sum);
 }
```

**Primer.** Svi prosti brojevi do datog prirodnog broja $n$.

```
#include<stdio.h>
#include <math.h>
int prost(int k)
  { int i=3;
      if(k%2==0) return(0);
      if(k==3)   return(1);
      while((k%i!=0) && (i<=sqrt(k)))  i+=2;
      if(k%i==0) return(0);     else return(1);
   }

void main()
  { int i,n;     int prost(int);
     printf("Dokle? "); scanf("%d", &n);
     for(i=3;  i<=n; i++) if(prost(i)==1)  printf("%d\n",i);
   }
```

**Primer.** Trocifreni brojevi jednaki sumi faktorijela svojih cifara

```
#include<stdio.h>
long fakt(int k)
  { long f=1,i=1;
     for(;i<=k;i++)  f*=i;
     return(f);
  }

void main()
   { long i;   unsigned a,b,c;
      for(i=100;i<=999;i++)
           { a=i%10;    b=i%100/10;    c=i/100;
             if(fakt(a)+fakt(b)+fakt(c)==i) printf("%d\n",i);
            }
   }
```

**Primer.** Unositi prirodne brojeve dok se ne unese vrednost <=0. Izračunati NZS unetih brojeva.

```
void main()
   {int k,ns,nzs=0;    scanf("%d", &k);
      if(k<=0) printf("%d\n",nzs);
```

```
      else
       { ns=nzs=k;         scanf("%d", &k);
         while(k>0)
             {nzs=funnzs(ns,k); ns=nzs; scanf("%d",&k); };
         printf("nzs = %d\n",nzs);
    } }
int funnzs(int a, int b)
    { int nz;
       if(a>b)nz=a;
       else nz=b;
       while( ((nz % a) !=0) || ((nz % b)!=0) ) nz++;
       return(nz);
    }
```

**Primer.** Dekadni broj iz intervala [0, 3000] prevesti u rimski brojni sistem.

```
#define hi 'M'
#define ps 'D'
#define st 'C'
#define pd 'L'
#define ds 'X'
#define pe 'V'
#define je 'I'

void rim(int);
void main()
  { int n;
      printf("Unesi  prirodan broj -> "); scanf("%d", &n);
      printf("\nOdgovarajuci rimski broj je: ");
      rim(n);      printf("\n");
   }

void rim(int n)
 { while(n>= 1000)    { printf("%c",hi); n-=1000; }
    if(n>=900) {printf("%c",st); printf("%c",hi); n-=900;}
    if(n>= 500)    { printf("%c",ps); n-=500; }
    if(n>=400)
          { printf("%c",st); printf("%c",ps); n-=400; }
    while(n>=100)      { printf("%c",st); n-=100; }
    if(n>=90)
        { printf("%c",ds); printf("%c",st); n-=90; }
    if(n>= 50)      { printf("%c",pd); n-=50; }
    if(n>=40)
          { printf("%c",ds); printf("%c",ps);  n-=40; }
    while(n>= 10)      { printf("%c",ds); n-=10; }
    if(n==9)
      { printf("%c",je);   printf("%c",ds); n-=9; }
    if(n>= 5)      {printf("%c",pe); n-=5; }
    if(n==4)    {printf("%c",je); printf("%c",pe); n-=4; }
    while(n>=1)      { printf("%c",je); n--; }  }
```

**Primer.** Najbliži prost broj datom prirodnom broju.

```
#include<stdio.h>
int prost(int n)
  { int i=2,prosti=1;
     while ((i<=(n/2))&&(prosti))  {prosti=(n%i)!=0; i++;}
     return(prosti); }

void main()
  {int m,n,i=1,prosti=1;   int prost(int n);
    printf("Broj za koji se racuna najblizi prost broj");
    scanf("%d",&m);
```

```
    if(m==1) printf("najblizi prost broj je 2\n");
    else if (prost(m))
    printf("najblizi prost broj je%d\n",m);
    else
      { while (i<=(m-1)&&(prosti))
          { if (prost(m+i))
              { prosti=0; printf("najblizi prost je%d\n",(m+i)); }
            if(prost(m-i) && (!prosti))
              {prosti=0; printf("najblizi prost je %d\n",(m-i)); }
            i++;
     }    }
    getch();
 }
```

## 5.1.2. Makroi u jeziku C

Makro je ime kome je pridružena tekstualna definicija. Postoje dve vrste makroa: makro kao objekat (koji nema parametre) i makro kao funkcija (koji ima parametre).

**Primer.** Primeri makroa kao objekata.

```
#define OPSEG "Greska: Ulazni parametar van opsega"
```

Makro kao funkcija se definiše izrazom oblika

#define ime(lista_identifikatora) tekst_zamene

Pri definiciji makroa definiše se lista njegovih formalnih parametara, dok se prilikom pozivanja makroa koriste stvarni parametri.

**Primer.** Makro kojim se ispituje da li je njegov argument paran broj.

```
#define NETACNO 0
#define TACNO 1
#define PARAN(broj) broj%2 ? NETACNO : TACNO
void main()
 { int n1=33;
   if(PARAN(n1)) printf("%d je paran\n",n1);
   else printf("%d je neparan\n",n1);
 }
```

U pozivu makroa *paran*(*a*+*b*) ispituje se parnost broja *a*+*b* pomoću izraza *a*+*b*%2 umesto izraza (*a*+*b*)%2. Da bi se ovakve greške izbegle, potrebno je da se formalni parametri pišu između zagrada. U našem slučaju, pravilna je sledeća definicija makroa *PARAN*:

```
#define PARAN(broj) ((broj)%2) ? NETACNO : TACNO
```

# 5.2. Procedure

Procedurama se definišu potprogrami opšteg tipa, kojima se može realizovati svaki algoritam koji săm za sebe ima određeni smisao. Procedurama se obično u glavni program prenosi jedan ili skup rezultata.

Opis potprograma opšteg tipa obično ima sledeću strukturu:

**procedure** IME_PROCEDURE (Lista argumenata)
  Opisi lokalnih promenljivlh;
  Telo procedure
**end;**

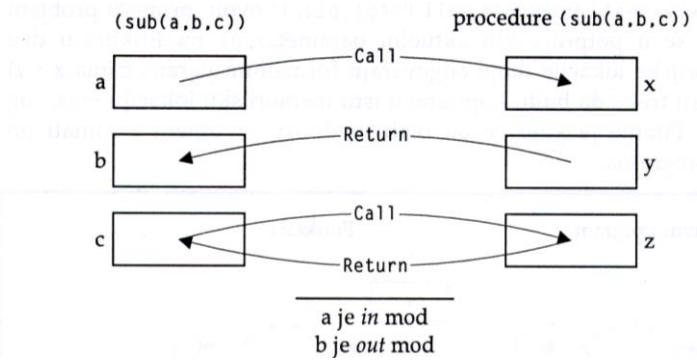
Opis počinje nekom ključnom reči kojom se eksplicitno navodi da definišemo proceduru. U mnogim programskim jezicima koristi se reč *procedure*, mada ima i izuzetaka, na primer u FORTRAN-u se koristi *SUBROUTINE*. Identifikator *IME_PROCEDURE* je uvedena reč kojom se imenuje potprogram. Ime potprograma se koristi za poziv potprograma. Za razliku od funkcija, lista

argumenata kod procedura može da sadrži ulazne, izlazne i ulazno-izlazne argumente. Obično se zahteva da se u listi argumenata navede njihov tip i način prenošenja vrednosti između glavnog programa i potprograma. U Adi na primer, argumenti mogu da budu ulazni, izlazni i ulazno-izlazni što se eksplicitno navodi rečima *in*, *out* i *inout* uz odgovarajuće argumente.

*Deklaracija procedure* služi da definiše programski deo i da ga pridruži identifikatoru, tako da se moze aktivirati *procedurnim iskazom.* Deklaracija ima isti oblik kao i program, osim što započinje zaglavljem procedure umesto zaglavljem programa.

# 5.3. Prenos argumenata

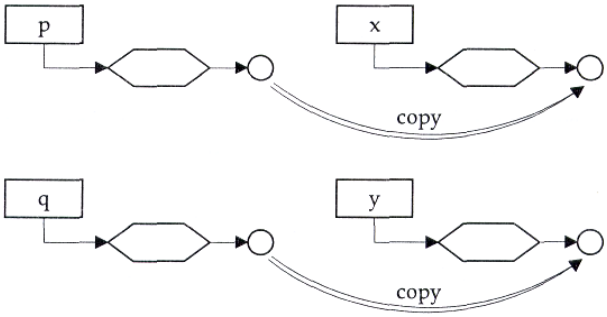
Funkcije i procedure se koriste tako što se tamo gde je potrebno u kôdu programa, pozivaju sa spiskom stvarnih argumenata. Pri tome stvarni argument zamenjuju fiktivne. Prenos argumenata između glavnog programa i potprograma može da se ostvari na više različitih načina. Semanticki posmatrano, to može da bude po jednom od tri semantička modela. Potprogram može da primi vrednost od odgovarajućeg stvarnog parametra (koji se stavlja na mesto ulaznog parametra), može da mu preda vrednost (ako se postavlja na mesto izlanog formalnog parametra) ill da primi vrednost i preda rezultat glavnom programu (kada se stvarni parametar postavlja na mesto ulazno-izlaznog parametra). Ova tri semantička modela nazivaju se *in*, *out* i *inout*, respektivno i prikazana su na slici. Sa druge strane, postoje dva konceptualna modela po kojima se ostvaruje prenos parametara između glavnog programa i potprograma. Po jednom, vrednosti parametara se fizički prebacuju (iz glavnog programa u potprogram ili u oba smera), a po drugom prenosi se samo informacija o tome gde se nalazi vrednost stvarnog parametra. Najčešće je to jednostavno pokazivač (pointer) na memorijsku lokaciju parametra.




## 5.3.1. Prenos po vrednosti (Call by Value)

Sastoji se u tome što se pri pozivu funkcije vrednosti stvarnih argumenata *kopiraju* u pomoćne memorijske lokacije fiktivnih argumenta koje su definisane u potprogramu. Funkcija preuzima kopije vrednosti stvarnih parametara, dok su originali yaštićeni od promena. Zbog toga se ova tehnika prenosa može koristiti samo za prenos ulaznih argumenata potprograma, a ne i za vraćanje rezultata glavnom programu. Razmotrimo situaciju koja nastane kada se potprogram *PP* sa argumentima *x* i *y* pozove sa stvarnim argumentima *p* i *q*. To znači imamo deklaraciju potprograma oblika `PP(x,y:integer)` i njegov poziv `PP(p,q)`. Za smeštaj vrednosti parametara *x* i *y* u potprogramu su rezervisane memorijske lokacije promenljivih *p* i *q*, čiji je sadržaj u trenutku prevođenja potprograma nedefinisan. U glavnom programu, sa druge strane, postoje memorijske lokacije za smeštaj stvarnih vrednosti *p* i *q*. Vrednosti ovih memorijskih lokacija moraju da budu definisane pre poziva potprograma. Kada se pozove potprogram, vrši se kopiranje vrednosti iz memorijskih lokacija *p* i *q* u lokacije fiktivnih argumenata *x* i *y* u potprogramu, što je ilustrovano na slici.

Glavni program Funkcija



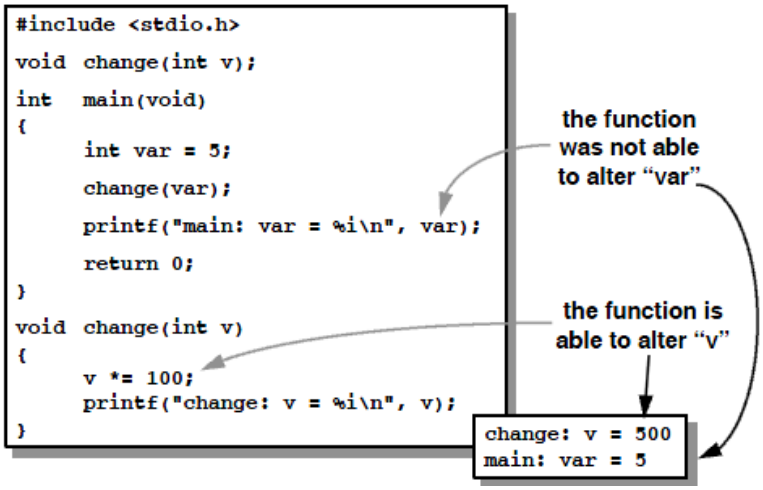

Osnovni nedostatak prenosa po vrednosti je u tome što se za formalne parametre potprograma rezerviše memorijski prostor. Uz to i sama operacija kopiranja vrednosti može da utiče na efikasnost programa. Ovi nedostaci posebno dolaze do izražaja kada se prenose strukture podataka, kao na primer veliki vektori i matrice.

Obično se u okviru programskih jezika implicitno ili eksplicitno navodi da se argumenti prenose po vrednosti. U Algolu se to postiže pomoću reči *value*, u Adi sa *in*, dok se u Pascal-u ovaj način podrazumeva kada nije navedeno *var* uz sekciju formalnih argumenata. U FORTRAN-u se argumenti prenose po vrednosti kada su stavljeni između kosih crta. U jeziku C se prenos po vrednosti podrazumeva za sve parametre koji nisu pokazivači.

## Prenos parametara po vrednosti u C

Funkcija se može pozvati stavljajući posle njenog imena odgovarajuću listu stvarnih argumenata između zagrada. Ovi argumenti se moraju slagati sa brojem i tipom formalnih parametara u definiciji funkcije. Svaki argument se izračunava, i njegova vrednost se uzima umesto formalnog parametra. Međutim, vrednost stvarnog parametra ostaje nepromenjena u pozivajućoj programskoj jedinici. Kada se koristi poziv po vrednosti, u momentu kada se koristi izraz kao stvarni argument funkcije, pravi se kopija njegove vrednosti, i ona se koristi u funkciji. Pretpostavimo da je *v* promenljiva, a *f*() funkcija. Tada poziv funkcije *f*(*v*) ne menja vrednost promenljive *v* u funkciji *f*, jer se koristi kopija vrednosti *v* u funkciji *f*().

Poziv po vrednosti je ilustrovan sledećim primerom.




**Primer.** Pronaći sve brojeve-blizance do zadatog broja *n*. Dva broja su blizanci ako su prosti i razlikuju se za 2.

```
#include<stdio.h>
#include<math.h>
void main()
  { int n,i;
    int prost(int);
     scanf("%d",&n);
     for (i=3;i<=n;i++)
      if ((prost(i)) && (prost(i-2))) printf("%d %d\n",i-2,i);
```

```
  }

int prost(int k)
  { int  i=2,prosti=1;
    while((i<=sqrt(k)) && (prosti))  { prosti=((k%i)!=0);   i++; }
    return(prosti);
  }
```

U nekim drugim programskim jezicima (FORTRAN), takav poziv funkcije može da promeni vrednost za *v*. Mehanizam izvršen u tom slučaju se naziva poziv po adresi (*call by Reference*). Da bi se u C postigao efekat poziva po adresi moraju se koristiti pointeri promenljivih u listi parametara prilikom definisanja funkcije, kao i adrese promenljivih koje su argumenti u pozivu funkcije.

## 5.3.2. Prenos po rezultatu (Call by Result)

Prenos po rezultatu je tehnika koji se primenjuje za prenos izlaznih (*out*) parametara u jeziku Ada. Kada se argumenti prenose po rezultatu, njihove vrednosti se izračunavaju u lokalnim memorijskim lokacijama formalnih argumenata potprograma, pri čemu nema prenosa vrednosti parametara iz glavnog programa u potprogram. Međutim, pre nego što se upravljanje tokom izvršenja prenese u glavni program, ove vrednosti se kopiraju u memorijske lokacije stvarnih argumenata. Stvarni argumenti u ovom slučaju moraju da budu promenljive. Kao i kod prenosa po vrednosti, za fiktivne argumente potprograma rezervišu se lokalne memorijske lokacije.

Kod ovog načina prenosa mogući su i neki negativni efekti, kao što je na primer kolizija stvarnih argumenata, koja nastaje u slučajevima kada se potprogram poziva tako da isti stvarni argument zamenjuje više fiktivnih argumenata. Razmotrimo sledeći primer u kome se potprogram definisan kao procedure `PP(x,y:real)` poziva sa `PP(p1,p1)`.U ovom primeru problem je u tome što se u potprogramu aktuelni parametar *p*1 dobija vrednost iz dve različite memorijske lokacije (koje odgovaraju formalnim parametrima *x* i *y*). Nije unapred poznato koju će od ovih vrednosti argument *p*1 imati po završetku potprograma. To zavisi od toga koja je vrednost poslednji put dodeljena stvarnom argumentu.

## 5.3.3. Prenos po vrednosti i rezultatu (Call by Value-Result)

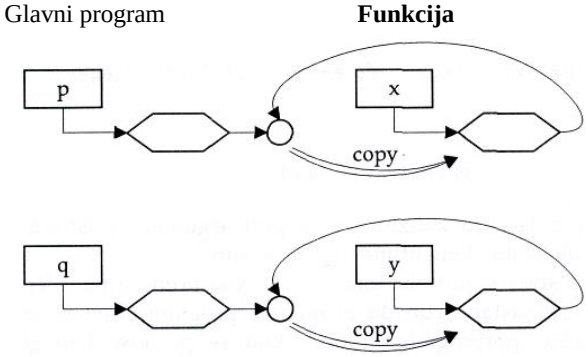
Prenos po vrednosti i rezultatu je način prenosa ulazno-izlaznih (*inout*) argumenata. To je u suštini kombinacija prenosa po vrednosti i prenosa po rezultatu. Vrednosti stvarnih argumenta kopiraju se u memorijske lokacije fiktivnih argumenata potprograma. To su lokalne memorijske lokacije potprograma i sve izmene nad argumentima pamte se u ovim lokacijama. Pre vraćanja upravljanja glavnom programu, rezultati potprograma se iz ovih lokalnih memorijskih lokacija ponovo kopiraju u memorijske lokacije stvarnih argumenata. Ovaj način prenosa se često naziva i prenos kopiranjem (*call by copy*) jer se stvarni argumenti najpre kopiraju u potprogram, a zatim ponovnim kopiranjem rezultati vraćaju u glavni program.

Nedostaci i negativni efekti koji su prisutni kod prenosa po vrednosti i prenosa po rezultatu ostaju i kod ovog načina prenosa.

## 5.3.4. Prenos po referenci (Call by Reference)

Kod ovog načina prenosa nema kopiranja vrednosti parametara iz glavnog programa u potprogram ill obrnuto. U potprogram se samo prenosi referenca (obično samo adresa) memorijske lokacije u kojoj je sačuvana vrednost stvarnog argumenta. Efekat je kao da se reference stvarnih argumenata preslikavaju u reference fiktivnih. Potprogam pristupa istim memorijskim lokacijama kojima i glavni program, nema rezervisanja memorijskih lokacija za fiktivne argumente u okviru potprograma. Sve promene nad argumentima koje se izvrše u okviru potprograma neposredno su vidljive i u glavnom programu. Zbog toga je ovo način prenosa koji se koristi za prenos ulazno-izlaznih argumenata. U mnogim programskim jezicima to je osnovni način prenosa argumenata i podrazumeva se implicitno

(Algol, FORTRAN). U Pascal-u se koristi uvek kada uz argumente stoji *var*. U jeziku C se za prenos po referenci koriste pokazivači.




Prednosti prenosa po referenci su u njegovoj efikasnosti kako u pogledu vremena tako i u pogledu memorijskog prostora. Nema potrebe za dupliranja memorijskog prostora niti se gubi vreme u kopiranju vrednosti argumenata iz jednih memorijskih lokacija u druge. Međutim, treba imati u vidu da se ovaj prenos implementira indirektnim adresiranjem pa se tu ipak nešto gubi na efikasnosti. Takođe, mogući su i određeni bočni efekti u određenim posebnim slučajevima poziva potprograma. Razmotrićemo neke od njih.

Kao u slučaju prenosa po rezultatu i ovde je moguća kolizija između stvarnih argumenata. Uzmimo na primer potprogram *PP* u Pascal-u sa dva argumenta koji se prenose po vrednosti, čija je deklaracija oblika

```
procedure PP(var x,y: integer);
```

Ako se ovaj potprogram pozove tako da se oba fiktivna argumenta zamene istim stvarnim parametrom, na primer sa *PP*(*m*, *m*); , dolazi do kolizije jer se oba fiktivna argumenta referenciraju na istu memorijsku lokaciju stvarnog argumenta *m*.

Kod prenosa po referenci kolizija može da nastane između elemenata programa koji se prenose kao stvarni argumenti potprograma i onih koji se prenose kao globalni parametri. Razmotrimo sledeći primer u Pascal-u:

```
program LocalGlobal;
  var  global: integer;
  procedure PPG(var local: integer);
    begin
      local := local+1; writeln('global = ',global);local := local+global;
    end;
  begin
    global := 2;  PPG(global); writeln(global);
  end.
```

Odgovarajući primer u jeziku C je predstavljen sledećim kôdom:

```
#include<stdio.h>
int global;
void PPG(int *local)
{ (*local)++;
  printf("global = %d\n",global);
  *local = *local+global;
}

void main()
{ global = 2;
  PPG(&global);
  printf("%d\n",global);
}
```

Posle poziva potprograma *PPG* biće odštampana vrednost je 6. Zaista, vrednost 2 dodeljena promenljivoj *global* u potprogramu najpre koriguje naredbom `local:=local+1;` (posle čega je `global=3`), a zatim i naredbom `local:=local+global;` (posle čega je `local=global=3`). Promenljiva *global* se u potprogramu koristi i kao globalna promenljiva i kao stvarni argument, pa se i promene argumenta potprograma i promene same promenljive *global* registruju u istoj memorijskoj lokaciji. Napomenimo da bi u ovom primeru prenos sa kopiranjem (call by value-result) dao drugačiji rezultat. U tom slučaju, vrednost stvarnog argumenta *global* kopira se u memorijsku lokaciju fiktivnog argumenta *local* i u potprogramu se sve promene argumenta registruju u toj lokaciji. U prvoj naredbi dodeljivanja promenljiva *local* dobija vrednost 3, a u drugoj se ova vrednost inkrementira za 2, što je vrednost globalne promenljive *global* (dodeljena pre ulaska u potprogram). Time argument *local* dobija vrednost 5, i to je vrednost koja se po završetku potprograma kopira natrag u memorijsku lokaciju stvarnog argumenta *global*. To znači da se u ovom slučaju štampa vrednost 5. Slična analiza istog programa moguća je i za slučaj prenosa argumenta potprograma po vrednosti. U tom slučaju vrednost stvarnog argumenta *global* se kopira u potprogram ali su sve promene u potprogramu nevidljive za glavni program, pa se po završetku potprograma štampa vrednost 2 koja je dodeljena promenljivoj *global* pre poziva potprograma.

## Poziv po adresi pomoću pokazivača u C

### Deklaracije pointera i dodeljivanje

Pointeri se koriste za pristup memoriji i manipulaciju adresama. Vrednosti pokazivača jesu adrese promenljivih. Do sada smo već videli upotrebu adresa kao argumenata u naredbi *scanf*(). Ako je *v* promenljiva, tada je *&v* adresa (lokacija) u memoriji u kojoj je smeštena vrednost za *v*. Operator *&* je unarni, ima asocijativnost sdesna na levo, a prioritet kao i ostali unarni operatori. Pointerske promenljive se mogu deklarisati u programu, a onda koristiti tako da uzimaju adrese za svoje vrednosti.

Na primer, deklaracija

```
int *p;
```

deklariše promenljivu *p* tipa "pointer na *int*". Rang vrednosti svakog pointera uključuje posebnu adresu 0, definisanu kao *NULL* u `<stdio.h>`, kao i skup pozitivnih celih brojeva koji se interpretiraju kao mašinske adrese.

**Primer.** Pointeru *p* se vrednosti mogu dodeljivati na sledeći način:.

```
p=&i;
p=0;
p=NULL;          /* isto što i p=0; */
p=(int*)1507     /* apsolutna adresa u memoriji */ }
```

U prvom primeru *p* je "pointer na *i*", odnosno "sadrži adresu od *i*". Treći i drugi primer predstavljaju dodelu specijalne vrednosti 0 pointeru *p*. U četvrtom primeru kast je nužan da bi se izbeglo upozorenje kompajlera, a koristi se aktuelna memorijska lokacija kao vrednost promenljive *p*. Ovakve naredbe se ne koriste u korisničkim programima, već samo u sistemskim.

### Adresiranje i operator indirekcije

Neka su date deklaracije

```
int i, *p;
```

Tada naredbe

```
p=&i;       scanf("%d",p);
```

uzrokuju da se sledeća uneta vrednost smešta u adresu zadatu pointerom *p*. Ali, s obzirom da *p* ukazuje na *i*, to znači da se vrednost smešta u adresu od *i*.

Operator indirekcije * je unarni i ima isti prioritet i asocijativnost sa desna ulevo kao i ostali unarni operatori. Ako je *p* pointer, tada **p* predstavlja vrednost promenljive na koju ukazuje *p*. Direktna vrednost za *p* je memorijska lokacija, dok je **p* indirektna vrednost od *p*, tj. vrednost memorijske lokacije sadržane u *p*. U suštini, * je inverzni operator u odnosu na operator *&*.

Na taj način, pointeri se mogu koristiti za dodelu vrednosti nekoj promenljivoj. Na primer, neka su date deklaracije

```
int *p,x;
```

Tada se izrazima

```
p=&x; *p=6;
```

promenljivoj *x* dodeljuje vrednost 6.

**Primer.** Uočimo deklaracije

```
float x,y, *p;
```

i naredbe

```
p = &x;  y=*p;
```

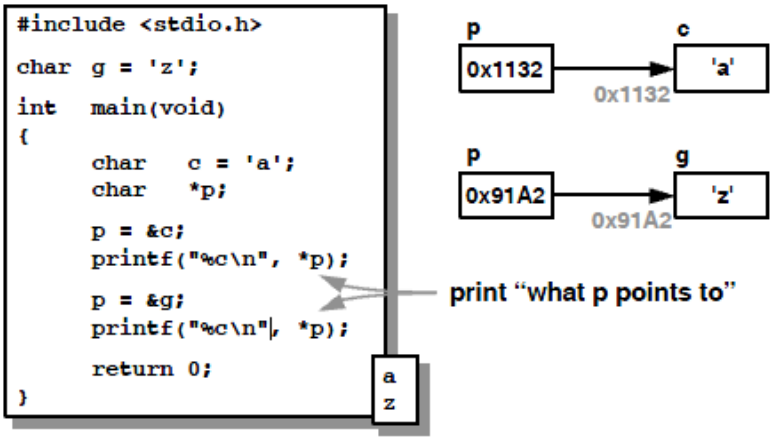
Prvom naredbom se adresa od *x* pridružuje pointeru *p*, a drugom se vrednost na koju ukazuje *p* dodeljuje *y*. Ove dve naredbe su ekvivalentne sa

```
y=*&x;
```

odnosno

```
y=x.
```

**Primer.** Ilustracija pokazivača.




**Primer.** Neka je dat sledeći kôd:

```
char c1,c2='A',*p,*q;
p=&c1;  q=&c2; *p=*q;
```

Adrese od *c*1 i *c*2 pridružuju se promenljivima *p* i *q* u prva dva izraza. Poslednja naredba izjednačuje vrednost na koju ukazuje *p* i vrednost na koju ukazuje *q*. Ove tri naredbe su ekvivalentne sa *c*1=*c*2.

**Primer.**

```
void main()
  { int i=777,*p;
     p=&i;     printf("Vrednost za i:%d\n",*p);
     printf("Adrese za i: %u ili %p\n",p,p);
  }
```

Formati *%u* i *%p* se koriste za prikazivanje vrednosti promenljive *p* u obliku neoznačenog decimalnog broja, odnosno heksadecimalnog broja, respektivno.

**Primer.** Izrazima

```
int i=777, *p=&i;
```

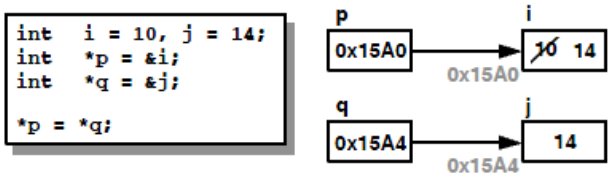
je inicijalizovano *p*, a ne **p*. Promenljiva *p* je tipa "pointer na *int*", i njena početna vrednost je *&i*.

**Primer.**

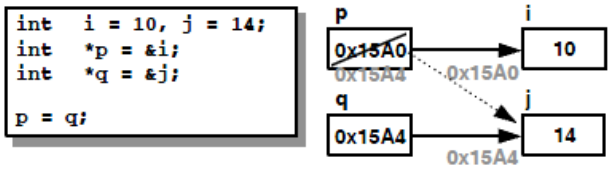
```
#include <stdio.h>
void main()
 { int x=7, y=10;
```

```
   printf("x=%d &x=%p\n", x,&x);
   printf("y=%d &y= %p%u\n",y,&y, &y);
 }

void main()
 { int i=5, *pi=&i;  printf("i= %d ili = %d\n", i, *pi); }
```

**Primer.** Postoji velika razlika između




i

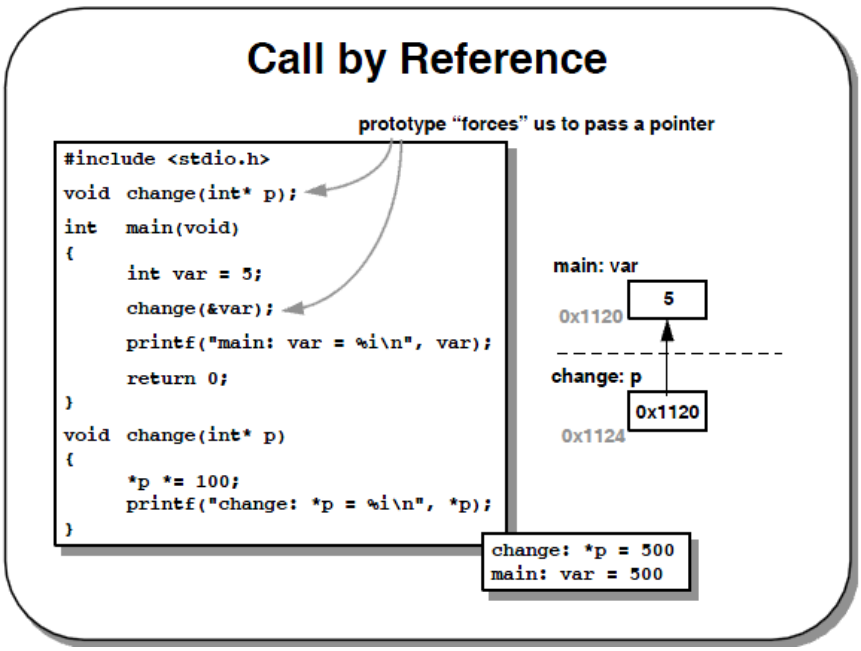



Using pointers allows us to:
– Achieve call by reference (i.e. write functions which change their parameters)
– Handle arrays efficiently
– Handle structures (records) efficiently
– Create linked lists, trees, graphs etc.
– Put data onto the heap
– Create tables of functions for handling Windows events, signals etc.

### Call by reference (poziv po adresi) koristeći pokazivače

Prenos vrednosnih parametara je pogodan kada funkcija vraća jednu izlaznu veličinu bez izmene svojih stvarnih parametara. Međutim, kada je potrebno da funkcija menja vrednosti svojih stvarnih parametara, ili ukoliko funkcija vraća više izlaznih veličina, može se koristiti druga tehnika predaje parametara: prenos adrese stvarnih parametara.

Adrese promenljivih se mogu koristiti kao argumenti funkcija u cilju izmene zapamćenih vrednosti promenljivih u pozivajućoj programskoj jedinici. Tada se pointeri koriste u listi parametara pri definisanju funkcije. Kada se funkcija poziva, moraju se koristiti adrese promenljivih kao argumenti.




**Primer.** Primer u jeziku C kojim se ilustruju globalna promenljiva i prenos po adresi. Rezultat je kao u slučaju prenosa po adresi.

```
#include<stdio.h>
int global;
void PPG(int *local)
{ (*local)++; printf("global = %d\n",global); *local = *local+global;}
void main()
{ global = 2;  PPG(&global); printf("%d\n",global); }
```

**Primer.** Date su vrednosti za dve celobrojne promenljive *i* i *j*. Urediti ih u poredak *i*<= *j*.

```
#include <stdio.h>
void upis(int *x, int *y);
void sredi(int *x, int *y);
void ispis(int x,int y);
void main()
  { int x,y;
    upis(&x,&y);  sredi(&x,&y);  ispis(x,y);
  }

void upis(int *x, int *y)
 { printf("Unesi dva cela broja"); scanf("%d%d",x,y);  }

void sredi(int *x, int *y)
  { int pom;
    if(*x>*y)     { pom=*x; *x=*y; *y=pom; }
  }

void ispis(int x, int y)
  {  printf("\n To su brojevi %d %d\n",x,y); }
```

Tehnika predaje parametara po adresi (*Call by reference*) se sastoji iz sledećih baznih pravila:

1. deklarisati formalne parametre funkcije kao pointere;
2. koristiti operatore indirekcije u telu funkcije;
3. pri pozivu funkcije koristiti adrese kao argumente.

Kako *sredi*() radi sa pokazivačima? Kada predajete pokazivač, predajete adresu objekta i tako funkcija može manipulisati vrednošću na toj adresi. Da bi *sredi*() promenila stvarne vrednosti,

korišćenjem pokazivača, funkcija *sredi*(), trebalo bi da bude deklarisana da prihvata dva *int* pokazivača. Zatim, dereferenciranjem pokazivača, vrednosti *x* i *y* će, u stvari, biti promenjene.

**Primer.** Predavanje po referenci korišcenjem pokazivača.

```
1:  // Prenos parametara po referenci
2:
3:  #include <iostream.h>
4:
5:  void swap(int *x, int *y);
6:
7:  int main()
8:   {
9:     int x = 5, y = 10;
10:
11:    cout << "Main. Pre swap, x:”<<x<<" y: "<<y<<"\n";
12:    swap(&x,&y);
13:    cout<<"Main. Posle swap., x: "<<x<<" y: "<<y<<"\n";
14:    return 0;
15:  }
16:
17: void swap(int *px, int *py)
18:   {
19:     int temp;
20:
21:   cout<<"Swap. Pre swap, *px: "<<*px<<" *py:"<<*py<<"\n";
22:
23:   temp = *px;
24:   *px = *py;
25:   *py = temp;
26:
27:  cout<<"Swap. Posle swap, *px: "<<*px<<" *py: "<<*py<<"\n";
28:
29: }
```

Izlaz:

```
Main. Pre swap, x:5 y: 10
Swap. Pre swap, *px: 5 *py:10
Swap. Posle swap, *px: 10 *py:5
Main. Posle swap, x:10 y: 5
```

U liniji 5 prototip funkcije *swap*() pokazuje da će njegova dva parametra biti pokazivači na *int*, a ne *int* promenljive, Kada se pozove *swap*() u liniji 12, kao argumenti se predaju adrese od *x* i *y*. U liniji 19 lokalna promenljiva, *temp*, deklarisana je u funkciji *swap*() - nije potrebno da *temp* bude pokazivač; ona će čuvati samo vrednost od **px* (to jest, vrednost promenljive *x* u pozivajućoj funkciji), tokom života funkcije. Posle povratka iz funkcije, promenljiva *temp* više neće biti potrebna.

U liniji 23 promenljivoj *temp* se dodeljuje vrednost memorijske lokacije koja je jednaka vrednosti pokazivača *px*. U liniji 24 vrednost na adresi *px* se dodeljuje vrednosti na adresi *py*. U liniji 25 vrednost čuvana u *temp* (to jest, originalna vrednost na adresi *px*) stavlja se u adresu *py*.

Efekat ovoga je da će vrednosti u pozivajućoj funkciji, čije su adrese predate funkciji *swap*(), biti zamenjene.

## Prenos po referenci koristeći reference u C++

Prethodni program radi, ali sintaksa funkcije *swap*() je problematična zbog dve stvari. Prvo, ponavljajuća potreba za dereferenciranjem pokazivača unutar funkcije *swap*() čini je podložnom greškama i teškom za čitanje. Drugo, potreba da se preda adresa promenljivih u pozivajućoj funkciji čini unutrašnji rad funkcije *swap*() više nego očiglednim za korisnike.

Cilj C++ je da spreči korisnika funkcije da se brine o tome kako ona radi. Zbog toga se prenos po adresi u C++ može uraditi pomoću referenci.

**Primer.**

```
#include<iostream.h>
void f(int i, int &j){   // i je poziv po vrednosti, j po referenci
  i++;                   // stvarni argument si se neće promeniti
  j++;                   // stvarni argument sj će se promeniti
}
void main () {
  int si=0,sj=0;
  f(si,sj);
  cout<<"si="<<si<<", sj="<<sj<<endl;
}
```

```
Izlaz: si=0, sj=1
```

**Primer.** Prepisana funkcija *swap*() sa referencama.

```
1: //Predavanje po referenci
2: // korišćenjem referenci!
3:
4:  #include <iostream.h>
5:
6:  void swap(int &rx, int &ry);
7:
8:  int main()
9:   {
10:    int x = 5, y = 10;
11:
12:    cout<<"Main. Pre swap, x: "<<x<<" y: "<<y<<"\n";
13:    swap(x,y);
14:    cout<<"Main. Posle swap, x: "<<x<<" y: "<<y<<"\n";
15:    return 0;
16:  }
17:
18: void swap (int &rx, int &ry)
19:   {
20:     int temp;
21:
22:     cout<<"Swap. Pre swap, rx: "<<rx<<" ry: "<<ry<<"\n";
23:
24:     temp =rx;
25:     rx =ry;
26:     ry =temp,
27:
28:     cout<<"Swap. Posle swap, rx: "<<rx<<" ry: "<<ry<<"\n";
29:
30:  }
```

Rezultat je isti kao u prethodnom primeru.

Kao i u primeru si pokazivačima, deklarisane su dve promenljive u liniji 10, a njihove vrednosti se štampaju u liniji 12. U liniji 13 poziva se funkcija *swap*(), ali uočite da se predaju *x* i *y*, a ne njihove adrese. Pozivajuća funkcija *main*() jednostavno predaje promenljive, kao i u slučaju poziva po vrednosti.

Kada se pozove *swap*(), izvršenje programa "skače" na liniiju 18, gde se promenljive *rx* i *ry* identifikuju kao reference. Njihove vrednosti se štampaju u liniji 22, ali uočite da se za štampanje ne zahtevaju posebni operatori. Promenljive *rx* i *ry* su alijasi za originalne vrednosti i mogu se samostalno koristiti.

U linijama 24-26 vrednosti se zamenjuju, a onda se štampaju u liniji 28, Izvršenje programa "skače" u pozivajuću funkcuju i u liniji 14 vrednosti se štampaju u *main*(). Zato što su parametri za *swap*() deklarisani kao reference, vrednosti iz *main*() se predaju po referenci i time se, takođe, menjaju i u *main*().

Reference obezbeđuju pogodnost i lakoću upotrebe normalnih promenljivih, sa snagom i sposobnošću predavanja po referenci koju imaju pokazivači.

**Primer.** Prenos parametra na dva načina.

```
#include<iostream.h>
int* f(int* x) {
  (*x)++;
  return x; // Vrednost za x se prenosi u stvarni parametar
}

int& g(int& x) {
  x++; // Isti efekat kao u f()
  return x; // Vrednost za x se prenosi u stvarni parametar
}

int main() {
  int a = 0;
  f(&a); // Ružno, ali eksplicitno
  cout << "a = " << a << "\n";
  g(a);  // Jasno, ali sakriveno
  cout << "a = " << a << "\n";
  return 0;
}
```

**Primer.** Modifikacija prethodnog primera, koja se sastoji u upotrebi pokazivača *p*. Naredbom `p=f(&a);` identifikovani su *p* i &*a*.

```
#include<iostream.h>
int* f(int* x) {
  (*x)++;
  return x; // Vrednost za x se prenosi u stvarni parametar
}

int& g(int& x) {
  x++; // Isti efekat kao u f()
  return x; // Vrednost za x se prenosi u stvarni parametar
}

void main() {
  int a = 0,*p;
  p=f(&a); // Ružno, ali eksplicitno
  cout << "a = " << a << "\n";  //a=1
  cout << "*p = " << *p << "\n"; //*p=1
  g(a);  // Jasno, ali sakriveno
  cout << "a = " << a << "\n";  //a=2
  cout << "*p = " << *p << "\n";  // *p=2
}
```

## Vraćanje višestrukih vrednosti

Kao što je rečeno, pomoću naredbe *return* funkcije mogu vratiti samo jednu vrednost. Šta učiniti ako je potrebno da vratite dve vrednosti iz funkcije? Jedan način da rešite ovaj problem je da funkciji predate objekte po referenci. Kako predavanje po referenci dozvoljava funkciji da promeni originalne objekte, ovo efektno dozvoljava funkciji da vrati više informacija. Povratna vrednost funkcije se može rezervisati za izveštavanje o greškama.

Ovo se može ostvariti referencama, ili pokazivačima. Sledeći primer demonstrira funkciju koja vraća tri vrednosti: dve kao pokazivačke parametre i jednu kao povratnu vrednost funkcije.

**Primer.** Vraćanje vrednosti pomoću pokazivača.

```
1:
2: // Vraćanje više vrednosti iz funkcije
3:
4: #include <iostream.h>
5:
6: typedef unsigned short USHORT;
7:
8: short Factor(USHORT, USHORT*, USHORT*);
9:
10: int main()
11: {
12:   USHORT number, squared, cubed;
13:   short error;
14:
15:   cout << "Unesite broj (0-20): ";
16:   cin >> number;
17:
18:   error = Factor(number, &squared, &cubed);
19:
20:   if(!error)
21:    {
22:      cout << "broj: " << number << "\n";
23:      cout << "kvadrat: " << squared << "\n";
24:      cout << "kub: " << cubed << "\n";
25:    }
26:   else
27:      cout << "Doslo je do greske!!\n";
28:   return 0;
29: }
30:
31: short Factor(USHORT n, USHORT *pSquared, USHORT *pCubed)
32: {
33:   short Value = 0;
34:   if (n > 20)
35:      Value = 1;
36:   else
37:    {
38:       *pSquared = n*n;
39:       *pCubed = n*n*n;
40:       Value = 0;
41:    }
42:   return Value;
43: }
```

U liniji 12 *number*, *squared* i *cubed* su definisani kao *USHORT*. Promenljivoj number se dodeljuje broj baziran na korisničkom ulazu. Ovaj broj i adresa od *squared* i *cubed* se predaju funkciji *Factor*().

Funkcija *Factor*() ispituje prvi parametar, koji se predaje po vrednosti. Ako je on veći od 20 (maksimalan broj kojim ova funkcija može rukovati), ona postavlja povratnu vrednost (*Value*) na jednostavnu vrednost greške (*Value* =1). Uočite da povratna vrednost 0 iz funkcije *Factor*() pokazuje da je sve prošlo dobro, i uočite da tunkcija vraća ovu vrednost u liniji 42.

Stvarne potrebne vrednosti, kvadrat i kub promenljive *number*, vraćaju se menjanjem pokazivača koji su predati funkciji.

U linijama 38 i 39 pokazivačima se dodeljuju njihove povratne vrednosti. U liniji 40 return Value dobija uspešnu vrednost.

Jedno poboljšanje u ovom programu bi moglo biti deklarisanje sledećeg:

```
enum ERROR_VALUE {SUCCESS, ERROR};
```

Zatim, umesto vraćanja 0, ilii, program bi mogao da vrati SUCCESS, ili ERROR.

## Vraćanje višestrukih vrednosti po referenci

Prethodni program se može učiniti lakšim za čitanje i održavanje, korišćenjem referenci, umesto pokazivača. Sledeći primer prikazuje isti program, prepisan radi korišćenja referenci i uključivanja *ERROR* enumeracije.

**Primer.** Vraćanje višestrukih vrednosti korišćenjem referenci.

```
1: //
2: // Vraćanje više vrednosti iz funkcije
3: // konščenje referenci
4:
5: #include <iostream.h>
6:
7: typedef unsigned short USHORT;
8: enum ERR_CODE {SUCCESS, ERROR};
9:
10: ERR_CODE Factor(USHORT, USHORT &, USHORT &);
11:
12: int main()
13: {
14:   USHORT number, squared, cubed;
15:   ERR_CODE result;
16:
17:   cout << "Unesite broj (0 - 20): ";
18:   cin >> number;
19:
20:   result = Factor(number, squared, cubed);
21:
22:   if (result == SUCCESS)
23:   {
24:     cout << "broj: " << number << "\n";
25:     cout << "kvadrat: " << squared << "\n";
26:     cout << "kub: " << cubed << "\n";
27:   }
28:  else
29:     cout << "Doslo je do greske!!\n";
30:   return 0;
31: }
32:
33: ERR_CODE Factor(USHORT n, USHORT &rSquared, USHORT &rCubed)
34: {
35:   if (n > 20)
36:      return ERROR;   // jednostavan kôd za grešku
37:   else
38:    {
39:       rSquared = n*n;
40:       rCubed = n*n*n;
41:       return SUCCESS;
42:    }
43: }
```

Ovaj program je identičan prethodnom, uz dva izuzetka. Enumeracija ERR_CODE čini izveštavanje o grešci u linijama 36 i 41, kao i rukovanje greškama u liniji 22.

Ipak, veća promena je ta da je *Factor*() sada deklarisana da prihvata reference na *squared* i *cubed*, a ne na pokazivače, što čini manipulaciju ovim parametrima daleko jednostavnijom i lakšom za razumevanje.

## Predavanje po referenci, zbog efikasnosti

Svaki put kada predate objekat funkciji po vrednosti, pravi se njegova kopija. Svaki put kada vratite po vrednosti objekat iz funkcije, pravi se druga kopija.

Ovi objekti se kopiraju na stek. Zato ovo uzima vreme i memoriju. Za male objekte, kao što su ugrađene celobrojne vrednosti, to je trivijalna cena. Međutim, za veće korisnički kreirane objekte, cena je veća. Veličina korisnički kreiranih objekata na steku je suma veličina svih njegovih promenljivih članica. Svaka od njih, dalje, može biti korisnički kreiran objekat i predavanje tako masivne strukture njenim kopiranjem na stek može biti veoma skupo zbog performansi i korišćenja memorije.

Postoji, takođe, i druga cena. Sa klasama koje kreirate, svaka ova privremena kopija se kreira kada kompajler pozove specijalan konstruktor: *konstruktor kopije*. Za sada, dovoljno je da znate da se konstruktor kopije poziva svaki put kada se privremena kopija objekta stavi na stek.

**Primer.** Urediti tri broja *x, y, z* u neopadajući poredak *x*<= *y*<= *z*.

```
void razmeni(int *a, int *b)
  { int pom;     pom=*a; *a=*b; *b=pom;  }

void main()
 { int x,y,z;     void razmeni(int*, int*);
   scanf("%d%d%d", &x,&y,&z);
   if(x>y) razmeni(&x,&y);     if(x>z) razmeni(&x,&z);
   if(y>z) razmeni(&y,&z);
   printf("x= %d y= %d z= %d\n",x,y,z);
 }
```

**Primer.** Napisati proceduru kojom se izračunava najmanji zajednički sadržalac i najveći zajednički delilac dva prirodna broja.

```
#include<stdio.h>
void unos(int *,int *); void nzds(int,int, int *,int *);
void main()
  { int x,y, nzd,nzs;
    unos(&x, &y);    nzds(x,y,&nzd,&nzs);
    printf("Nzd unetih brojeva = %d a nzs= %d\n",nzd,nzs);
  }

void unos(int *a, int *b)
 { printf("\nZadati dva cela broja: "); scanf("%d%d",a,b);  }

void nzds(int a, int b, int *nd, int *ns)
   { if(a>b) *ns=a; else *ns=b;
      int v=*ns;
      while(*ns%a !=0 || *ns%b !=0)(*ns)+=v;
      while(a!=b)    { if(a>b)a-=b;   else if(b>a)b-=a;  }
      *nd=a;
   }
```

**Primer.** Sa tastature se unosi jedan ceo broj, a za njim neodređen broj celih brojeva. Napisati proceduru kojom se izračunava minimum i maksimum unetih brojeva.

```
#include<stdio.h>
void minmax(int *,int*);
void main()
  { int mn,mx; minmax(&mn,&mx);
     printf("Minimum je %d a maksimum  %d\n",mn,mx);
   }

void minmax(int *min, int *max)
 { int n;
   *min=*max=scanf("%d",&n);
```

```
   while(scanf("%d",&n)==1)
          if(n<*min)*min=n;
          else if(n>*max)*max=n;
 }
```

**Primer.** Napisati funkciju za izračunavanje determinante kvadratnog trinoma $ax^2+bx+c$. Napisati proceduru kojom se izrašavaju sledeće aktivnosti:

a) Određuje da li su rešenja realna i različita, realna i jednaka ili konjugovano-kompleksna;

b) u slučaju da su rešenja realna izračunava njihove vrednosti, a u slučaju da su rešenja konjugovano-kompleksna izračunava realni i imaginarni deo tih rešenja.

Napisati proceduru za štampanje odgovarajućih izveštaja. U glavnom programu unositi koeficijente kvadratnih jednačina u beskonačnom ciklusu i prikazivati odgovarajuće rezultate koristeći proceduru za prikazivanje izveštaja.

```
#include<stdio.h>
#include<math.h>
float diskriminanta(float, float, float);
void solution(float, float, float, float *, float *,int *); void
rezultati(float, float, float, float *,float *,int *);
void main()
 { float a,b,c, x1,x2;    int tip;
    while(1)
     { printf("Unesi koeficijente -> "); scanf("%f%f%f", &a,&b,&c);
        rezultati(a,b,c,&x1, &x2, &tip);
}  }
float diskriminanta(float a, float b, float c)
   { return(b*b-4.0*a*c); }
void solution(float a,float b,float c, float *x,float *y,int *t)
  { float d;
     d=diskriminanta(a,b,c);
     if(d>0){*t=0; *x=(-b+sqrt(d))/(2*a);*y=(-b-sqrt(d))/(2*a); }
     else if(d==0) { *t=1; *x=*y=-b/(2*a);}
  else { *t=2; *x=-b/(2*a); *y=sqrt(-d)/(2*a); }
}
void rezultati(float a,float b,float c, float *x, float *y, int *t)
 {solution(a,b,c,x,y,t);
  if(*t==0)
    {printf("Resenja su realna i razlicita\n");
     printf("x1=%f x2=%f\n",*x,*y);}
  else if(*t==1)
    {printf("Resenja su realna i jednaka\n");
     printf("x1=x2=%f\n",*x);}
  else    { printf("Resenja su konjugovano-kompleksna\n");
            printf("x1 = %f + i* %f\n",*x,*y);
            printf("x2 = %f -i* %f\n",*x,*y);
}         }
```

**Primer.** Napisati proceduru za deljenje dva cela broja na proizvoljan broj decimala. Deljenik, delilac i broj decimala zadati u posebnoj proceduri.

```
#include<stdio.h>
#include<conio.h>
void unos(int *, int*, int *);
void deljenje(int, int, int);
main()
  { int n,i,bdec, brojilac, imenilac;
    clrscr(); printf("Koliko puta? "); scanf("%d", &n);
    for(i=1; i<=n; i++)
         { unos(&brojilac, &imenilac, &bdec);
```

```
           deljenje(brojilac, imenilac, bdec);
  }      }
void unos(int *br, int *im, int *bd)
  { printf("Brojilac = ?");  scanf("%d",br);
    printf("Imenilac = ? "); scanf("%d",im);
    printf("Broj decimala = ? "); scanf("%d",bd);
  }
void deljenje(int br, int im, int bd)
 { int i;
   if(br*im<0) printf("-");
   br=br<0?-br:br;  im=im<0?-im:im;
   printf("\n%d.",br/im);    br %= im;
   for(i=1; i<=bd; i++)
     { br *=10; printf("%d",br/im); br %=im; }
   printf("\n");
 }
```

**Primer.** a) Napisati proceduru za unošenje brojioca i imenioca jednog razlomka. U toj proceduri, po potrebi, vrednost imenioca promeniti tako da bude pozitivan.

b) Napisati rekurzivnu funkciju za izračunavanje najvećeg zajedničkog delioca dva prirodna broja.

c) Napisati funkciju za izračunavanje najvećeg zajedničkog sadržaoca dva prirodna broja.

d) Napisati proceduru za kraćenje brojioca i imenioca zadatim prirodnim brojem.

e) Napisati proceduru za sabiranje dva razlomka. Pri sabiranju razlomaka koristiti najveći zajednički sadržalac za imenioce jednog i drugog razlomka. Zatim skratiti brojilac i imenilac izračunatog razlomka najvećim zajedničkim deliocem za brojilac i imenilac.

f) U glavnom programu učitati brojilac i imenilac za *n* razlomaka i izračunati zbir svih razlomaka.

```
#include<stdio.h>
#include<math.h>
#include<conio.h>
/* razlomci.c */
int nzd(int, int); int nzs(int, int); void unos(int *, int *);
void sabiranje(int, int,int, int, int *, int *);

void kracenje(int *, int*, int); main()
 { int i,n, broj,imen, brojilac, imenilac, brez, irez;
    clrscr();    printf("Koliko razlomaka sabirate? ");
    scanf("%d", &n);
    for(i=1; i<=n; i++)
       {unos(&brojilac, &imenilac);
        if(i==1){ brez=brojilac; irez=imenilac;        }
        else
         {broj=brez; imen=irez;
          sabiranje(brojilac,imenilac,broj,imen, &brez,&irez);
              }
       printf("%d/%d\n", brez, irez);
}
void unos(int *br, int *im)
{ printf("Brojilac -> " ); scanf("%d",br);
  printf("Imenilac -> " ); scanf("%d",im);
  if(*im<0) {*br=-*br; *im=-(*im); }
}
int nzd(int br, int im)
 { if(br == im) return(br);
    else if(br>im)return(nzd(br-im,im));
   else return(nzd(br,im-br));
  }
int nzs(int br, int im)
 { int ns;
```

```
   if(br>im) ns=br; else ns=im;
    while((ns %br !=0) || (ns %im != 0))ns++;
    return(ns);
 }
void kracenje(int *br, int *im, int k)
   { *br /=k; *im /=k;  }

void sabiranje(int pb,int pi,int db,int di,int *rb,int *ri)
 { int ns, nd;
    ns=nzs(pi,di); *ri=ns;  *rb=pb*ns/pi+db*ns/di;
    nd=nzd(*rb, *ri);
    kracenje(rb, ri, nd);
 }
```

**Primer.** Napisati proceduru za učitavanje i ispis kompleksnih brojeva kao i procedure za izvođenje osnovnih aritmetičkih operacija sa kompleksnim brojevima. Napisati test program.

```
#include<stdio.h>
#include<math.h>
void unos(float *,float *, float *, float *);
void saberi(float, float, float, float, float *, float *);
void oduzmi(float, float, float, float, float *, float *);
void mnozi(float, float, float, float, float *, float *);
void deli(float, float, float, float, float *, float *);
void ispis(float, float);
void main()
 { float x1,y1,x2,y2,re,im;
   unos(&x1,&y1,&x2,&y2);
   saberi(x1,y1,x2,y2, &re,&im);  printf("\nNjihov zbir je: ");
   ispis(re,im); printf("\n");
   oduzmi(x1,y1,x2,y2, &re,&im);
   printf("\nNjihova razlika je: "); ispis(re,im); printf("\n");
   mnozi(x1,y1,x2,y2,&re,&im);
   printf("\nNjihov proizvod je: "); ispis(re,im); printf("\n");
    deli(x1,y1,x2,y2,&re,&im);
    printf("\nNjihov kolicnik je: "); ispis(re,im);
    printf("\n");
  }
void unos(float *re1, float *im1, float *re2, float *im2)
  { printf("Prvi kompleksni broj? "); scanf("%f%f", re1,im1);
     printf("Drugi kompleksni broj? "); scanf("%f%f", re2,im2);
  }
void saberi(float re1,float im1,float re2,float im2,
            float *rez, float *imz)
  {*rez=re1+re2; *imz=im1+im2;  }
void oduzmi(float re1, float im1, float re2, float im2,
            float *rez, float *imz)
 {*rez=re1-re2; *imz=im1-im2;  }
void mnozi(float re1, float im1, float re2, float im2,
           float *rez, float *imz)
 {*rez=re1*re2-im1*im2; *imz=re1*im2+re2*im1;  }
void deli(float re1, float im1, float re2, float im2,
          float *rez, float *imz)
 {*rez=(re1*re2+im1*im2)/(re2*re2+im2*im2);
  *imz=(-re1*im2+re2*im1)/(re2*re2+im2*im2);
 }
void ispis(float x, float y)
 { printf("\n %f",x);
   if(y<0) printf("-"); else printf("+");
   printf("i*%f\n",fabs(y));
 }
```

**Primer.** a) Napisati funkciju *NZD*(*a*,*b*) kojom se izračunava najveći zajednički delilac za prirodne brojeve *a* i *b*.

b) Koristeći funkciju *NZD*(*a*,*b*) izračunati *NZD*(*a*,*b*,*c*).

d) Napisati funkciju *skratic*(*int a*, *int b*, *int *c*, *int *d*) koja razlomak *a/b* (*b*≠0) dovodi do neskrativog razlomka *c/d*.

e) Napisati funkciju *skrati*(*int a*, *int b*, *int &c*, *int &d*) koja razlomak *a/b* (*b*≠0) dovodi do neskrativog razlomka *c/d*.

f) U funkciji *main*:

- Za tri učitana prirodna broja naći njihov NZD.
- Za dva razlomka koji su učitani preko tastature i učitani operacijski znak (+ za sabiranje, - za oduzimanje, * za množenje, / za deljenje) odrediti rezultat računanja u neskrativom obliku.

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

int NZD(int a, int b)
{  int r;
   while(a%b)     { r=a%b; a=b;  b=r; }
   return(b);
}

int NZD3(int a, int b, int c)
{ return(NZD(a,NZD(b,c))); }

void skratic(int a, int b, int *c, int *d)   /* Verzija za C */
{ int nd=NZD(a,b);
  *c=a/nd; *d=b/nd;
}

void skrati(int a, int b, int &c, int &d)   /* Verzija za C++ */
{  int nd=NZD(a,b);
   c=a/nd; d=b/nd;
}

int NZS(int a, int b)
{ int p, n;
   if(a<b) { p=a; a=b; b=p; }
   n=a;
   while(n%b) n+=a;
   return n;
}

void operacija(int a, int b, int c, int d, char ch, int &br, int &im)
{  int as, bs, cs, ds;
      skrati(a,b,as,bs); skrati(c,d,cs,ds);
      switch(ch)
      { case '+':  im=NZS(bs,ds),   br=as*im/bs+cs*im/ds; break;
        case '-':  im=NZS(bs,ds),   br=as*im/bs-cs*im/ds; break;
        case '*':  br=as*cs; im=bs*ds; break;
        case '/':  br=as*ds; im=bs*cs; break;
      }
      int nd=NZD(br, im);
      im= im/nd; br=br/nd;
}

void ispis(int br, int im)
{ if(br*im<0)printf("-");
  printf("%d/%d\n",abs(br),abs(im));
}
```

```
void main()
{ int a,b,c,d;
  printf("a,b = ?  ");     scanf("%d%d",&a,&b);
  printf("c= ?  ");      scanf("%d",&c);
  printf("NZD3(%d,%d,%d)=%d\n",a,b,c,NZD3(a,b,c));
  int br,im;
  skratic(a,b,&br,&im); // skrati(a,b,br,im);
  printf("Prvi razlomak? "); scanf("%d%d",&a,&b);
  printf("Drugi razlomak? "); scanf("%d%d",&c,&d);
  char op;
  printf("Operacija = ? "); scanf("%c",&op);  scanf("%c",&op);
  operacija(a,b,c,d,op,br,im);
  printf("Skraceni razlomak = "); ispis(br,im);
}
```

**Primer.** Napisati funkciju koja vraća *n* dana stariji datum.

**Rešenje u C++:**

```
#include<stdio.h>

int prestupna(int g)
{ return ((g%4==0) && !(g%100==0))||(g%400==0);  }

void sutra(int & d, int & m, int & g)
{   switch(m)
      {   case 1: case 3: case 5: case 7: case 8: case 10: case 12:
            if (d<=30) d++;
            else
            { d=1;
              if(++m==13)
                  {  m=1; g++;      }
            } break;
            case 4: case 6: case 9: case 11:
                  if (d<30) d++;
                  else
                  {     d=1; m++;   } break;
            case 2:
                  if(++d>28+prestupna(g))
                  {     d=1; m++;   } break;
      }
}

void main()
{  int d,m,g,n;
   scanf("%d%d%d",&d,&m,&g);  scanf("%d",&n);
   for(int i=1;i<=n;sutra(d,m,g),i++);
   printf("%d. %d. %d.\n",d,m,g);
}
```

**Rešenje u C:**

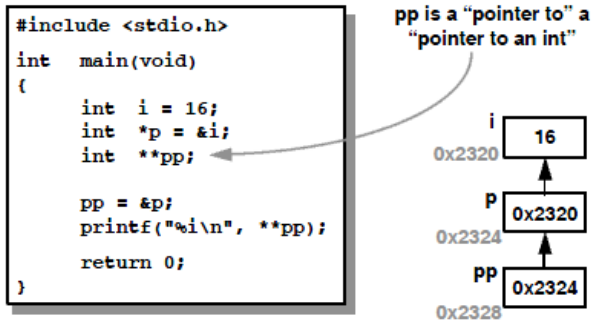
```
#include<stdio.h>
int prestupna(int g)
{ return ((g%4==0) && !(g%100==0))||(g%400==0);  }

void sutra(int *d, int *m, int *g)
{  printf("Na pocetku %d. %d. %d.\n",*d,*m,*g);
      switch(*m)
      { case 1: case 3: case 5: case 7: case 8: case 10: case 12:
                  if (*d<=30) (*d)++;
                  else
                  { *d=1; if(++(*m)==13){  *m=1; (*g)++;    }
                  } break;
```

```
        case 4: case 6: case 9: case 11:
                  if (*d<30) *d++;
                  else {  *d=1; (*m)++;   } break;
        case 2: if(++(*d)>28+prestupna(*g))
                  {     *d=1; (*m)++;     } break;
      }
printf("U funkciji %d. %d. %d.\n",*d,*m,*g);
}

void main()
{     int d,m,g,n;
      printf("Datum = ? "); scanf("%d%d%d",&d,&m,&g);
      printf("Posle koliko dana ? "); scanf("%d",&n);
      for(int i=1;i<=n;sutra(&d,&m,&g),i++);
      printf("%d. %d. %d.\n",d,m,g);
}
```

**Primer.** Napisati funkciju
`int* biggest_of_two(int*, int*);`
koja vraća pokazivač na minimum dva cela broja.
U glavnom programu učitati tri cela broja *a*, *b*, *c* a zatim izračunati njihov minimum.

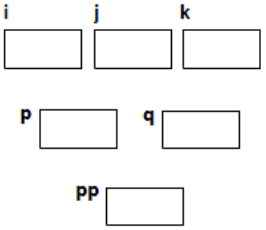
```
#include <stdio.h>
int* biggest_of_two(int*, int*);
int main(void)
{ int a, b, c;
  int *p;
  scanf("%d%d%d",&a,&b,&c);
  p = biggest_of_two(&a, &b);
  printf("the biggest of %i and %i is %i\n", a, b, *p);
  printf("the biggest of %i %i and %i is %i\n", a, b, c,
                   *biggest_of_two(&a, biggest_of_two(&b, &c)));
  return 0;
}

int* biggest_of_two(int * p, int * q)
{ return (*p > *q) ? p : q; }
```

C dozvoljava upotrebu pokazivača na proizvoljni tip, uključujući i pokazivače.




**Primer.** Šta se ispisuje izvršenjem sledećeg programa?

```
#include<stdio.h>
void main()
  { int i=10,j=7,k;
     int *p = &i;
       int *q = &j;
       int **pp=&p;
       **pp+=1;
       *pp=&k;
```



```
        **pp=*q;
        i=*q***pp;
        printf("i = %d\n",i);    // 49
        i=*q/(**pp);
        printf("i = %d\n",i);    //  1
 }
```

## 5.4. Globalne promenljive kao parametri potprograma

Sve promenljive definisane u okruženju u kome je definisan i sam potprogram mogu se koristiti i u potprogramu po konceptu globalnih promenljivih. Potprogram može da bude i bez spiska argumenata i da sve veze sa okruženjem ostvaruje preko globalnih promenljivih.

**Primer.** Potrebno je napisati program kojim se izračunava izraz

$$D=\frac{\tanh(a\cdot n+b)}{\tanh^2(a^2+b^2)}-\frac{\tanh^2(a+m\cdot b)}{\tanh(a^2-b^2)}.$$

Za izračunavanje vrednosti *tanh*(*x*) koristiti potprogram kojim se ova funkcija izračunava prema jednoj od relacija:

$$\tanh=\frac{e^x-e^{-x}}{e^x+e^{-x}} \qquad \text{ili} \qquad \tanh=\frac{e^{2x}-1}{e^{2x}+1}.$$

Slede četiri moguća rešenja ovog problema u kojima se koriste različiti potprogrami sa i bez parametara. Rešenja su data u programskom jeziku Pascal.

(a) Rešenje funkcijskim potprogramom bez argumenata. Rezultat se dodeljuje imenu funkcije. Vrednost za *x* se unosi u funkciju preko globalnog parametra.

```
program TANH1;
   var  D,a,b,m,n,R1,R2,R3,x : real;
   function th: real;
     var R :real;
     begin
       R:=exp(2*x); th := (R-1)/(R+1)
     end;
   begin
     read(a,b,m,n);
     x:=a*n+b; R1:=th;    x:=a+m*b; R2:=th;
     x:=a*a+b*b; R3:=th;  x:=a*a-b*b;
     D:=R1/(R3*R3)-R2*R2/th;  writeln(D);
   end.
```

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

float D,a,b,m,n,R1,R2,R3,x;

float th()
{ float R;
   R=exp(2*x); return ((R-1)/(R+1));
}

void main()
{ scanf("%f%f%f%f",&a,&b,&m,&n);
  x=a*n+b;   R1=th();     x=a+m*b; R2=th();
  x=a*a+b*b; R3=th();     x=a*a-b*b;
  D=R1/(R3*R3)-R2*R2/th();  printf("%f\n",D);
}
```

(b) Rešenje pomoću procedure bez argumenata. Rezultat se dodeljuje globalnom parametrom. Takođe, i vrednost za *x* se unosi preko globalnog parametra.

```
program TANH2;
var  D,a,b,m,n,R1,R2,R3,x,y : real;
procedure th;
 begin
  y := exp(2*x);   y := (y-1)/(y+1)
end;
begin
   read(a,b,m,n);
   x := a*n+b;    th;   R1 := y;
   x := a+m*b;    th;   R2 := y;
   x := a*a+b*b;  th;   R3 := y;
   x := a*a-b*b;  th;
   D := R1/(R3*R3)- R2*R2/y; writeln(D);
end.
```

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

float D,a,b,m,n,R1,R2,R3,x,y;

void th()
{ y=exp(2*x);  y=(y-1)/(y+1);   }

void main()
{ scanf("%f%f%f%f",&a,&b,&m,&n);
  x=a*n+b;    th();   R1=y;     x=a+m*b;    th();   R2=y;
  x=a*a+b*b;  th();   R3=y;     x=a*a-b*b;  th();
  D=R1/(R3*R3)- R2*R2/y;   printf("%f\n",D);
}
```

(c) Program sa funkcijskim potprogramom sa argumentima. Rezultat se dodeljuje imenu funkcije. Vrednost za *x* se unosi pomoću parametra funkcije.

```
program TANH3;
var D,a,b,m,n : real;
function th(x: real):real;
  var R : real;
   begin
       R := exp(2*x);   th:= (R-1)/(R+1)
   end;
begin
  read(a,b,m,n);
  D := th(a*n+b)/(th(a*a+b*b)*th(a*a+b*b)) -
       th(a+m*b)*th(a+m*b)/th(a*a-b*b);
  writeln(D)
end.
```

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

float D,a,b,m,n;

float th(float x)
{ float R=exp(2*x);
  return ((R-1)/(R+1));
}

void main()
{ scanf("%f%f%f%f", &a,&b,&m,&n);
  D= th(a*n+b)/(th(a*a+b*b)*th(a*a+b*b)) -
```

```
       th(a+m*b)*th(a+m*b)/th(a*a-b*b);
  printf("%f\n",D);
}
```

(d) Rešenje pomoću procedure sa argumentima. Prvi argument je izlazni a drugi ulazni.

```
program TANH4;
var D,a,b,m,n,Rl,R2,R3,R4 :real;
procedure th(var y: real; x: real);
  begin
     y := exp(2*x); y := (y-1)/(y+1)
  end;
     begin
   read(a,b,m,n);
   th(Rl, a*n+b);   th(R2, a+m*b);  th(R3,a*a+b*b);  th(R4,a*a-b*b);
   D:=Rl/(R3*R3)-R2*R2/R4;   writeln(D)
end.
```

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

float D,a,b,m,n,Rl,R2,R3,R4;
void th(float *y, float x)
{  *y=exp(2*x); *y=(*y-1)/(*y+1); }

void main()
{ scanf("%f%f%f%f", &a,&b,&m,&n);
  th(&Rl, a*n+b);   th(&R2, a+m*b); th(&R3,a*a+b*b);  th(&R4,a*a-b*b);
  D=Rl/(R3*R3)-R2*R2/R4;  printf("%f\n",D);
}
```

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

float D,a,b,m,n,Rl,R2,R3,R4;
void th(float &y, float x)
{  y=exp(2*x); y=(y-1)/(y+1); }

void main()
{ scanf("%f%f%f%f", &a,&b,&m,&n);
  th(Rl, a*n+b);   th(R2, a+m*b);  th(R3,a*a+b*b);  th(R4,a*a-b*b);
  D=Rl/(R3*R3)-R2*R2/R4;  printf("%f\n",D);
}
```

U programskom jeziku FORTRAN konceptu globalnih promenljivih na neki način odgovara koncept COMMON područja. Sve promenljive koje su zajedničke za više programskih modula obuhvataju se sa COMMON i pamte se u zasebnom memoriskom bloku koji je dostupan svim potprogramima koji se referenciraju na isto COMMON područje.

## Primeri

**Primer. (C)** Za paran broj *N* proveriti hipotezu Goldbaha. Prema toj hipotezi, svaki paran broj veći od 2 može se predstaviti zbirom dva prosta broja.

Rešenje se sastoji u proveri da li je za svaki prost broj *i* (*i* = 3, ... , *n*/2) broj *n-i* takođe prost. Da li je broj prost proverava se funkcijom *prost*.

```
#define nepar(x) ((x)%2)?1:0)
#include "math.h"

 int prost(long n)
  {  long i,koren;   int q;
     koren=floor(sqrt(n));   q=(nepar(n) || (n==2);
```

```
     i=3;
     while ((q) && (i<=koren)) {  q=((n % i) != 0); i=i+2;   };
     return (q);
  };

 void main()
   {long n,i;
    int q;
    do  { printf("\nunesite paran broj: ");     scanf("%ld",&n);     }
    while((n<4) || ((n % 2) != 0));
    i=2;   q=0;
    while ((i<= n/2) && (!q))
     { q=(prost(i) && prost(n-i));
       if (q) printf("\nTo su brojevi %ld i %ld",i,n-i);
       i++;
      };
    if (!q)     printf("\n Hipoteza ne vazi za %ld ",n);
 }
```

**Primer.** Napisati funkcijski potprogram koji izračunava *n*-ti stepen broja *x* koristeći relaciju

$$x^n = \begin{cases} 1, & n = 0 \\ (x^k)^2, & n=2k \\ x(x^k)^2, & n=2k+1 \end{cases}$$

Napisati program za stepenovanje koji u beskonačnom ciklusu učitava realan broj $x$ i prirodan broj $n$ i izračunava $x^n$.

```
program Stepenovanje2;

  type PrirodniBroj = O..Maxint;

  var  Pi, PiKvadrat : Real ;

  function Stepen(Osnova:Real; Eksponent:PrirodniBroj): Real;
   var Rezultat:Real;
   begin
     Rezultat := 1;
     while Eksponent > 0 do
       begin
         while not Odd(Eksponent) do
           begin
             Eksponent := Eksponent div 2; Osnova := Sqr(Osnova)
           end;
         Eksponent:= Eksponent-1; Rezultat := Rezultat*Osnova
       end;
      Stepen:=Rezultat
   end { Stepen };

begin
 repeat
    readln(x);  readln(n);
    Writeln(x:11:6, n:3, Stepen(x,n):11:6);
 until false
end { Stepenovanje2 }.
```

*Test primeri:*

```
   2.000000   7   128.000000
   3.141593   2    9.869605
   3.141593   4    97.409100
```

**Primer. (PASCAL)** Napisati funkciju koja izračunava vrednost verižnog razlomka

$$v(m)=\cfrac{1}{1+\cfrac{1}{2+\cfrac{1}{\cfrac{...}{m-2+\cfrac{1}{m-1+\cfrac{1}{m}}}}}}$$

Napisati program kojim se izračunava suma

$$S=1-\frac{1}{v(3)}+\frac{1}{v(5)}-...+(-1)^n\frac{1}{v(2n+1)}.$$

```
program verizni_razlomci;
  var s,znak:real;
      i,n:integer;
  function verizni(m:integer):real;
    var s:real;
        i:integer;
    begin
      s:=0;
      for i:=m downto 1 do s:=1/(s+i);
      verizni:=s;
    end;

  begin
    readln(n);
    s:=1.0; znak:=-1;
    for i:=1 to n do
      begin
        s:=s+znak/verizni(2*i+1);       znak:=-znak;
      end;
    writeln(s);
  end.
```

**Primer.** Naći najveći, najmanji i drugi po veličini element između unetih realnih brojeva, bez korišćenja nizova.

```
 program najveci;
  var max1,max2,min,x:real;
      n,i:integer;
  begin
    write('Koliko brojeva? '); readln(n);
    if n>2 then
       begin
         writeln('Unesi ',n,' brojeva');    read(x);
         max1:=x; min:=x;     read(x);
         if x>=max1 then
           begin
             max2:=max1; max1:=x
           end
         else if x>=min then        max2:=x
         else begin
                 max2:=min; min:=x;
               end;
         for i:=3 to n do
           begin
              read(x);
              if x>=max1 then
               begin
                  max2:=max1; max1:=x
               end
             else if x>max2 then        max2:=x
```

```
            else if x<min then         min:=x;
          end;
        writeln('Najveci je ',max1:10:6);
        writeln('Drugi po velicini je ',max2:10:6);
        writeln('Najmanji je ',min:10:6);
      end
    else writeln('Manjak brojeva')
  end.
```

**Primer.** Šta se ispisuje posle izvršenje sledećeg programa?

```
program UzgredniEfekat(Output);

  var A, Z : Integer ;

  function Skriven(X:Integer):Integer;
   begin
     Z := Z-X { uzgredni efekat na Z};
     Skriven := Sqr(X)
   end { Skriven } ;

  begin
    Z := 10; A := Skriven(Z);   Writeln(A,'  ', Z);
    Z := 10 ; A := Skriven(10) ; A := A*Skriven(Z);
    Writeln(A,'   ', Z) ;
    Z := 10; A := Skriven(Z); A := A*Skriven(10);
    Writeln(A,'   ', Z)
  end { UzgredniEfekat}
```

```
#include<stdio.h>
int a,z;
int Skriven(int x)
{     z=z-x; // uzgredni efekat na z
      return x*x;
}
void main()
{     z=10; a=Skriven(z); printf("%d  %d\n",a,z);
      z=10; a=Skriven(10); a*=Skriven(z);
      printf("%d   %d\n",a,z);
} // { UzgredniEfekat}
```

Rezultat:

```
100  0
0   0
10000   -10
```

# 5.5. Rekurzivni potprogrami

Funkcija, odnosno procedura je rekurzivna ako sadrži naredbu u kojoj poziva samu sebe. Rekurzija može biti direktna, kada funkcija direktno poziva samu sebe. Takođe, rekurzija može biti i indirektna, kada funkcija poziva neku drugu proceduru ili funkciju, koja poziva polaznu funkciju (proceduru).

U realizaciji rekurzije moraju se poštovati dva osnovna principa:
- Mora da postoji rekurentna veza između tekućeg i prethodnog (ili narednog) koraka u izračunavanju.
- Mora da postoji jedan ili više graničnih uslova koji će prekinuti rekurzivno izračunavanje.

Neki programski jezici dozvoljavaju poziv potprograma u telu samog tog potprograma. Takvi potprogrami nazivaju se rekurzivnim i pogodno su sredstvo za rešavanje problema koji su rekurzivno definisani.

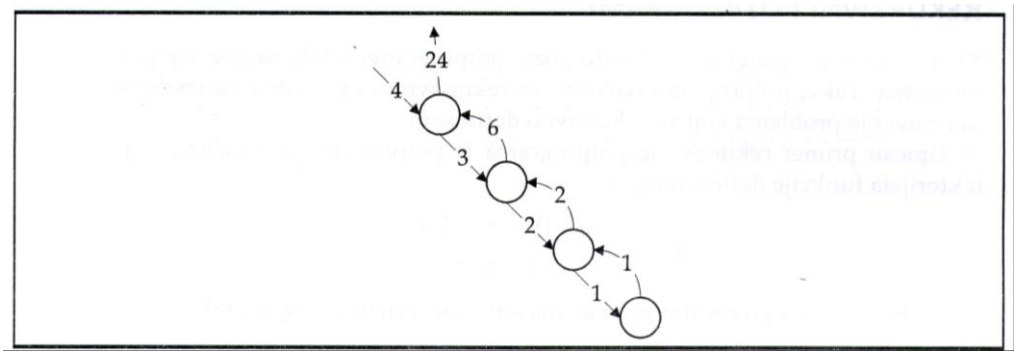
Tipičan primer rekurzivnog potprograma je potprogram za izračunavanje faktorijela funkcije definisanog sa:

$$n! = \begin{cases} n*(n\text{-}1)!, & \text{za } n>0 \\ 1, & \text{za } n = 0 \end{cases}$$

**Primer.** Program za izračunavanje faktorijela realizovan u Pascal-u:

```
function FAKT(N:integer): integer;
  begin
     if N>0 then  FAKT := N*FAKT(N-1)
            else  FAKT := 1
  end;
```

Koncept rekurzivnih potprograma se najbolje može sagledati pomoću grafa poziva potprograma na kome se predstavlja svaki od poziva potprograma. Na slici je prikazan graf poziva potprograma *FAKT* za $N = 4$.



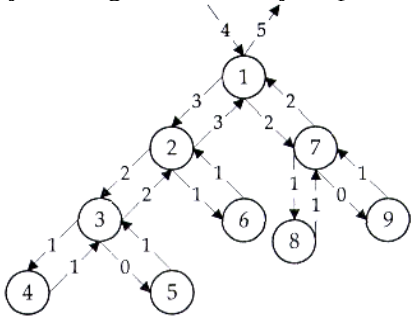

U praksi se često sreću i problemi u kojima postoji višestruka rekurzija. Jedan od takvih je problem izračunavanja članova Fibonačijevog niza brojeva. Fibonačijeva funkcija je definisana kao:

$$f(n) = \begin{cases} f(n\text{-}1) + f(n\text{-}2), & n>1 \\ 1, & n=1 \\ 1, & n = 0 \end{cases}$$

Potprogram za izračunavanje članova Fibonačijevog niza biće dvostruko rekurzivan. Dajemo implementaciju u Pascal-u:

```
function Fib(n: Integer):   integer;
  begin
    if n > 1 then
       Fib := Fib(n-1) + Fib(n-2)
    else  Fib  := 1;
  end;
```

U ovom potprogramu rekurzija dakle nije u stvaranju jedne, već dve kopije iste funkcije. Moguće su i trostruke, cetvorostruke, itd. rekurzije jer broj poziva kopija ničim nije ograničen. Graf poziva ovakve dvostruke rekurzije razlikuje se od grafika rekurzija iz prethodnog primera.




Još jedan od klasičnih rekurzivnih problema je čuveni problem Hanojskih kula.

**Primer.** Hanojske kule.

```
procedure Hanoj (n,sa,na,preko : integer);
 begin
   if n > 0 then
     begin
       Hanoj(n-1, sa preko, na);
       Write(sa, ' ->', na);
       Hanoj(n-1, preko, na, sa);
     end
 end;
```

Prema staroj indijskoj legendi, posle stvaranja sveta je Bog Brama (Brahma) postavio tri dijamantska stuba i na prvi postavio 64 zlatna prstena različitih prečnika tako da svaki naredni bude manji od prethodnog. Sveštenici hrama moraju da prebacuju te prstenove sa prvog na treći stub koristeći pri tome drugi, all samo jedan po jedan i to tako da se veći prsten ne može naći iznad manjeg. Kad svi prstenovi budu prebačeni na treći stub nastupiće kraj sveta.

Ovde ce biti prikazan primer programa koji vrši ovu operaciju prebacivanja i koji ne zavisi od broja prstenova. Međutim uobičajeno je da se ovaj primer izvodi za manji broj krugova 3 do 5 već samim tim što je najmanji broj potrebnih poteza $2^n$ - 1. Za slučaj sa 64 kruga dolazimo do broja od 18.446.744.073.709.551.615 poteza.

U opstem slučaju, međutim, problem se sastoji u tome da se n prstenova prebaci sa prvog stuba (1) na treći stub (3) preko drugog stuba (2). Cilj je, dakle, ostvarljiv u tri "koraka". Sve prstenove osim najvećeg (*n*-tog), prebaciti na drugi stub, koristeći treći stub. Zatim *n*-ti prsten prebaciti na treći stub, a onda na njega staviti pomoćnu gomilu sa drugog stuba, koristeći prvi stub kao pomoćni. Da bi se ovo izvelo potrebno je ceo postupak izvesti za *n* -1 prsten, a za to je potrebno izvršiti istu proceduru za *n* - 2 prstena itd. Tako dolazimo do rekurzije.

Procedura se poziva za naprimer *Hanoj*(3,1,3,2).

Razmotrićemo još primer rekurzivnog potprograma za generisanje svih mogućih permutacija bez ponavljanja skupa sa *n* elemenata $P_{(n)}$, dat kao prethodni primer.

## 5.5.1. Primeri rekurzivnih funkcija u C

**Primer.** Jedmostavni primeri rekurzivnih funkcija.

```
#include <stdio.h>
void fun1(int);
void fun2(int);
void fun3(int);
void main()
 { printf("\n fun1(5)\n");   fun1(5);
   printf("\n fun2(5)\n");   fun2(5);
   printf("\n fun3(5)\n");   fun3(5);
 }

void   fun1(int n)
  { printf("%2d",n);    if(n) fun1(n-1);  return;  }

void fun2(int n)
 { if(n) fun2(n-1);   printf("%2d",n);   return;  }

void fun3(int n)
  { printf("%2d",n);   if(n) fun3(n-1);   printf("%2d",n);  return;
  }
```

Test primeri:

```
      fun1(5)
      5 4 3 2 1 0
```

```
fun2(5)
0 1 2 3 4 5

fun3(5)
5 4 3 2 1 0 0 1 2 3 4 5
```

**Primer.** Rekurzivno izračunavanje faktorijela.

```
#include <stdio.h>
long fact(short);
void main()
  { short n;     long rezultat;
    scanf("%d",&n);     rezultat = fact(n);
    printf("Faktorijel od %d = %ld\n", n,rezultat);
 }

long fact(short n)
 { if(n<=1)return(1L);    else return(n*fact(n-1));  }
```

**Primer.** Napisati rekurzivnu funkciju za izračunavanje *n*-tog člana Fibonačijevog niza:

$$fib(n)=\begin{cases} 1, & n=1 \text{ ili } n=0, \\ \text{fib}(n\text{-}1)+\text{fib}(n\text{-}2), & n>1\ . \end{cases}$$

```
#include <stdio.h>
long fib(short);
void main()
 {  short n;
    scanf("\n %d", &n);
    printf("%d-ti clan Fibonacijevog niza = %ld\n",  n,fib(n));
  }

long fib(short n)
  { if(n<=1)return((long)1); else return(fib(n-1)+fib(n-2));}
```

Broj rekurzivnih poziva pri izračunavaju Fibonačijevih brojeva može se smanjiti na sledeći način:

```
long fib(long a, long b, short n);
  { if(n==2)return b;
    else { b=a+b; a=b-a; n--;  return fib(a,b,n);  }
  }
```

**Primer.** Zadati broj tipa `int` ispisati znak po znak.

```
#include <stdio.h>
void printd(int n)
  { if(n<0)        { putchar('-');  n=-n; }
     if(n/10) printd(n/10);
     putchar(n%10+'0');
  }
void main()
   {int n;  scanf("%d", &n);    printd(n);  }
```

**Primer.** Rekurzivno izračunavanje najvećeg zajedničkog delioca dva cela broja.

```
#include <stdio.h>
int nzd(int n, int m)
  { if(n==m) return(m);
    if(n<m) return(nzd(n,m-n));
    else return(nzd(n-m,m));
  }
void main()
  { int p,q;
    scanf("%d%d",&p,&q);  printf("nzd(%d,%d)=%d\n",p,q,nzd(p,q));
```

```
 }
```

**Primer.** Rekurzivno izračunati Akermanovu funkciju $A(n,x,y)$, koja je definisana na sledeći način:

$A(n, x, y) = x+1$, za $n=0$
$A(n, x, y) = x$, za $n=1, y=0$,
$A(n, x, y) = 0$, za $n=2, y=0$
$A(n, x, y) = 1$, za $n=3, y=0$
$A(n, x, y) = 2$, za $n>3, y=0$
$A(n, x, y) = A(n-1, A(n, x, y-1), x)$, inače.

```
#include <stdio.h>
long aker(int,int,int);
void main()
  { int x,y,n;
     printf("Unesit n"); scanf("%d",&n);     printf("Unesit x");
     scanf("%d",&x);     printf("Unesit y"); scanf("%d",&y);
     printf("A(%d,%d,%d)=%ld\n",n,x,y,aker(n,x,y));
 }
long aker(int n, int x, int y)
  { int pom;
     if(n==0)return(x+1);    if(n==1 && y==0) return(x);
     if(n==2 && y==0) return(0);    if(n>3 && y==0) return(2);
     else { pom=aker(n,x,y-1);  return(aker(n-1,pom,x)); }
  }
```

**Primer.** Koristeći rekurzivnu definiciju proizvoda dva prirodna broja napisati odgovarajuću funkciju.

```
#include <stdio.h>
long p(int a, int b)
  { if(b==1) return(a); return(a+p(a,b-1));  }
void main()
  { int x,y;
     printf("Unesi dva prirodna broja ");     scanf("%d%d",&x,&y);
     printf("%d*%d=%ld\n",x,y,p(x,y));
 }
```

**Primer.** Napisati rekurzivni program za izračunavanje sume cifara zadatog prirodnog broja.

```
#include <stdio.h>
int sumacif(long n)
  { if(!n)return(0);     else    return(n%10+sumacif(n/10));    }

void main()
  {  long k;
       printf("Unesi dug ceo broj "); scanf("%ld",&k);
       printf("Suma cifara od %ld=%d\n",k,sumacif(k));
  }
```

**Primer.** Napisati rekurzivni program za izračunavanje količnika dva prirodna broja na proizvoljan broj decimala.

```
#include <stdio.h>
void kolicnik(int ind, int n, int m, int k)
  { if(ind) printf("%d.",n/m);
    else printf("%d",n/m);
     if(k) kolicnik(0,(n%m)*10,m,k-1);  }

void main()
    { int n,m,k;
      printf("Brojilac? "); scanf("%d",&n);
      printf("Imenilac? "); scanf("%d",&m);
```

```
      printf("Broj decimala? "); scanf("%d",&k);
      kolicnik(1,n,m,k);
    }
```

**Primer.** Hanojske kule.

```
#include <stdio.h>
void main()
  { int bd;    void prebaci();
     printf("Broj diskova = ? "); scanf("%d",&bd);
     prebaci(bd,1,3,2);
   }

void prebaci(int n, int sa, int na, int preko)
 { if(n) {  prebaci(n-1,sa,preko,na);
            printf("%d->%d  ",sa,na); prebaci(n-1,preko,na,sa);
 }
```

**Primer.** Sa tastature se učitava bez greške formula oblika

```
<formula>::=<cifra>|M(<formula>,<formula>)|m(<formula>,<formula>)
<cifra>::=0|1|2|3|4|5|6|7|8|9
M- maksimum,  m-minimum.
```

Napisati rekurzivnu funkciju kojim se izračunava vrednost izraza.

Na primer, `M(4,m(6,9))=6`.

```
#include<iostream.h>
#include<stdio.h>
int f()
{ char ch,z;
  int x,y;
  ch=getchar();  if(ch>='0' && ch<='9') return ch-'0';
  //sada je ch= M ili m
  z=getchar(); // cita otvorenu zagradu
  x=f();
  z=getchar(); // cita zarez
  y=f();
  z=getchar(); // cita zatvorenu zagradu
  switch(ch)
  { case 'M': return (x>y)?x:y;
    case 'm': return (x>y)?y:x;
  }
}

void main()
{ cout<<"----> "<<f()<<endl;  }
```

# 5.6. Implementacija potprograma

Prilikom prevođenja programa kompilator vrši planiranje memorijskog prostora koji će biti dodeljen programu (*Storage allocation*). Pored toga što treba da se rezerviše memorijski prostor za smeštaj kôda programa i podataka koji se obrađuju u njemu, za svaki od potprograma koji se poziva iz programa generiše se i jedan *aktivacioni slog* u koji se smeštaju podaci koji se preuzimaju od glavnog programa, lokalni podaci potprograma i takozvane privremene promenljive koje generiše kompilator prilikom generijsanja mašinskog koda. Struktura jednog aktivacionog sloga data je na slici.

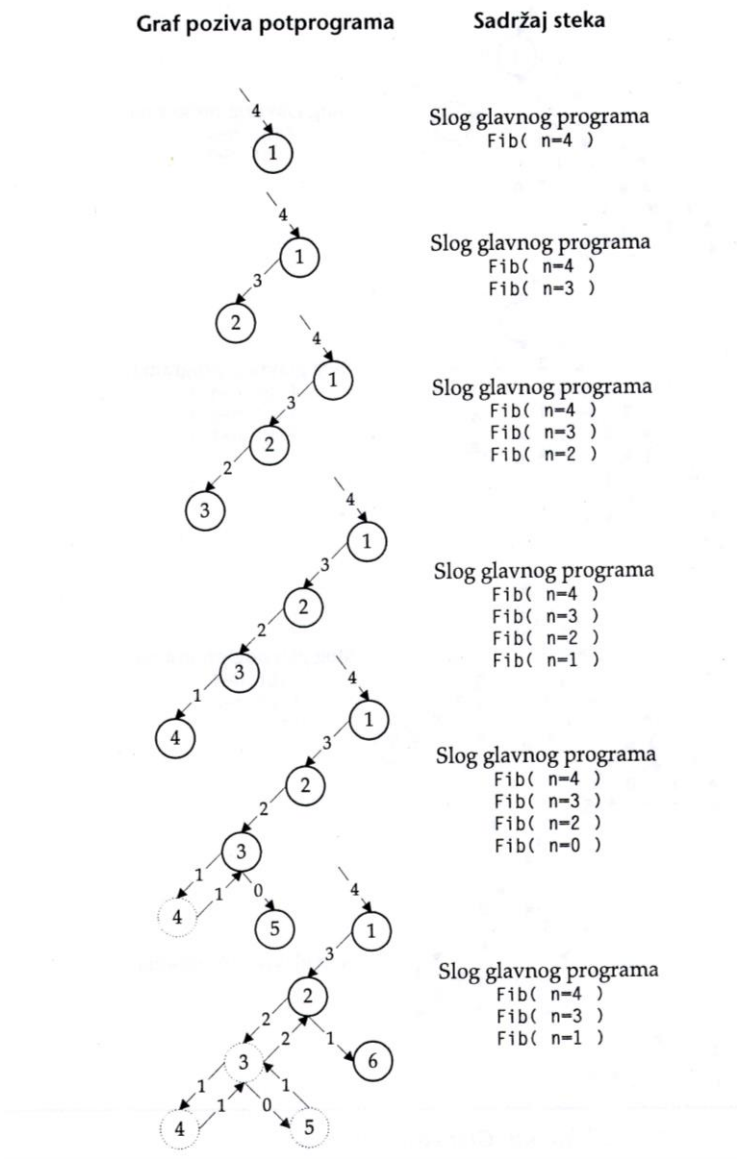
| Rezultati koje vraća potprogram |
|---|
| Stvarni argumenti |
| Opcioni upravljački linkovi aktivacionog sloga glavnog programa |
| Opcioni linkovi za pristup podacima u drugim aktivacionim slogovima |

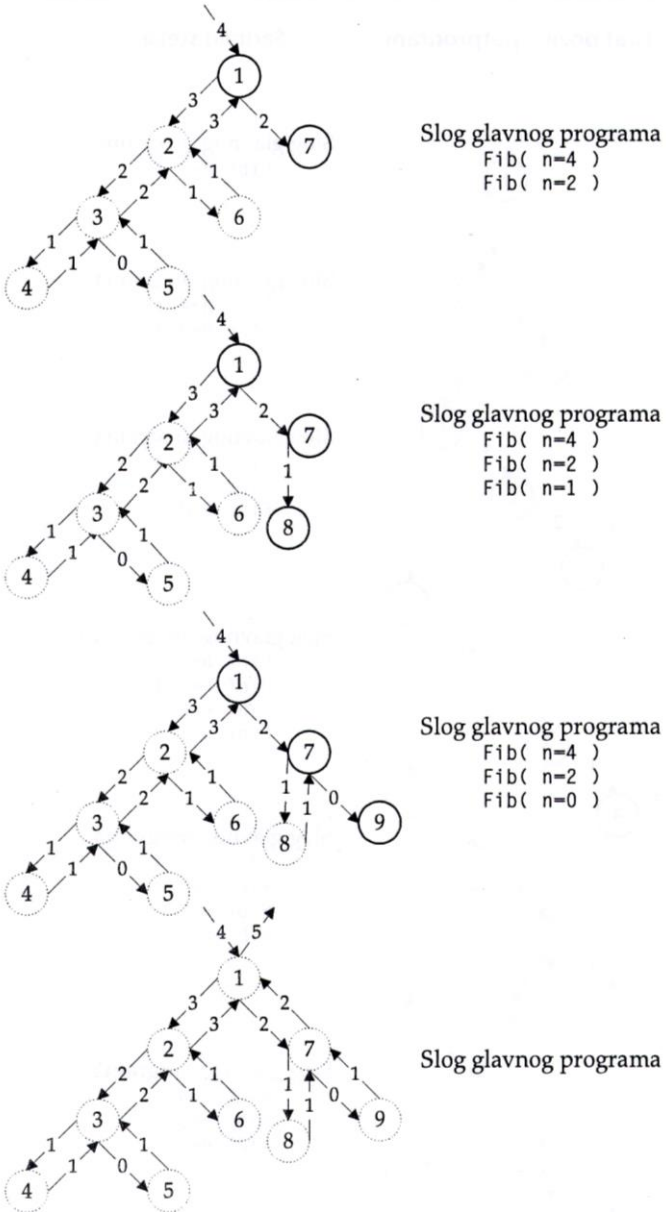
| Podaci o statusu |
| --- |
| Lokalni podaci |
| Privremene promenljive koje generiše kompilator |

Za smeštaj aktivacionih slogova koristi se jedna od sledeće tri strategije: statička, pomoću steka i dinamička. U zavisnosti od toga koja je strategija primenjena zavisi da li će ill ne biti mogući rekurzivni pozivi potprograma.

Kod statičkog smeštanja aktivacionih slogova, kompilator unapred rezerviše fiksan memorijski prostor čiji se sadrzaj obnavlja kod svakog poziva potprograma. To znači da kod ovakvog smestanja nisu mogući rekurzivni pozivi potprograma jer se kod svakog novog poziva potprograma gubi informacija o prethodnom pozivu. Ovakva tehnika primenjuje se na primer kod realizacije kompilatora za programski jezik FORTRAN, pa se u njemu zbog toga ne mogu koristiti rekurzije.

Kod strategije koja se zasniva na primeni steka za smeštaj aktivacionih slogova koristi se stek u koji se za svaki poziv potprograma smešta jedan aktivacioni slog. Ova tehnika dozvoljava rekurziju jer se kod svakog novog poziva potprograma generiše novi slog i smešta u stek. Na kraju potprograma slog se izbacuje iz steka. To znači da je u toku izvršavanja programa ovaj stek dinamički zauzet aktivacionim slogovima onih potprograma koji su trenutno aktivni. Od veličine memorijskog prostora koji je dodeljen steku zavisi i dubina rekurzivnih poziva potprograma. Na slici ispod prikazana je promena strukture steka u toku izvršavanja programa u kome se poziva rekurzivni potprogram za izračunavanje Fibonačijevih brojeva za $n = 4$.


Graf poziva potprograma
Sadržaj steka
Slog glavnog programa
Fib( n=4 )
Slog glavnog programa
Fib( n=4 )
Fib( n=3 )
Slog glavnog programa
Fib( n=4 )
Fib( n=3 )
Fib( n=2 )
Slog glavnog programa
Fib( n=4 )
Fib( n=3 )
Fib( n=2 )
Fib( n=1 )
Slog glavnog programa
Fib( n=4 )
Fib( n=3 )
Fib( n=2 )
Fib( n=0 )
Slog glavnog programa
Fib( n=4 )
Fib( n=3 )
Fib( n=1 )




Treća, dinamička strategija sastoji se u tome da se za svaki poziv potprograma generiše aktivacioni slog koji se smešta u poseban deo memorije nazvan *Heap*. U ovom slučaju aktivacioni slogovi potprograma povezuju se u dinamičku strukturu podataka koja odgovara stablu poziva potprograma. Očigledno je da i ova tehnika dozvoljava rekurziju.

# 5.7. Scope rules (domen važenja)

Komponovana naredba (blok) je serija deklaracija iza koje sledi serija naredbi između zagrada { i }. Funkcije se mogu tretirati kao imenovani blokovi sa parametrima i dozvoljenom return naredbom.

Bazično pravilo domena važenja je da su identifikatori dostupni samo u bloku u kome su deklarisani. Jedno ime u spoljašanjem bloku važi dok se ne redefiniše u unutrašnjem. Tada je ime u spoljašnjem bloku skriveno (ili maskirano) tim imenom unutrašnjeg bloka.

**Primer.**

```
 /* spoljasnji blok */
int a=2;    printf("%d\n",a)   /* 2 */
/* unutrašnji blok */
int a=3;       print("%d\n",a);       /* 3 */
```

```
    printf("%d\n",a);          /*2 */
```

# 5.8. Memorijske klase u C

Svaka promenljiva i funkcija u C ima dva atributa: tip i memorijsku klasu. Memorijske klase su definisane ključnim rečima *auto, extern, static* ili *register*

Promenljive deklarisane unutar tela funkcije jesu, podrazumevano (by default), memorijske klase *automatic*. promenljive ove klase deklarišu se eksplicitno ključnom rečju *auto*.

**Primer.** Deklaracija unutar nekog bloka

char c; int i,j,k; …

je ekvivalentna sa

auto char c; auto int i,j,k; …

Kada je blok unet, sistem rezerviše adekvatnu memoriju za promenljive klase *auto*. Ove promenljive se tretiraju kao "lokalne" za blok. Kada se izađe iz bloka sistem više ne pamti vrednosti ovih promenljivih.

Sve funkcije i promenljive deklarisane izvan tela funkcija imaju memorijsku klasu *external*, i one su globalne za sve funkcije deklarisane posle njih.

**Primer.**

```
#include<stdio.h>
int a=7;
void main()
 { void f(void);
   void g(void);
     { printf("%d\n",a);              /* 7 */
       f();  printf("%d\n",a);        /* 8 */
       g();  printf("%d\n",a);        /* 8 */
 }

void f()
    { printf("%d\n",a);               /*7 */
      a++;     printf("%d\n",a);     /* 8 */
  }

void g()
  { int a=10;
     printf("%d\n",a);               /* 10 */
  }
```

Promenljiva je globalna na nivou modula ako je deklarisana van svih funkcija u modulu. memorijski prostor se trajno dodeljuje, a memorijska klasa te promenljive je *extern*. Globalnost je određena mestom deklaracije. Globalna promenljiva može da bude maskirana lokalnom promenljivom istog imena.

Da bi globalna promenljiva bila dostupna iz drugih modula, ona mora da se u tim modulima definiše pomoću ključne reči *extern*.

**Primer.** Izrazom oblika

extern int a;

promenljiva a se ne deklariše, već definiše, tj. ne dodeljuje se memorijski prostor, nego se C prevodilac informiše o tipu promenljive. Eksterna promenljiva se može inicijalizovati isključivo na mestu svoje deklaracije.

**Primer.** Dat je sledeći program koji se nalazi u dve datoteke (dva modula). U datoteci modul1.c se nalazi program

```
#include <stdio.h>
#include "modul2.c"
char c='w';
void main()
{ void f();void g();
  printf("pre funkcije f: c= %c\n",c);
  f();   printf("posle funkcije f: c= %c\n",c);
  g();   printf("posle funkcije g: c= %c\n",c);
}
```

Promenljiva *c* je globalna, pa se njoj može pristupiti iz drugih modula pomoću ključne reči *extern*. Neka je datoteka *modul*2.c sledećeg sadržaja

```
extern char c;
void f()
{ c = 'a'; }

void g()
{ char c='b'; }
```

Dobijaju se sledeći rezultati:

```
Pre funkcije f: c=w
Posle funkcije f: c=a
Posle funkcije g: c=a
```

Funkcija *f* menja vrednost eksterne promenljive *c*, a to mogu i druge funkcije iz ove datoteke (modula).

Ako datoteka *modul*2.c ima sadržaj

```
f()  { extern char c;  c='a';  }
```

takođe se menja vrednost promenljive *c*. Međutim, to nebi mogle da učine druge funkcije iz datoteke *modul*2.*c*.

Statičke promenljive se koriste u dva slučaja. U prvom slučaju, omogućava lokalnoj promenljivoj da zadrži vrednost kada se završi blok u kome je deklarisana. Druga primena je u vezi sa globalnom deklaracijom, i omogućuje mehanizam privatnosti globalnih promenljivih.

**Primer.** (1.primena)

```
#include<stdio.h>
void fun1()
 { static int x=0;    int y=0;
   printf("static=%d auto = %d\n",x,y);     ++x,++y;
 }
void main()
{ int i;    for(i=0; i<3; ++i) fun1();  }
```

Izlaz je:

```
static=0  auto=0
static=1  auto=0
static=2  auto=0
```

U vezi sa drugom primenom, koristi se činjenica da su statičke promenljive lokalne u okviru modula, jer im mogu pristupiti samo funkcije iz istog modula.

Pored promenljivih, i funkcije mogu da se definišu kao *extern* ili *static*. Memorijska klasa *extern* se uzima po definiciji. Statičke funkcije su dostupne samo funkcijama iz istog modula.

Korišćenje registarskih promenljivih omogućava programeru da utiče na efikasnost izvršenja programa. Ako funkcija često koristi neku promenljivu, može se zahtevati da se njena vrednost memoriše u brzim registrima centralne procesorske jedinice (CPU), uvek kada se funkcija izvršava. Registarske promenljive su najčešće promenljive za kontrolu petlje i lokalne promenljive u

funkcijama. Promenljiva se može učiniti registarskom pomoću ključne reči register.

**Primer.**
```
register char c;
register int i;
```

Promenljiva deklarisana kao registarska takođe je i automatska. Ako nema slobodnih registara u *CPU*, C prevodilac ne prepoznaje grešku.

**Primer.** Promenljiva *i* deklariše se kao registarska neposredno pre upotrebe u *for* petlji.
```
register int i;
for(i=0;i<5;++i)  ...
```
Završetak bloka u kome je deklarisana registarska promenljiva oslobađa registar.

### 5.8.1. Životni vek objekata

Životni vek objekta: vreme u toku izvršavanja programa u kojem objekat postoji i za koje mu se može pristupati. Na početku životnog veka, objekat se kreira, poziva se njegov konstruktor, ako ga ima. Na kraju životnog veka se objekat uništava, poziva se njegov destruktor, ako ga ima.

### 5.8.2. Vrste objekata po životnom veku

- Po životnom veku, objekti se dele na:
  statičke, automatske, dinamičke, tranzijentne (privremene).
- Vek atributa klase = vek objekta kome pripadaju.
- Vek formalnog argumenta = vek automatskog objekta.
  Formalni parametri se inicijalizuju vrednostima stvarnih argumenata.

### Statički i automatski objekti

- Automatski objekat je lokalni objekat koji nije deklarisan kao *static*.
  Životni vek: od njegove definicije, do napuštanja oblasti važenja.
  Kreira se iznova pri svakom pozivu bloka u kome je deklarisan.
  Prostor za automatske objekte se alocira na stack-u.
- Statički objekat je globalni objekat ili lokalni deklarisan kao *static*.
  Životni vek: od izvršavanja definicije do kraja izvršavanja programa.
  Globalni statički objekti:
  - kreiraju se samo jednom, na početku izvršavanja programa,
  - kreiraju se pre korišćenja bilo koje funkcije ili objekta iz istog fajla,
  - nije obavezno da se kreiraju pre poziva funkcije *main*(),
  - prestaju da žive po završetku funkcije *main*().

  Lokalni statički objekti počinju da žive pri prvom nailasku toka programa na njihovu definiciju.

**Primer.**
```
int a=1;
void f() {
  int b=1;           // inicijalizuje se pri svakom pozivu
  static int c=1;    // inicijalizuje se samo jednom
  cout<<"a="<<a++<<" b="<<b++<<" c="<<c++<<endl;
}
void main() {
  while (a<3) f();
}
```

```
izlaz:
a = 1 b = 1 c = 1
a = 2 b = 1 c = 2
```

# 6. STRUKTURNI TIPOVI PODATAKA

Pojam struktura podataka, nizova, slogova, skupova i datoteka je poznat i prisutan u višim programskim jezicima od samog njihovog nastanka. U okviru strukturnih tipova podataka, vrednosti koje tip obuhvata definisane su kao jednorodne ili raznorodne strukture podataka. U jeziku Pascal postoji mogućnost definisanja strukturnih tipova koji omogućavaju rad sa nizovima, slogovima, skupovima i datotekama. Skupovi se kao strukturni tipovi podataka ređe sreću kod drugih programskih jezika, dok su nizovi, slogovi i datoteke postali standardni koncept, prisutan u mnogim programskih jezicima (Ada, C).

## 6.1. Polja u programskim jezicima

Niz (Polje) (***array*)** je indeksirana sekvenca komponenti istog tipa. Polja u programskim jezicima jesu jednodimenzionalne ili višedimenzionalne strukture podataka koje obuhvataju više vrednosti istog tipa. Može se reći, da jednodimenzionalna polja odgovaraju pojmu vektora, a višedimenzionalna pojmu matrica. Bilo koji niz koji ima komponente nekog tipa *T* pri čemu su vrednosti indeksa iz nekog tipa *S* predstavlja preslikavanje $S \rightarrow T$. Dužina (***length***) niza jeste broj njegovih komponenti, koji se označava sa *#S*. Nizovi se mogu naći u svakom imperativnom i objektno orijentisanom jeziku. Tip *S* mora da bude konačan, tako da je niz konačno preslikavanje. U praksi, *S* je uvek rang uzastopnih vrednosti, što se naziva rang indeksa niza (array's ***index range***). Granice opsega indeksa nazivaju se donja granica (***lower bound***) i gornja granica (***upper bound***).

Kao i sve druge promenljive u programu, i niz se mora deklarisati pre nego što se upotrebi. Deklaracijom niza kompajleru se saopštavaju sledeći podaci o nizu: ime (koje je predstavljeno nekim identifikatorom), tip elemenata niza i veličinu niza (broj elemenata tog niza). Bazične operacije nad nizovima jesu:

• konstruisanje (***construction***) niza iz njegovih komponenti;

• indeksiranje (***indexing***), tj. selektovanje partikularne komponente niza, kada je zadat njegov indeks. Indeks koji se koristi za selekciju komponente niza jeste vrednost koja se izračunava.

Programski jezici C i C++ ograničavaju indeks niza na celobrojne vrednosti čija je donja granica 0. Opšti oblik deklaracije niza u jeziku C je

<tip elemenata> <ime tipa> [<celobrojni izraz>];

PASCAL omogućava da rang za indeks niza može biti odabran od strane programera, jedina restrikcija je rang indeksa mora da bude diskretan primitivni tip.

Za opis ovih struktura u nekim programskim jezicima se koristi ključna reč **array**. Takav je slučaj u jeziku PASCAL. Opšti oblik deklaracije niza u PASCALu je

array[<tip indeksa>] of <tip elemenata>;

Razmotrimo sledeći primer u jeziku PASCAL:

```
type VEKTOR = array [1..10] of  real;
var    A,B  : VEKTOR;
```

Ovom definicijom definisan je tip *VEKTOR*, kao niz od 10 elemenata tipa *real*. Promenljive *A* i *B* su deklarisane kao promenljive tipa *VEKTOR*. Svakoj od promenljivih *A* i *B* odgovara struktura podataka koja se sastoji od 10 komponenti tipa real. Svakoj od komponenti vektora opisanog na ovaj način može se pristupiti preko indeksa. U nekim jezicima (Pascal) koristi se i notacija sa uglastim zagradama na primer *A*[3] i *A*[*J*]. U ovakvim slučajevima indeks može da bude definisan i kao celobrojni izraz. Ove indeksirane promenljive mogu se u programu upotrebiti svuda gde se javljaju i proste promenljive odgovarajućeg tipa. U datom primeru *A*(*J*) i *B*(*J*) su promenljive tipa *real*. Evo nekih primera koji ilustruju upotrebu niza:

`A[3]:= 0.0;` - Trećem elementu vektora *A* dodeljuje se vrednost 0;

`X := A[3];` - promenljiva *X* dobija vrednost trećeg elementa vektora *A*.

Polja se realizuju kao statičke strukture podataka, što znači da se za svaki određeni vektor rezerviše unapred definisani prostor u memoriji računara. Granice indeksa u definiciji tipa mogu da budu date i preko promenljivih ili čak i preko izraza, ali ove vrednosti treba da budu poznate u tački programa u kojoj se pojavljuje definicija tipa. Evo nekih definicija takvog oblika u jeziku Pascal:

```
type VEKTOR1 = array [1 .. N]  of  float;
type VEKTOR2 = array [N .. N+M]  of  float;
```

U svim ovim primerima, za definisanje indeksa je iskorišćen interval u skupu celih brojeva (integer). To je i prirodno, jer se najčešće za indeksiranje i u matematici koriste celobrojne vrednosti. U Pascal-u i Adi se za definisanje indeksa može koristiti bilo koji diskretni tip podataka. Razmotrimo sledeći primer u kome se kreira tabela sa podacima o količini padavina u svakom mesecu jedne godine u jeziku Ada:

```
type PADAVINE is delta 0.1 range 0.0  ..  200.0;
type MESECI is JAN,FEB,MAR,APR,MAJ,JUN,JUL,AVG,SEP,OKT,NOV,DEC);
type KOL_PAD is array [MESECI] of  PADAVINE;
```

U programu se mogu koristiti promenljive tipa *KOL_PAD*.

```
PODACI: KOL_PAD;
```

Analogni tip podataka u jeziku Pascal je dat sledećim definicijama:

```
type MESECI = (JAN,FEB,MAR,APR,MAJ,JUN,JUL,AVG,SEP,OKT,NOV,DEC);
type KOL_PAD = array [MESECI] of  real;
var PODACI: KOL_PAD;
```

U ovom primeru definisan je strukturni tip podataka *KOL_PAD*, za čije indeksiranje se koristi diskretni tip nabrajanja *MESECI*, tako da vektor *PODACI* ima ukupno 12 komponenti, od kojih svaka odgovara jednom mesecu u godini. Svakoj od ovih komponenti pristupa se preko imena meseca kao indeksa. Na taj način, u programu će *PODACI*[*MART*] biti promenljiva kojoj se dodeljuje vrednost koja odgovara padavinama u mesecu martu. Diskretni tipovi podataka se mogu koristiti za definisanje opsega indeksa, tako što se u okviru skupa vrednosti određenog diskretnog tipa koristi samo interval vrednosti. Na primer, u razmatranom primeru moglo je da bude postavljeno ograničenje da se razmatraju padavine u letnjem periodu godine. U tom slučaju, pogodno je indeks vektora *PODACI* definisati tako da ima samo komponente koje odgovaraju letnjim mesecima.

## 6.2. Jednodimenzionalni nizovi u C

Deklaracija jednodimenzionalnog niza se sastoji od imena tipa iza koga sledi identifikator (ime niza) i na kraju celobrojni izraz između srednjih zagrada. Vrednost ovog izraza mora da bude pozitivan ceo broj i predstavlja veličinu niza. Indeksiranje elemenata niza počinje od 0.

Opšta forma izraza kojom se deklariše niz je sledeća:

<tip elemenata> <ime> [<celobrojni izraz>];

Veličina niza zadaje se konstantom ili konstantnim celobrojnim izrazom. Na primer, izrazom `int b[100]` deklariše se niz sa 100 celobrojnih vrednosti. Niz *b* se sastoji od 100 celobrojnih (indeksiranih) promenljivih sa imenima *b*[0], ..., *b*[99].

Komponentama niza se pristupa navođenjem imena niza i celobrojnog izraza između srednjih zagrada. Ovim izrazom se definiše indeks željenog elementa niza. Ako je indeks manji od 0 ili veći ili jednak od broja elemenata u nizu, neće se pristupiti elementu niza.

Nizovi mogu da imaju memorijsku klasu *auto*, *extern*, *static*, a ne mogu biti memorijske klase *register*.

Nizovi se mogu inicijalizovati u naredbi deklaracije.

**Primer.** Posle inicijalizacije

```
float x[7]={-1.1,0.2,33.0,4.4,5.05,0.0,7.7};
```

dobija se

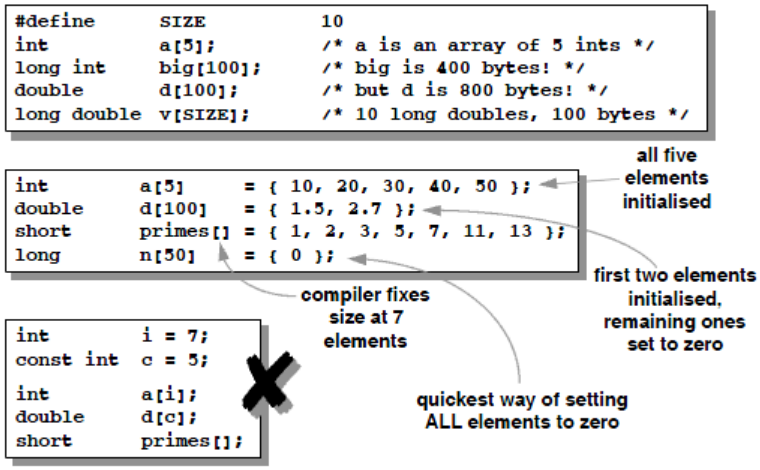
```
x[0]=-1.1, x[1]=0.2, … x[6]=7.7.
```

Lista elemenata koji se koriste za inicijalizaciju vektora može biti manja od broja njegovih elemenata. Ako je niz memorijske klase *static* ili *extern*, njegovi preostali elementi postaju 0. U slučaju da je niz memorijske klase *auto*, za ove vrednosti će biti neke vrednosti koje su zaostale od ranije u delu memorije koja se koristi za smeštanje elemenata niza. Ako *extern* ili *static* niz nije inicijalizovan, tada kompajler automatski inicijalizuje sve elemente na 0.

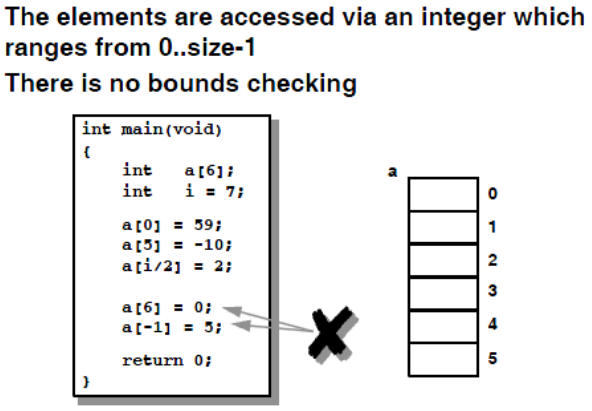
Ako je niz deklarisan bez preciziranja dužine, i pritom inicijalizovan, tada se dužina niza implictno određuje prema broju inicijalizatora.

Na primer, izraz int `a[]={3,4,5,6};` proizvodi isto dejstvo kao i izraz
`int a[4]={3,4,5,6};`

**Primer.** Deklaracije nizova.




**Primer.** Pristup elementima nizova.




## Primeri.

**Primer.** Izračunati broj pojavljivanja svakog velikog slova u unetom tekstu.

```
#include <stdio.h>
#include <ctype.h>
void main()
 { int c,i, slova[26];
    for(i=0;i<26;++i) slova[i]=0;
    while((c=getchar()) !=EOF)
       if(isupper(c))    ++slova[c-'A'];
    for(i=0;i<26;++i)
```

```
       { if(i%6==0) printf("\n");
         printf("%5c:%4d", 'A'+i, slova[i]);
       }
    printf("\n\n");
 }
```

**Primer.** Napisati program kojim se izračunava *n*-ti stepen broja 2, gde je *n*<= 500, i traženi stepen nema više od 200 cifara.

```
void main()
 { int i,n,d=0,j,p, x[200]; // d+1 je broj cifara u stepenu, p je prenos
    printf("\n Unesi stepen --> "); scanf("%d",&n);
    x[0]=1;
    for(i=0; i<n; i++)
      { p=0;
        for(j=0; j<=d; j++){x[j]=x[j]*2+p; p=x[j]/10; x[j] %=10; }
        if(p!=0) { d++; x[d]=p; }
      }
    printf("%d. stepen broja 2 je ",n);
    for(i=d; i>=0; i--)printf("%d",x[i]);
 }
```

**Primer.** Binarno traženje.

Posmatra se sledeći matematički problem: zadat je uređen realni niz $a[0]<a[1]<\cdots<a[n-1]$ i realan broj $b$; ustanoviti da li se $b$ nalazi u nizu, i ako se nalazi odrediti indeks $p$ za koji važi a[$p$]=b.

Najjednostavnije, ali i najneefikasnije je takozvano linearno pretraživanje: redom se upoređuju elementi niza $a$ sa brojem $b$, do prvog elementa niza $a$ za koji je $a[i]>=b$. Ako je $a[i]=b$, tada je $p = i$ traženi indeks. Ako je $a[i]>b$, tada se broj $b$ ne nalazi u nizu $a$.

Za brzo pretraživanje se koristi algoritam binarnog pretraživanja.

Pretpostavimo da postoji $p\in[0, n-1]$ takav da je $a[p] = b$. Izaberemo srednji element niza a sa indeksom $s = (0+n-1)/2$. Ako je $a[s]$=b, tada je p=s traženi indeks, a pretraživanje se prekida. Ako je ispunjen uslov $b < a[s]$, tada se indeks p nalazi u intervalu $[0,s-1]$, a inače se nalazi u intervalu $[s+1,n-1]$. U oba slučaja, prepolovljen je interval pretraživanja.

Napisati funkciju koja prema opisanom algoritmu određuje indeks onog elementa rastućeg niza $a$ koji je jednak zadatoj vrednosti $b$, a inače vraća rezultat -1.

```
#include<stdio.h>
void main()
 { float br, a[100];
   int n,i,p;
   int bintra(float a[], int n, float b);
    printf("\nBroj elemenata? "); scanf("%d",&n);
    printf("Elementi niza ");    for(i=0; i<n; i++)scanf("%f", &a[i]);
    printf("Jedan realan broj? "); scanf("%f",&br);
    p=bintra(a, n, br);
    printf("Pozicija broja %f u nizu je %d\n",br,p);
 }

int bintra(float a[], int n, float b)
 { int l,d,s;
    l=0; d=n-1;
    while(l<=d)
     { s=(l+d)/2;
       if(b==a[s]) return(s);
       else if(b<a[s])d=s-1;
       else l=s+1;
     }
    return(-1);
```

```
 }
```

Odgovarajuća rekurzivna funkcija je

```
#include<stdio.h>
main()
 { float br, a[100];
   int n,i,p;
   int bintra1(float a[], int l, int d, float b);
    printf("\nBroj elemenata? "); scanf("%d",&n);
    printf("Elementi niza ");
    for(i=0; i<n; i++)scanf("%f", &a[i]);
    printf("Jedan realan broj? "); scanf("%f",&br);
    p=bintra1(a, 0, n-1, br);
    printf("Pozicija broja %f u nizu je %d\n",br,p);
 }

int bintra1(float a[], int l, int d, float b)
 { int s;
    if(l>d) return(-1);
    else
     { s=(l+d)/2;
       if(b==a[s]) return(s);
       else if(b<a[s])return bintra1(a,l,s-1,b);
       else return bintra1(a,s+1,d,b);
     }
 }
```

**1. (C)** Napisati funkciju koja učitava broj elemenata u nizu kao i elemente niza. Napisati funkciju koja od dva neopadajuća niza *a*[0..*m*-1] i *b*[0..*n*-1] formirati rastući niz *c*[0..*k*-1] u kome se elementi koji se ponavljaju u polaznim nizovima pojavljuju samo jednom. Napisati test program.

```
#include<stdio.h>
int a[50], b[50], c[50];

void citaj(int x[], int *n)
{ printf("Broj elemenata = ? "); scanf("%d",n);
  printf("Elementi?\n");  for(int i=0; i<*n; i++)scanf("%d",x+i);
}

void formiraj(int a[], int b[], int c[], int m, int n, int *k)
{ int i=0,j=0,last;
  last = ((a[0]<b[0])?a[0]:b[0])-1;
  *k=0;
  while(i<m || j<n)
        if(i==m){ for(int l=j; l<n; l++)c[(*k)++]=b[l];  j=n;}
        else if(j==n){ for(int l=i; l<m; l++)c[(*k)++]=a[l];  i=m; }
        else if(a[i]<b[j])
        { if(a[i]!=last) {c[(*k)++]=a[i]; last=a[i];}
          i++;
        }
      else if(b[j]<a[i])
        { if(b[j]!=last){ c[(*k)++]=b[j]; last=b[j]; }
          j++;
        }
        else
        { if(a[i]!=last) {c[(*k)++]=a[i]; last=a[i];}
          i++; j++;
        }
}

void main()
{ int i,m,n,k;
```

```
  citaj(a,&m); citaj(b,&n);
  formiraj(a,b,c,m,n,&k);
  for(i=0; i<k;i++)printf("c[%d] = %d\n",i,c[i]);
}
```

**Primer.** Postupkom Eratostenovog sita ispisati sve proste brojeve od 2 do *n*. Postupak se sastoji u sledećem:

- Formira se niz koji sadrži prirodne brojeve 2 do *n*.
- Iz niza se izbacuju svi brojevi koji su deljivi sa 2, 3. Brojevi deljivi sa 4 su izbačeni zbog deljivosti sa 2, i tako dalje.
- U nizu ostaju prosti brojevi.

```
#include <stdio.h>
void main() {
   int a[1001], n, i, j;
   scanf("%d",&n);
   for (i=2; i<=n; i++) a[i]=1;
   for (i=2; i<=n; i++)
      if (a[i]) {
         j=2*i;     while (j<=n) { a[j]=0;    j+=i;  }
      }
   for (i=2; i<=n; i++) if (a[i]) printf("%d\n",i);
}
```

**Primer.** Napisati funkciju za deljenje polja *P* na dva dela, pri čemu u polje *P*1 ulaze svi elementi polja *P* veći od zadatog broja *k*, a u polje *P*2, svi elementi polja *P* manji od *k*. Elementi polja *P*1 i *P*2 treba da budu sortirani.

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

void main(void){
   int p[20],p1[20],p2[20],n,i,j,zadati,br1,br2;

    cout<<"\n Unesite dimenziju niza:"; cin>>n;
    cout<<"\n Unesite elemente niza:\n";
    for(i=0; i<n; i++) cin>>p[i];
    cout<<"\n Unesite zadati element:"; cin>>zadati;
    br1=0;  br2=0;
    for(i=0; i<n; i++)
     { if(p[i]>zadati) { p1[br1]=p[i];     br1++; }
       else      { p2[br2]=p[i]; br2++; };
     };
    for(i=0; i<br1-1; i++)
     { for(j=i+1; j<=br1-1; j++)
         { if(p1[i]>p1[j]){ zadati=p1[i]; p1[i]=p1[j]; p1[j]=zadati; };
         };
     };

    for(i=0; i<br2-1; i++)
     { for(j=i+1; j<=br2-1; j++)
        { if(p2[i]>p2[j]){ zadati=p2[i]; p2[i]=p2[j]; p2[j]=zadati; };
        };
     };
    cout<<"\n Polje P1\n";
    for(i=0; i<br1; i++) cout<<"\n P1["<<i<<"]="<<p1[i];
    cout<<"\n\n\n Polje P2\n";
    for(i=0; i<br2; i++) cout<<"\n P2["<<i<<"]="<<p2[i];
    cout<<"\n\n";
 }
```

**Primer.** Definisati sve varijacije sa ponavljanjem i varijacije bez ponavljanja.

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

```
bool mark[33];

void ispisi_varijacije_sa_ponavljanjem(int x, int n, int k, int a[], int
v[]) {
    int i;
      if (x==k) {
            printf("  ");
        for(i=0; i<k; i++) printf("%d ", v[i]);
            printf("\n");
      } else {
            for(i=0; i<n; i++) {
                  v[x]=a[i];
                  ispisi_varijacije_sa_ponavljanjem(x+1,n,k,a,v);
            }
      }
}

void ispisi_varijacije_bez_ponavljanja(int x, int n, int k, int a[], int
v[]) {
    int i;
      if (x==k) {
            printf("  ");
        for(i=0; i<k; i++) printf("%d ", v[i]);
            printf("\n");
      } else {
            for(i=0; i<n; i++) if (!mark[i]) {
                  mark[i]=true;
                  v[x]=a[i];
                  ispisi_varijacije_bez_ponavljanja(x+1,n,k,a,v);
                  mark[i]=false;
            }
      }
}

int main() {
      int n,i,k,a[33],v[33];
      scanf("%d", &n);
      for(i=0; i<n; i++) scanf("%d", &a[i]);

      printf("unesite k: "); scanf("%d", &k);

      printf("Sve varijacije(n,k) sa ponavljanjem: \n");
      ispisi_varijacije_sa_ponavljanjem(0,n,k,a,v);

      printf("Sve varijacije(n,k) bez ponavljanja: \n");
      ispisi_varijacije_bez_ponavljanja(0,n,k,a,v);

      //printf("Sve kombinacije: \n");
      //ispisi_sve_kombinacije(n,a);

      return 0;
}
```

Problem ranca

```
#include <cstdio>

int main (){
      int C, m, d[1000], t[100], v[100], uzeti[1000];
      scanf("%d%d", &C, &m);
```

```
	for (int i=0; i<m; i++) scanf("%d%d", t+i, v+i);

	d[0]=0;
	uzeti[0]=0;

	for (i=1; i<=C; i++){
		d[i]=0;
		for (int j=0; j<m; j++){
			if (i-t[j]>=0) {
				int tmp=d[i-t[j]]+v[j];
				if (tmp>d[i]) {
					d[i]=tmp;
					uzeti[i]=j;
				}
			}
		}
	}

	printf("Optimalna vrednost ranca sa nosivoscu %d je: %d\n", C, d[C]);
	printf("Uzeti predmeti redom su: \n");
	i=C;
	int k=1;
	while (i>0){
		k++;
		printf("Tezina %d-tog predmeta je: %d i vrednost %d-tog
predmeta je: %d\n", k, t[uzeti[i]], k, v[uzeti[k]]);
		i-=t[uzeti[i]];
	}

	return 0;
}
```

**Primer (C)** Dat je skup S od $n$ ($n \leq 100$) različitih prirodnih brojeva od kojih ni jedan nije veći od 200. Napisati program koji pronalazi jedan podskup $A$ datog skupa $S$ koji zadovoljava sledeća dva uslova:

a) Suma elemenata skupa $A$ je deljiva sa tri;
b) Od svih podskupova skupa $S$ koji zadovoljavaju osobinu a), podskup $A$ ima najveću sumu svojih elemenata.

Ulazni podaci se učitavaju iz fajla **zad1.dat**. U prvom redu fajla se nalazi broj $n$, a u nastavku fajla se nalazi $n$ elemenata skupa $S$ (svaki element je naveden u posebnom redu fajla).

Program treba da napravi fajl **zad1.res** i u njega treba da upiše sumu elemenata podskupa $A$ (u prvom redu fajla) i sve elemente podskupa $A$ (u nastavku fajla, svaki element se upisuje u posebnom redu fajla).

**Primer:**

| **zad1.dat** | **zad1.res** |
|---|---|
| 4 | 147 |
| 29 | 29 |
| 15 | 15 |
| 11 | 103 |
| 103 | |

```
#include <stdio.h>
int main(){
	FILE *f,*g;
	int am1,am2,bm1,bm2,n,i,a[100],ost,i1,i2,i3,i4;
	long sumaniza;
	f=fopen("zad12.dat","r");
	g=fopen("zad12.res","w");
	fscanf(f,"%d",&n);
	am1=am2=bm1=bm2=ost=0;
```

```
	sumaniza=0;
	i1=i2=i3=i4=-1;
	for(i=0;i<n;i++)
	{
		fscanf(f,"%d",&a[i]);
		if(((a[i]%3)==1)&&((am1==0)||(am1>a[i])))
		{
			am2=am1;
			am1=a[i];
			i1=i;
		}
		else if(((a[i]%3)==1)&&((am2==0)||(am2>a[i])))
		{
			am2=a[i];
			i2=i;
		}
		if(((a[i]%3)==2)&&((bm1==0)||(bm1>a[i])))
		{
			bm2=bm1;
			bm1=a[i];
			i3=i;
		}
		else if(((a[i]%3)==1)&&((bm2==0)||(bm2>a[i])))
		{
			bm2=a[i];
			i4=i;
		}
		sumaniza+=a[i];
		ost=((ost+a[i])%3);
	}
	if(((ost==1)&&(i1==-1)&&(i4==-1))||
	    ((ost==2)&&(i2==-1)&&(i3==-1)))
	{
		fprintf(g,"0\n0\n");
		fclose(f);
		fclose(g);
		return 0;
	}
	if(ost==0)
	{
		fprintf(g,"%ld\n",sumaniza);
		for(i=0;i<n;i++)
			fprintf(g,"%d\n",a[i]);
	}
	else if(ost==1)
	{
		if(am1<bm1+bm2)
		{
			fprintf(g,"%ld\n",sumaniza-am1);
			for(i=0;i<n;i++)
				if(i!=i1)
					fprintf(g,"%d\n",a[i]);
		}
		else
		{
			fprintf(g,"%ld\n",sumaniza-bm1-bm2);
			for(i=0;i<n;i++)
				if((i!=i3)&&(i!=i4))
					fprintf(g,"%d\n",a[i]);
		}
	}
```

```
      else
      {
                  if(bm1<am1+am2)
            {
                  fprintf(g,"%ld\n",sumaniza-bm1);
                  for(i=0;i<n;i++)
                        if(i!=i3)
                              fprintf(g,"%d\n",a[i]);
            }
            else
            {
                  fprintf(g,"%ld\n",sumaniza-am1-am2);
                  for(i=0;i<n;i++)
                        if((i!=i1)&&(i!=i2))
                              fprintf(g,"%d\n",a[i]);
            }
      }
      fclose(f);
      fclose(g);
      return 0;
     }
```

**Kratko objašnjenje rešenja.**
Prvo postavimo sve promenljive na odgovarajuće početne položaje. Zatim iz datoteke najpre učitamo broj elemenata skupa *S* a zatim i sve elemente skupa *S*. Pri učitavanju izdvajamo po dva najmanja elementa skupa *S* koji pri deljenu sa tri daju ostatke jedan odnosno dva. Ova dva elementa su bitna jer će mo odgovarajući skup A dobiti zapravo izbacivanjem najviše dva elementa skupa *S* koji daju potrebne ostatke pri deljenju sa tri, a da bi suma elememenata novodobijenog skupa *A* bila maksimalna potrebno je da elementi koje izbacujemo budu najmanji mogući. Zatim ispitujemo ostatak zbira elemenata skupa S pri deljenju brojem tri i u zavisnosti od tog ostatka ispitujemo najmanje elemente koji daju odgovarajući ostatak i u izlaznu datoteku štampamo skup S bez elemenata koje smo izbacili.

**Zadatak 1.** Dato je $n$ tegova ($n \leq 20$), čije su mase realni brojevi. Podeliti tegove u dve grupe, tako da ukupne mase tegova u grupama budu što približnije. štampati minimalnu razliku, i za svaku od grupa mase tegova u toj grupi, kao i ukupnu masu tegova u grupi.

```
#include<stdio.h>
#include<math.h>
main()
   { float teg[20];
     int grupa[20], grupa1[20];
     int i,j,k,n;
     float razlika, minrazlika, suma,suma1, minsuma1;
      printf("\n Broj tegova? "); scanf("%d", &n);
      printf("\nMase tegova? \n");
      for(i=0; i<n; i++) scanf("%f", &teg[i]);
      suma=0;
      for(i=0; i<n; i++)
        { grupa[i]=0; suma += teg[i]; }
      minrazlika = suma;
      printf("suma = %f\n", suma);
      do { suma1=0;
           for(i=0; i<n; i++) if(grupa[i]) suma1 += teg[i];
           razlika=fabs(suma-2*suma1);
           if(razlika < minrazlika)
             { minrazlika=razlika;
               minsuma1=suma1;
               for(i=0; i<n; i++) grupa1[i]=grupa[i];
             }
           j=n;
           while(j>=0 && grupa[j]) { grupa[j]=0; j--; }
```

```
            if(j>=0) grupa[j]=1;
          }
        while(j>=0);
        printf("\nMinimalna razlika =
                    %f\n",minrazlika);
        printf("Prva grupa ima sledece tegove:\n");
        for(i=0; i<n; i++) if(grupa1[i])printf("%f ",teg[i]);
        printf("\nUkupna masa tegova prve grupa je %f",
                     minsuma1);
        printf("\nDruga grupa ima sledece tegove:\n");
        for(i=0; i<n; i++) if(!grupa1[i])printf("%f ",teg[i]);
        printf("\nUkupna masa tegova druge grupa je %f",
                     (suma-minsuma1));
    }
```

**Primer.** Pretpostavimo da želite da postavite broj sobe na vratima. U prodavnici je moguće kupiti skup plastičnih cifara. Svaki skup sadrži po jednu cifru od 0 do 9. Broj sobe je između 1 i 1,000,000. Odrediti koliko skupova cifara je potrebno kupiti u prodavnici da bi ste ispisali broj sobe, imajući u vidu da 6 možete iskoristiti i kao 9 i obrnuto. U prvom redu datoteke **zad1.in** se nalazi broj ***n***, a u ostalih ***n*** redova po jedan broj koji označava broj koji bi trebalo ispisati. U datoteci **zad1.out** upisati minimalni broj skupova cifara koji uz pomoć kojih možemo ispisati dati broj.

| zad1.in | zad1.out |
|---|---|
| 4<br>122<br>9999<br>12635 | 2<br>2<br>1<br>6 |

**Primer.** U prvom redu datoteke 'zadl.in' nalazi se ceo broj $n$ ($1 \le n \le 1000$). U svakom od sledećih $n$ redova nalaze se po dva cela broja $x$ i $y$ ($-10000 \le x,y \le 10000$) koji predstavljaju koordinate tačke u ravni. Napisati program koji u datoteku 'zadl.out' upisuje, na dve decimale, rastojanje od koordinatnog početka tačke koja je najbliža koordinatnom početku. Iza rastojanja, odvojeno prazninom, upisati redni broj te najbliže tačke u datoteci.

| zad1.in | zad1.out |
|---|---|
| 7<br>-9 12<br>-12 -16<br>5 12<br>15 8<br>3 4<br>-8 -6<br>20 -15 | 5.00 5 |

**Primer.** Meteorološka stanica meri temperaturu jednom u minutu. Stanica ima termometar koji ne daje uvek tačne rezultate pri merenju temperature. Zbog nedostatka sredstava, odlučeno je da se umesto novog termometra finansira izrada programa koji uklanja netačne rezultate merenja i kao rezultat daje prosečnu temperaturu. Merenje se računa u netačno u dva slučaja:

a. Izmerena temperatura je manja od-50 stepeni;

b. Izmerena temperatura se od svake od temperatura merenih u prethodna 2 minuta i sledeća dva minuta razlikuje za više od 2.

Napisati program koji za zadati niz izmerenih temperatura izračunava srednju vrednost ispravnih temperatura.

U ulaznoj datoteci 'zad2.in' dati su broj $n$ ($1 \le n \le 1000$) koji predstavlja broj merenja, kao i brojevi $t_1$, ..., $t_n$ koji predstavljaju izmerene vrednosti ($-60 \le t_i \le 50$, $i \in \{1,...,n\}$). U izlaznu datoteku 'zad2.out' upisati, na dve decimale, srednju vrednost ispravno izmerenih temperatura.

| zad2.in | zad2.out |
|---|---|
| 5<br>9 11 12 13 15 | 12.00 |
| 7<br>0 0 0 2 5 0 10 | 0.16 |

# 6.3. Veza između nizova i pointera u C

Ime niza je konstantni pokazivač na početak niza, tj. adresa njegovog nultog elementa. Ime niza je samo po sebi jedna adresa, ili vrednost pointera (pointer je promenljiva koja uzima adrese za svoje vrednosti). Kada je niz deklarisan, kompajler alocira baznu adresu i dovoljan memorijski prostor za smeštanje svih elemenata niza. Kompajler jezika C sam prevodi oznake niza u pokazivače, pa se korišćenjem pokazivača povećava efikasnost u radu sa nizovima.

**Primer.** Neka je data deklaracija

```
#define N 100
long a[N], *p;
```

Pretpostavimo da je prva adresa za smeštanje niza *a* jednaka 300 (ostale su 304,308,…,696). Naredbe

```
p = a;          p=&a[0];
```

su ekvivalentne. U oba slučaja promenljiva *p* uzima za svoju vrednost adresu nultog elementa niza *a*. Preciznije, promenljivoj *p* se dodeljuje vrednost 300.

Takođe, naredbe

```
p=a+1;          p=&a[1];
```

su ekvivalentne i dodeljuju vrednost 304 promenljivoj *p*. Analogno, naredbe $p = a + i$ i $p = \&a[i]$ su ekvivalentne, za svaki element $a[i]$ niza *a*. Ako su elementima niza *a* pridružene vrednosti, one se mogu sumirati koristeći pointer *p*, na sledeći način:

```
sum=0;
for(p=a; p<&a[N]; sum += *p, ++p);
```

Isti efekat se može postići sledećim postupkom:

```
sum=0;
for(i=0;i<N;++i) sum += *(a+i);
```

Napomenimo da je izraz *(*a*+i) ekvivalentan sa *a*[i]. Takođe, može se pisati $p = \&(*(a+i))$ umesto izraza $p = a + i$, odnosno $p = \&a[i]$.

**Primer.** Još jedan način sumiranja nizova.

```
p=a; sum=0;
for(i=0;i<N;++i) sum += p[i];
```

Međutim, postoji razlika između pointera i nizova. Kako je ime niza *a* konstantni pointer, a ne promenljiva, ne mogu se koristiti izrazi

$$a = p{+}{+}a \qquad a{+}{=}2.$$

To znači da se adresa niza ne može menjati.

**Primer.** Napisati program koji transformiše niz celih brojeva tako da na početku budu negativni a na kraju nenegativni elementi tog niza.

```
#include<stdio.h>
void main(void)
 { int niz[100], p,i,k,n;
    printf("\nBroj elemenata --> "); scanf("%d",&n);
    for(i=0; i<n; i++)
     { printf("niz[%d]=--> ",i+1); scanf("%d",niz+i); }
```

```
    p=0; k=n-1;
    while(p<k)
      { while(niz[p]<0 && p<k)p++;
        while(niz[k]>=0 && p<k)k--;
        if(p<k)
         { int pom=niz[p]; niz[p]=niz[k]; niz[k]=pom;  }
      }
    for(i=0; i<n; i++)printf("niz[%d] = %d ",i,niz[i]); printf("\n");
 }
```

Dato je jedno rešenje koje koristi ime niza kao pokazivač.

```
#include<stdio.h>
void main(void)
 { int niz[100], *p,*k,i,n;
    printf("\nBroj elemenata --> "); scanf("%d",&n);
    for(i=0; i<n; i++){printf("niz[%d]= --> ",i+1); scanf("%d",niz+i); }
    p=niz; k=niz+n-1;
    while(p<k)
      { while(*p<0 && p<k)p++;
        while(*k>=0 && p<k)k--;
        if(p<k) { int pom;    pom=*p; *p=*k; *k=pom;   }
      }
    for(i=0; i<n; i++)printf("niz[%d]= %d ",i,*(niz+i)); printf("\n");
 }
```

Neka je, za ovaj primer zadat broj elemenata $n = 4$, i neka su elementi -1,2,-3,4. Neka je vrednost pokazivača $p$=&$a$[0] jednaka heksadekadnom broju `0x0012fdf0`. Ako je program napisan u programskom jeziku C++, tada svaka vrednost tipa *int* zauzima po 4 bajta, pa je vrednost pokazivača $k$=&$a$[3] jednaka heksadekadnom broju `0x0012fdfc`. Adrese i sadržaj lokacija koje su zauzete zadatim elementima niza predstavljene su u sledećoj tabeli:

| Element niza | Adresa prvog bajta | Adresa drugog bajta | Adresa trećeg bajta | Adresa četvrtog bajta |
|---|---|---|---|---|
| $a$[0] | 0x0012fdf0= &$a$[0] | 0x0012fdf1 | 0x0012fdf2 | 0x0012fdf3 |
| $a$[1] | 0x0012fdf4= &$a$[1] | 0x0012fdf5 | 0x0012fdf6 | 0x0012fdf7 |
| $a$[2] | 0x0012fdf8= &$a$[2] | 0x0012fdf9 | 0x0012fdfa | 0x0012fdfb |
| $a$[3] | 0x0012fdfc= &$a$[3] | 0x0012fdfd | 0x0012fdfe | 0x0012fdff |

**Primer.** Sortiranje niza pointerskom tehnikom. Kao algoritam izabrano je sortirane izborom uzastopnih minimuma.

```
#include<stdio.h>
void upis(int *a, int *n)
 { int i;
   printf("Broj elemenata? "); scanf("%d",n);
   printf("Elementi? ");
   for(i=0; i<*n; i++) scanf("%d",a+i);
 }

void ispis(int *a, int n)
 { int i;
    for(i=0; i<n; i++)printf("%d ",*(a+i));
 }

void uređenje(int *a, int n)
 { int i,j,pom;
    for(i=0; i<n-1; i++)
      for(j=i+1; j<n; j++)
        if(*(a+i)>*(a+j))
          {pom=*(a+i); *(a+i)=*(a+j); *(a+j)=pom; }
 }

void main()
```

```
{ void upis(int *, int*); void ispis(int *, int);
  int x[100],k;
  upis(x, &k);    uređenje(x,k);   ispis(x,k);
}
```

## 6.3.1. Pointerska aritmetika

U jeziku C dozvoljene su aritmetičke operacije nad pokazivačima. Dozvoljeno je:
- dodeliti pokazivaču adresu promenljive ili nulu (vrednost NULL);
- uvećati ili umanjiti vrednost pokazivača;
- dodati vrednosti pokazivača neki ceo broj;
- oduzeti od vrednosti pokazivača neki ceo broj;
- porediti dva pokazivača pomoću operacija ==, !=, i tako dalje;
- oduzeti od jednog pokazivača drugi, ako ukazuju na objekte istog tipa.

Pointerska aritmetika predstavlja veliku prednost jezika C u odnosu na druge jezike visokog nivoa.

U sledećoj tabeli su uvedene pretpostavke:
*a*: vektor sa elementima tipa *T*,
*n*: celobrojni izraz,
*v*: zraz tipa *T*,
*pa*1, *pa*2: ukazatelj na tip *T*, kojima se ukazuje na elemente vektora *a*.

U njoj su prikazana neka pravila pointerske (adresne) aritmetike:

| **Oznaka** | **Značenje** | **Tip rezultata** |
|---|---|---|
| *a* | ukazatelj na prvi element vektora *a* | ukazatelj na *T* |
| *&a*[0] | ukazatelj na prvi element vektora *a* | ukazatelj na *T* |
| *&a*[*n*] | ukazatelj na *n*+1 element vektora *a* | ukazatelj na *T* |
| **pa*1 | element vektora na koji pa1 ukazuje | *T* |
| **pa*1 = *v* | dodeljuje vrednost *v* elementu na koji *pa*1 ukazuje | *T* |
| ++*pa*1 | postavlja *pa*1 na sledeći element vektora *a* | ukazatelj na *T* |
| --*pa*1 | postavlja *pa*1 na prethodni element vektora *a* | ukazatelj na *T* |
| *++*pa*1 | inkrementira *pa*1, a zatim pristupa elementu na koji *pa*1 ukazuje | *T* |
| **pa*1++ | pristupa elementu vektora *a* na koji *pa*1 ukazuje, zatim inkrementira *pa*1 | *T* |
| *pa*1±*n* | ukazuje na *n*-ti naredni (prethodni) element počev od elementa na koji *pa*1 ukazuje | ukazatelj na *T* |
| *(*pa*1+*n*) = *v* | vrednost *v* dodejluje *n*-tom narednom elementu u odnosu na element na koji *pa*1 ukazuje | *T* |
| *pa*1>*pa*2 | ispituje vrednosti adresa u *pa*1 i *pa*2 pomoću relacionih operatora | *int* |
| *(*a*+*n*) | *n*-ti element vektora *a* | *T* |

**Primer.**

```
void main()
 { double a[2], *p, *q;
    p=&a[0]; q=p+1;
    printf("%d\n",q-p);                    /* 1 */
    printf("%d\n",(int)q-(int)p);          /* 8 */
 }
```

**Primer.** Date su deklaracija

```
int x[4], *pti;
```

i dodela

```
pti=x;
```

Ako je niz *x* smešten počev od adrese 56006(=&*x*[0]), odrediti lokacije i elemente niza *x* na koji ukazuju *pti*, *pti*+1, *pti*+2 i *pti*+3.

**Rešenje.** Dodavanje jedinice pokazivaču *pti* automatski uvećava adresu na koju on ukazuje za 2, jer se radi o pokazivaču na tip *int*. Znači, *pti*, *pti*+1, *pti*+2 i *pti*+3 ukazuju redom na adrese 56006, 56008, 56010, 56012 i na elemente *x*[0], *x*[1], *x*[2], *x*[3] niza *x*.

Iz prethodnog primera zaključujemo

*(x+2)=x[2], x+2=&x[2].

Izraz **x*+2 jednak je izrazu (**x*)+2 i predstavlja uvećanje vrednosti nultog elementa niza *x* za 2.

U sledećoj tabeli je prikazan odnos između * i ++.

| | |
|---|---|
| ***p++ means:** | |
| ***p++** | **find the value at the end of the pointer** |
| ***p++** | **increment the POINTER to point to the next element** |
| **(*p)++ means:** | |
| **(*p)++** | **find the value at the end of the pointer** |
| **(*p)++** | **increment the VALUE AT THE END OF THE POINTER (the pointer never moves)** |
| ***++p means:** | |
| ***++p** | **increment the pointer** |
| ***++p** | **find the value at the end of the pointer** |

# 6.4. Nizovi i dinamička alokacija memorije u C

Za razliku od vektora i skalarnih promenljivih, kojima se implicitno dodeljuje memorija prilikom njihovog stvaranja i oslobađa prilikom brisanja, nezavisno od volje programera, dinamičke strukture zahtevaju eksplicitno dodeljivanje i oslobađanje memorije.

U biblioteci *stdlib.h* nalaze se definicije funkcija *malloc*, *calloc*, *free* i *realloc*, kojima se može eksplicitno upravljati memorijom. Definicija i semantika ovih funkcija je sledeća:

1. char *malloc(size)

rezerviše memorijski blok veličine *size* uzastopnih bajtova. Argument *size* je tipa *unsigned*. U slučaju da rezervacija memorije uspe, ova funkcija vraća pointer na tip *char*, koji ukazuje na rezervisani memorijski blok, a inače vraća 0. Ova funkcija ne menja sadržaj memorije koja je izdvojena.

2. char *calloc(n, size)

rezerviše memorijski blok dovoljan za smeštanje *n* elemenata veličine *size* bajtova, tj. rezerviše *n***size* uzastopnih bajtova. Rezervisani memorijski prostor se inicijalizuje na 0. Ako je rezervacija uspešna, rezultat je pointer na *char*, koji ukazuje na rezervisani memorijski blok, a inače 0.

3. void free(point)

oslobađa memorijski prostor koji je rezervisan funkcijama *calloc* ili *malloc*. Argument *point* je pointer na *char*, koji ukazuje na memorijski prostor koji se oslobađa. Ova funkcija ne vraća vrednost. Pokazivač *point* mora da ima vrednost koja je dobijena primenom funkcija *malloc*() i *calloc*().

4. char *realloc(point, size)

oslobađa rezervisani memorijski blok i rezerviše novi, veličine *size* bajtova. Argument *point* je pointer na *char* i ukazuje na memorijski blok koji se realocira. Argument *size* je tipa *unsigned* i određuje veličinu realociranog memorijskog prostora.

**Primer.** Naredbama

```
int *pint;
pint = (int *)malloc(400);
```

izdvaja se 400 uzastopnih bajtova. Ovaj memorijski prostor je dovoljan za smeštanje 200 celih brojeva u jeziku C, odnosno 100 celih brojeva u C++.

Takođe, vrednost izraza

```
pint = (int*)malloc(400);
```

je pokazivač na tip *int*, koji se dobija konverzijom pokazivača na *char* pomoću kast operatora. Ako tip *int* zauzima 4 bajta, tada *pint*+1 uvećava pokazivač za 4 bajta, i predstavlja adresu sledećeg celog broja. Na taj način, 400 bajtova se koristi za smeštanje 100 celih brojeva.

Ako želimo da izdvojimo onoliko memorije koliko je potrebno za za čuvanje vrednosti izraza ili podataka određenog tipa, koristi se operator sizeof na jedan od sledeća dva načina:

`sizeof(izraz);` - vraća memorijski prostor koji je dovoljan za smeštanje izraza *izraz*;

`sizeof(T);` - vraća memorijski prostor koji je dovoljan za čuvanje vrednosti tipa *T*.

**Primer.** Pomoću operatora

```
pint = (int *)malloc(sizeof(int));
```

izdvaja se memorijski prostor za celobrojnu vrednost. Adresa te oblasti se dodeljuje promenljivoj pint.

Noviji C prevodioci ne zahtevaju promenu tipa pokazivača koji vraća funkcija *malloc*(), jer je njen rezultat tzv. generički pokazivač oblika *void **, koji pokazuje na objekat bilo kog tipa.

**Primer.** Operatorima

```
long *plong;
plong = (long*)calloc(100, sizeof(long));
```

izdvaja se (i inicijalizuje na 0) memorijski prostor dovoljan za smeštanje 100 vrednosti tipa long (400 uzastopnih bajtova).

**Primer.** Izostavljanje uzastopno jednakih elemenata niza.

```
#include<stdio.h>

#include<stdlib.h>

void redukcija (float *a, int *n)
{ int i, j;
  for (i=1, j=0; i<*n; i++) if (a[j] != a[i]) a[++j] = a[i];
  *n = j + 1;
}

/* Ispitivanje potprograma "redukcija" */

void main ()
{ float *a;  int n, i;
  while (1) {
    printf ("n? "); scanf ("%d", &n);
    if (n < 0) break;
    a=(float *)malloc(n*sizeof(float));
    printf ("A? "); for (i=0; i<n; scanf ("%f", &a[i++]));
    redukcija (a, &n);
    a=(float *)realloc(a,n*sizeof(float));
    printf ("A= "); for (i=0; i<n; printf("%f ", a[i++]));
    printf ("\n\n");
  }
}
```

**Primer.** Iz zadatog niza izbaciti sva pojavljivanja njegovog maksimalnog elementa. Zadatak uraditi dinamičkom alokacijom memorije.

```
#include<stdio.h>
#include<stdlib.h>
void main()
{ int i,n,im,k=0;
  float max, *a;
  scanf("%d",&n);
```

```
  a=(float *)malloc(n*sizeof(float));
  for(i=0;i<n;scanf("%f",&a[i++]));
  im=0; max=a[0];
  for(i=1;i<n;i++) if(a[i]>max) { max=a[i];im=i; }
  k=im-1;
  for(i=im+1; i<n; i++)  if(a[i]!=max)a[++k]=a[i];
  a=(float *)realloc(a,(k+1)*sizeof(float));
  for(i=0;i<=k;printf("a[%d]=%f  ",i,a[i++]));
}
```

**Primer.** Učitavanje, sortiranje i ispis nizova pomoću potprograma.

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

void ucitaj(int *a, int &n) {
    scanf("%d",&n);
    for (int i=0; i<n; i++)scanf("%d",a+i);
}

void razmeni(int &a, int &b) {
    int p=a;  a=b;      b=p;
}

void SORT(int **a, int n) {
      int *A;
      A=*a;
    for (int i=0; i<n-1; i++)
            for (int j=i+1; j<n; j++)
                if (*(A+i)>*(A+j)) razmeni(*(A+i),*(A+j));

}

int ispisi(int *a, int n) {
    for (int i=0; i<n; i++)   printf("%d, ",a[i]);
    printf("\n");
    return 0;
}

int main() {
    int n,*a;
      a=(int *)malloc(2000*sizeof(int));
      ucitaj(a,n);
      SORT(&a,n);
      ispisi(a,n);
      return 0;
}

Bubble sort
#include <stdio.h>
#include <math.h>
void bubbleSort(float *a, int n)
{int i,j; float p;
 j=(n-1);
 while(j>=1)
 {for(i=0;i<j;i++)
   if(a[i]>a[i+1])
   { p=a[i];    a[i]=a[i+1];    a[i+1]=p;   }
   j=j-1;
 }
}
void main()
{int i,n;float a[10];
```

```
 scanf("%d",&n);
 for(i=0;i<n;i++)scanf("%f",&a[i]);
 bubbleSort(a,n);
 for(i=0;i<n;i++) printf("%f  ",a[i]);
}
```

Drugi oblici funkcije SORT:

```
void SORT(int **a, int n) {
        for (int i=0; i<n-1; i++)
            for (int j=i+1; j<n; j++)
                if (*(*a+i)>*(*a+j)) razmeni(*(*a+i),*(*a+j));

}

void SORT(int **a, int n) {
        for (int i=0; i<n-1; i++)
            for (int j=i+1; j<n; j++)
                if (*(&(*a)[i])>*(&(*a)[j]))
                        razmeni(*(&(*a)[i]),*(&(*a)[j]));

}
```

---

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

void ucitaj(int *a, int &n) {
    scanf("%d",&n);
    for (int i=0; i<n; i++)scanf("%d",a+i);
}

void razmeni(int &a, int &b) {
    int p=a;  a=b;      b=p;
}

void SORT(int ***a, int n) {
        for (int i=0; i<n-1; i++)
            for (int j=i+1; j<n; j++)
                if ((*(*a))[i]>(*(*a))[j])
                        razmeni((*(*a))[i],(*(*a))[j]);

}


int ispisi(int *a, int n) {
    for (int i=0; i<n; i++)   printf("%d, ",a[i]);
    printf("\n");
    return 0;
}

int main() {
    int n,*a,**b;
      a=(int *)malloc(2000*sizeof(int));
      ucitaj(a,n);
      b=&a;
      SORT(&b,n);
      ispisi(a,n);
      return 0;
}
```

**Primer.** Nizovi kao skupovi pomoću dinamičke alokacije memorije.

```
#include<stdio.h>
#include<stdlib.h>
int sadrzi(const float *s1, int n1, float b);
void dodaj(float **s1, int *n1, float b);
void presek(const float *s1, int n1, const float *s2, int n2,
                      float **s3, int *n3);
void unija(const float *s1, int n1, const float *s2, int n2,
                     float **s3, int *n3);
/* void razlika(const float *s1, int n1, const float *s2, int n2,
                     float **s3, int *n3); n*/
void citaj(float **s, int *n);
void pisi(const float *s, int n, const char *f);

int sadrzi(const float *s1, int n1, float b)
{ int d=0, g=n1-1,p;
  while(d<=g)
        if(s1[p=(d+g)/2]==b)return 1;
        else if(s1[p]<b)d=p+1;
        else g=p-1;
  return 0;
}

void dodaj(float **s1, int *n1, float b)
{ float *s=*s1; int n=*n1,i;
  if(!sadrzi(s,n,b))
  { *s1=s=(float *)realloc(s,(n+1)*sizeof(float));
    for(i=n-1; i>=0&&s[i]>b;i--)s[i+1]=s[i];
    s[i+1]=b; *n1=n+1;
  }
}

void presek(const float *s1, int n1, const float *s2, int n2,
                  float **s3, int *n3)
{ int n=0,i1=0,i2=0;
  float *s=(float *) malloc((n1<n2 ? n1 : n2)*sizeof(float));
  while(i1<n1 && i2<n2)
    if(s1[i1]<s2[i2])i1++;
    else if(s1[i1]>s2[i2])i2++;
    else s[n++]=s1[i1++],i2++;
  *s3=(float *)realloc(s,(*n3=n)*sizeof(float));
}

void unija(const float *s1, int n1, const float *s2, int n2,
               float **s3, int *n3)
{ int n=0, i1=0, i2=0;
  float *s=(float *)malloc((n1+n2)*sizeof(float));
  while(i1<n1 || i2<n2)
        if(i1==n1) s[n++]=s2[i2++];
        else if(i2==n2)s[n++]=s1[i1++];
        else if(s1[i1]<s2[i2])s[n++]=s1[i1++];
        else if(s2[i2]<s1[i1])s[n++]=s2[i2++];
        else s[n++]=s1[i1++],i2++;
  *s3=(float *)realloc(s,(*n3=n)*sizeof(float));
}

void citaj(float **s1, int *n1)
{ int n,i;
  float *s;
  scanf("%d",&n); *n1=n;
  if(n>=0)
  { *s1=s=(float *)malloc(n*sizeof(float));
    for(i=0; i<n; scanf("%f",&s[i++]));
```

```
  }
  else *s1=0;
}

void pisi(const float *s, int n, const char *f)
{ int i;
  putchar('{');
  for(i=0;i<n; i++)
  { if(i>0) putchar(',');  printf(f,s[i]); }
  putchar('}');
}

void main()
{ float *s1, *s2, *s3;
  int n1,n2,n3;
  while(1)
  { citaj(&s1,&n1);  if(n1<0)break;
    citaj(&s2,&n2);  if(n2<0)break;
    printf("s1   "); pisi(s1,n1,"%.2f"); putchar('\n');
    printf("s2   "); pisi(s2,n2,"%.2f"); putchar('\n');
    presek(s1,n1,s2,n2,&s3, &n3);
    printf("s1*s2   "); pisi(s3,n3,"%.2f"); putchar('\n');
    unija(s1,n1,s2,n2,&s3, &n3);
    printf("s1+s2   "); pisi(s3,n3,"%.2f"); putchar('\n');
    free(s1); free(s2); free(s3);
  }
 }
```

**Primer.** Za svaku vrstu robe u magacinu beleži se njena šifra i količina, kao dva cela broja. Za magacin *A* dati su nizovi *sa* i *ka* od *na* elemenata, sa značenjem da robe sa šifrom *sa*[*i*] ima u količini *ka*[*i*], i da ukupno ima *na* vrsta robe u magacinu. Za magacin *B* dati su analogni podaci u nizovima *sb* i *kb* od *nb* elemenata. Podaci o robi su uređeni po rastu šifara. Formirati nizove *sz* i *kz* koji daju podatke o zbirnom stanju u magacinima *A* i *B*, i koji su takođe uređeni po siframa artikala. Koristiti dinamičku alokaciju nizova.

```
#include<stdio.h>
#include<stdlib.h>
void main()
{ int *sa,*ka,*sb,*kb, *sz,*kz;
  int i,ia,ib,iz,na,nb,nz;
  printf("Broj vrsta robe u magacinu A? "); scanf("%d",&na);
  sa=(int *)malloc(na*sizeof(int)); ka=(int *)malloc(na*sizeof(int));
  for(i=0; i<na; i++)
  { printf("cifra i kolicina za %d. artikal? ",i);
    scanf("%d%d", &sa[i], &ka[i]);
  }
  printf("Broj vrsta robe u magacinu B? "); scanf("%d",&nb);
  sb=(int *)malloc(nb*sizeof(int));  kb=(int *)malloc(nb*sizeof(int));
  for(i=0; i<nb; i++)
  { printf("cifra i kolicina za %d. artikal? ",i);
    scanf("%d%d", &sb[i], &kb[i]);
  }
  sz=(int *)malloc((na+nb)*sizeof(int)); kz=(int *)malloc((na+nb)*sizeof(int));
  ia=ib=iz=0;
  while (ia<na && ib<nb)
    if(sa[ia]<sb[ib])
       { sz[iz]=sa[ia]; kz[iz++]=ka[ia++]; }
       else if(sa[ia]>sb[ib])
       { sz[iz]=sb[ib]; kz[iz++]=kb[ib++]; }
    else // sa[ia]=sb[ib]
       { sz[iz]=sa[ia]; kz[iz++]=ka[ia++]+kb[ib++]; }
  while(ia<na)
  { sz[iz]=sa[ia]; kz[iz++]=ka[ia++]; }
  while(ib<nb)
  { sz[iz]=sb[ib]; kz[iz++]=kb[ib++]; }
```

```
  nz=iz-1;
  sz=(int *)realloc(sz,nz*sizeof(int));
  kz=(int *)realloc(kz,nz*sizeof(int));
  for(i=0;i<nz; printf("sifra %d kolicina %d\n",sz[i],kz[i++]));
  free(sa); free(sb); free(ka); free(kb); free(sz); free(kz);
}
```

**Primer.** Napisati program kojim se zamenjuje prvo pojavljivanje stringa *s*1 u stringu *s* stringom *s*2. Stringovi *s*1 i *s*2 mogu biti različitih dužina. Sve stringove pamtiti u dinamičkoj zoni memorije i za svaki od njih rezervisati onoliko memorijskog prostora koliko je neophodno.

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

void main()
{ char *s,*s1,*s2,c;
  int n,n1,n2,i,j,k;
  printf("Duzine stringova = ? "); scanf("%d%d%d",&n,&n1,&n2);
scanf("%c",&c);
  s=(char *)malloc(n+1); s1=(char *)malloc(n1+1); s2=(char *)malloc(n2+1);
  printf("Unesite stringove\n");
  gets(s); gets(s1); gets(s2);

  //Pronalazi s1 u s
  //i je pocetna adresa od koje trazimo u s,
  // j je broj pronadjenih znakova iz s1 u s
  for(i=0,j=0;i<=n-n1 && j<n1; )
  { if(s[i+j]==s1[j])j++;
    else {j=0; i++; }
  }

  if(j<n1)printf("s1 se ne sadrzi u s\n");
  else
  { if(n2>n1)  //s se prosiruje
      { k=n2-n1;
      realloc(s,n+k+1);
        for(j=n; j>=i+n1; j--)s[j+k]=s[j];
      }
      else if(n1>n2)//  s se smanjuje
      { k=n1-n2;
        for(j=i+n1; j<=n; j++)s[j-k]=s[j];
        realloc(s,n-k+1);
      }
       printf("i = %d\n",i);
      for(j=0; j<n2; j++) s[i+j]=s2[j];
      printf("Transformisani string je %s\n",s);
  }
}
```

Test primer.

```
Duzine strigova = ?     9   3   2
Unesite stringove
rat i mir
mir
mi
Transformisani string je  rat i mi
```

# 6.5. Višedimenzionalna polja

Kod rešavanja mnogih problema postoji potreba za radom sa višedimenzionalnim strukturama

podataka. U programskim jezicima se za rad sa ovakvim strukturama podataka mogu definisati višedimenzionalna polja, kao jednorodne strukture podataka u okviru kojih se pojedinačnim elementima pristupa preko dva ili više indeksa. Posebno su značajne dvodimenzionalne strukture, koje odgovaraju pojmu matrica i omogućavaju rešavanje veoma široke klase problema u kojima se koriste matrične reprezentacije podataka.

## 6.5.1. Višedimenzionalni nizovi u C

Jednim parom zagrada definišu se jednodimenzionalni nizovi. Svaki par zagrada definiše jednu dimenziju niza. Na primer, sledeće deklaracije su korektne:

```
int a[100];  int b[3][5];  int c[7][9][2];
```

U jeziku C proizvoljni *k*-dimenzionalni niz poseduje veličinu svake od svojih *k* dimenzija.

**Primer.** Zadavanje elemenata matrice na mestu njene deklaracije.

```
#include <stdio.h>
main()
 { int i,j,a[2]={1,2},  b[2][3]={{45,67,88},{67,777,54}}, s=0;
    for(i=0;i<2;++i) s+=a[i];
    printf("\nSuma elemenata niza je:%d\n",s);
    for(i=0;i<2;i++)
        for(j=0;j<3;j++)  printf("b[%d%d]=%d\n",i,j,b[i][j]);
 }
```

**Primer.** Program za množenje matrica koji koristi korisnički tip *mat* za definiciju matrica reda 3×3.

```
void main()
 {int i, j, k;
   typedef double mat[3][3];
   static mat m1={{1.0,2.0,3.0},{4.0,5.0,6.0},{7.0,8.0,9.0}},
              m2={{9.0,8.0,7.0},{6.0,5.0,4.0},{3.0,2.0,1.0}};
   mat m3;
   for(i=0; i<3; ++i)
       for(j=0; j<3; ++j)
           for(m3[i][j]=0, k=0; k<3; ++k)
                 m3[i][j]+= m1[i][k]*m2[k][j];
    printf("\nProizvod matrica\n");
    for (i=0; i<3; ++i)
      for (j=0; j<3; ++j)
        printf("%15.2f%c", m3[i][j], (j==2) ? '\n' : ' ');
 }
```

## 6.5.2. Pokazivači i višedimenzionalni nizovi u C

Dvodimenzionalni nizovi (matrice) se u memoriji registruju po vrstama, koristeći uzastopne memorijske lokacije. Na primer, dvodimenzionalni niz int *a*[3][2] se u memoriji raspoređuje u sledećem poretku:

```
a[0][0], a[0][1], a[1][0], a[1][1],  a[2][0], a[2][1] .
```

Ako se deklariše pokazivač p pomoću

```
int *p
```

posle dodele *p* = *a* pointer *p* uzima za svoju vrednost adresu elementa koji leži u nultoj vrsti i nultoj koloni matrice *a* (tj. adresu prvog elementa matrice *a*). Takođe, važe sledeće pointerske jednakosti:

```
p=&a[0][0],  p+1=&a[0][1],  p+2=&a[1][0], p+3 =&a[1][1],  p+4=&a[2][0],  p+5=&a[2][1].
```

Dvodimenzionalni nizovi jesu jednodimenzionalni nizovi jednodimenzionalnih nizova. Na primer, ako je *r* ime dvodimenzionalnog niza, tada je *r*[*i*] jednodimenzionalni niz koji sadrži elemente *i*-te vrste matrice *r*.

U našem slučaju, vrste matrice *a* date su izrazima *a*[0], *a*[1] i *a*[2]. S obzirom na poznata svojstva jednodimenzionalnih nizova, važe sledeće jednakosti:

a[0]=&a[0][0], a[1]=&a[1][0], a[2]=&a[2][0] .

Osim toga, važe sledeće jednakosti:

a[i]=&a[i][0], a[i] + j = &a[i][j], a[i][j] = *( a[i] + j), &a[i][j] = a+i*2+j.

Takođe je

a[j][k]=*(a+j*duzina_vrste +k).

Neka je mat[][7] dvodimenzionalni niz. Tada je

mat[i][j]=*(mat[i]+j)=(*(mat+i))[j]=*(mat+i*7+j)= *(&mat[0][0]+i*7+j)= *((*(mat+i))+j).

U gornjim primerima zagrade su neophodne zbog toga što operator selekcije ima viši prioritet u odnosu na operator indirekcije.

**Primer.** Definisan je tip *Matrica* u jeziku C++. Napisane su funkcije za ispis i upis elemenata matrice, za transponovanje matrice, za sabiranje dve matrice, kao i funkcija za ciklično pomeranje vrsta u matrici.

```
#include<stdio.h>

typedef float matrica[10][20];

void main()
{ int m,n;
  matrica a,b,c;
  void pisi(int, int, matrica);
  void citaj(int &, int &, matrica);
  void saberi(int, int, matrica, matrica,matrica);
  void rotiraj(int, int, matrica);
  void transponuj(int, int, matrica);
  citaj(m,n,a);
  citaj(m,n,b);
  saberi(m,n,a,b,c);  pisi(m,n,c);
  rotiraj(m,n,c);  pisi(m,n,c);
  transponuj(m,n,c);  pisi(n,m,c);
}
void citaj(int &k, int &l, matrica x)
{ int i,j;
  scanf("%d%d",&k,&l);
  for(i=0;i<k; i++)
        for(j=0; j<l; j++)
              scanf("%f",x[i]+j);
}

void pisi(int k, int l, matrica x)
{ int i,j;
  for(i=0;i<k; i++)
  { for(j=0; j<l; j++)
         printf("%.2f  ",x[i][j]);
    printf("\n");
  }
  printf("\n");
}

void saberi(int k, int l, matrica x, matrica y, matrica z)
{ int i,j;
      for (i=0;i<k;i++)
            for (j=0;j<l;j++)
              z[i][j]=x[i][j]+y[i][j];
}
```

```
void uzmi(int n, float *x, float *y)
{ int j;
  for(j=0; j<n; j++)x[j]=y[j];
}

void rotiraj(int m, int n, matrica a)
{ int i;
  float p[20];
  uzmi(n,p,a[m-1]);
  for(i=m-2;i>=0;i--)
    uzmi(n,a[i+1],a[i]);
  uzmi(n,a[0],p);
}

void transponuj(int m, int n, matrica a)
{ int i,j;
  for(i=0; i<m; i++)
        for(j=i+1; j<n; j++)
        { float p=a[i][j]; a[i][j]=a[j][i]; a[j][i]=p; }
}
```

**Primer.** Napisati potprogram koji pronalazi minimalni element niza i njegov indeks. U glavnom programu učitati matricu $m \times n$ i uz pomoć formiranog potprograma naći minimalni element u matrici i njegove indekse.

```
#include <stdio.h>
void minunizu(int a[], int n, int *mn, int *k)
 { int i;
   *k=0; *mn=a[0];
    for(i=1; i<n; i++)if(a[i]<*mn)   { *k=i; *mn=a[i]; }
 }

void main()
 { int m,n,i,j, min, ind;    int a[10][10], b[10], p[10];
    printf("Dimenzije matrice?: "); scanf("%d%d", &m,&n);
    printf("\nElementi matrice?: ");
    for(i=0; i<m; i++)
      for(j=0; j<n; j++) scanf("%d", &a[i][j]);
    for(i=0; i<m; i++)
        minunizu(a[i], n, &b[i], &p[i]);
       // Moze i minunizu(a[i], n, b+i, p+i);
    minunizu(b,m, &min, &ind);
    i=ind; j=p[ind];
    printf("Minimal-ni je na poziciji [%d,%d] i jednak je %d\n",i,j,min);
 }
```

**Primer.** Konstruisati magični kvadrat dimenzija $n \times n$, za $n$ neparno.

```
#include<stdio.h>
void main()
 { int i,j,k,n, mag[29][29];
    printf("Dimenzije kvadrata? ");
    scanf("%d", &n);
    for(i=1; i<=n; i++)
      for(j=1; j<=n; j++)
        mag[i][j]=0;
    i=1; j=(n+1)/2; k=1;
    while(k<=n*n)
      { mag[i][j]=k++;
        if(i==1 && j==1) i++;
        else if(i==1)        { i=n; j--; }
```

```
        else if(j==1)        { j=n; i--; }
        else if(mag[i-1][j-1]!=0) i++;
        else { i--; j--; }
      }
    for(i=1; i<=n; i++)
      { for(j=1; j<=n; j++) printf("%4d",mag[i][j]); printf("\n"); }
 }
```

**Primer.** Data je celobrojna matrica $A$ reda $n \times n$, gde je $n$ neparan broj. Matrica $A$ predstavlja magični kvadrat sa jednim pogrešnim brojem. Napisati program koji pronalazi grešku, ispravlja je, štampa odgovarajuću poruku i ispravljeni magični kvadrat.

```
void main()
 { int i,j,n, ik,iv, s1,s2,s3,s,suma, mag[29][29];
   printf("Dimenzije kvadrata? ");   scanf("%d", &n);
   printf("Zadati magicni kvadrat sa jednom greskom\n");
   for(i=1; i<=n; i++)
     for(j=1; j<=n; j++)   scanf("%d", &mag[i][j]);
   s1=s2=s3=0;
   for(j=1; j<=n; j++)
     { s1 += mag[1][j];   s2 += mag[2][j];   s3 += mag[3][j]; }
   if(s1==s2 || s1==s3) suma=s1;
      else suma=s2;
   i=0;
   do
      { i++; s=0;    for(j=1; j<=n; j++)s+=mag[i][j];   }
   while(s==suma);
   iv=i;
   j=0;
   do
     {j++; s=0; for(i=1; i<=n; i++)s+=mag[i][j];}
   while(s==suma);
   ik=j;
   printf("Greska je u vrsti %d i koloni %d\n",iv,ik);
   printf("Pogresna vrednost je %d\n",mag[iv][ik]);
   printf("Pravilna vrednost je %d\n",mag[iv][ik]+suma-s);
   mag[iv][ik]=mag[iv][ik]+suma-s;
   printf("Pravilni magicni kvadrat je:\n");
   for(i=1; i<=n; i++)
     { for(j=1; j<=n; j++) printf("%4d",mag[i][j]);
       printf("\n");
     }
 }
```

**Primer.** Napisati program koji najefikasnijim postupkom izračunava $n$-ti stepen matrice $A$. Takav stepen se može izračunati rekurzivno sa najmanjim brojem matričnih množenja na sledeći način:

$$A^n = \begin{cases} A, & n = 1 \\ (A^k)^2, & n = 2k \\ A(A^k)^2, & n = 2k+1 \end{cases}$$

```
#include<stdio.h>
void mnozi(int n, int a[10][10], int b[10][10], int c[10][10])
  { int i,j,k;
     int s;
     for(i=0; i<n; i++)
      for(j=0; j<n; j++)
       { s=0;
         for(k=0; k<n; k++) s+=a[i][k]*b[k][j];
         c[i][j]=s;
       }
  }
```

```
void kopi(int n, int a[10][10], int b[10][10])
  { int i,j;
     for(i=0; i<n; i++)
     for(j=0; j<n; j++) b[i][j]=a[i][j];
  }

void citaj(int n, int a[10][10])
  { int i,j;
    printf("Unesi %d*%d elemenata matrice\n", n,n);
    for(i=0; i<n; i++)
     for(j=0; j<n; j++) scanf("%d",&a[i][j]);
  }

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

void stepen(int m, int n, int a[10][10], int b[10][10])
  { int c[10][10], p[10][10];
    if(m==1) kopi(n,a,b);
    else { stepen(m/2,n,a,c);
           mnozi(n,c,c,p);
           if(m%2==1)mnozi(n,p,a,b);
           else kopi(n,p,b);
         }
  }

void main()
  { int n,m;
    int a[10][10], b[10][10];
    printf("Red kvadratne matrice = "); scanf("%d", &n);
    citaj(n,a);
    printf("Stepen = "); scanf("%d", &m);
    stepen(m,n,a,b); pisi(n,b);
  }
```

**Primer.** Svaki element matrice koji predstavlja maksimum svoje vrste i minimum svoje kolone naziva se sedlasta tačka. Pronaći sve sedlatse tačke u zadatoj matrici.

```
#define MAXX 100
#define MAXY 100
#include<stdio.h>
#include<conio.h>
#include<limits.h>
#include<stdlib.h>
int a[MAXX][MAXY], mini[MAXX], maxi[MAXY],m,n;

void main()
 { int i,j;
   printf("Dimenzije matrice? "); scanf("%d%d",&m,&n);
   printf("Elementi?\n");
   for(i=0;i<m;i++)
     for(j=0;j<n;j++) scanf("%d", &a[i][j]);
   for(i=0;i<m;i++)maxi[i]=maximum(i);
   for(j=0;j<n;j++)mini[j]=minimum(j);
   for(i=0;i<m;i++)
     for(j=0;j<n;j++)
```

```
        if(maxi[i]==a[i][j] && mini[j]==a[i][j])
            printf("a[%d][%d]=%d ",i,j,a[i][j]);
  }

 int minimum(int kol)
  { int i,rez=INT_MAX;
    for(i=0;i<m;i++) rez=min(rez,a[i][kol]);
    return(rez);
  }

 int maximum(int vrs)
  { int j,rez=INT_MIN;
    for(j=0;j<n;j++) rez=max(rez,a[vrs][j]);
    return(rez);
  }
```

Još jedno rešenje:

```
void main()
  { int a[100][100],p[100], q[100];
    int m,n,i,j,k;
    scanf("%d%d", &m,&n);
    for(i=0;i<m;i++)
      for(j=0;j<n;j++)        scanf("%d", &a[i][j]);
    for(i=0; i<m; i++)
     { k=0;
       for(j=1;j<n;j++)  if(a[i][j]>a[i][k])k=j;
       p[i]=k;
     }
    for(j=0; j<n; j++)
     { k=0;
       for(i=1;i<m;i++)
         if(a[i][j]<a[i][k])k=i;
       q[j]=k;
     }
    for(i=0;i<n;i++)
      if(q[p[i]]==i)    printf("%d %d\n",i,p[i]);
  }
```

**Primer.** Data je matrica $A_{m\times n}$ celih brojeva. Element $a[i,j]$ je vrh ako je veći od svojih susednih elemenata koji su iznad, ispod, sa leve i sa desne strane. Visina vrha je razlika između elementa i njegovog najvišeg suseda. Napisati program koji će formirati niz vrhova sortiran u nerastući redosled po visini.

```
#include<stdio.h>
int vrh(int i, int j, int m, int n, int a[10][10])
  { if(i==0 && j==0)
       return(a[i][j]>a[i][j+1] && a[i][j]>a[i+1][j]);
    else if(i==m-1 && j==n-1)
       return(a[i][j]>a[i-1][j] && a[i][j]>a[i][j-1]);
    else if(i==0 && j==n-1)
       return(a[i][j]>a[i][j-1] && a[i][j]>a[i+1][j]);
    else if(i==m-1 && j==0)
       return(a[i][j]>a[i-1][j] && a[i][j]>a[i][j+1]);
    else if(i==0)
       return(a[i][j]>a[i][j-1] &&
            a[i][j]>a[i+1][j] && a[i][j]>a[i][j+1]);
    else if(j==0)
       return(a[i][j]>a[i-1][j] && a[i][j]>a[i][j+1]&&
                                    a[i][j]>a[i+1][j]);
    else if(i==m-1)
       return(a[i][j]>a[i-1][j] && a[i][j]>a[i][j-1] &&
```

```
                                        a[i][j]>a[i][j+1]);
     else if(j==n-1)
        return(a[i][j]>a[i-1][j] && a[i][j]>a[i][j-1] &&
                                        a[i][j]>a[i+1][j]);
     else return(a[i][j]>a[i-1][j] && a[i][j]>a[i][j-1] &&
              a[i][j]>a[i+1][j] && a[i][j]>a[i][j+1]);
  }

int min2(int x, int y)
  {if(x<y) return(x); else return(y); }

int min3(int x, int y, int z)
  { int min;
     min=x;   if(y<min) min=y;  if(z<min) min=z;
     return(min);
  }

int min4(int x, int y, int z, int w)
  { int min;
     min=x; if(y<min) min=y; if(z<min) min=z;   if(w<min)min=w;
     return(min);
  }


int visina(int i, int j, int m, int n, int a[10][10])
 {  int min2(int,int);
    int min3(int,int,int);
    int min4(int,int,int,int);
    if(i==0 && j==0)
      return(min2(a[i][j]-a[i][j+1],a[i][j]-a[i+1][j]));
    else if(i==m-1 && j==n-1)
      return(min2(a[i][j]-a[i-1][j],a[i][j]-a[i][j-1]));
    else if(i==0 && j==n-1)
      return(min2(a[i][j]-a[i][j-1],a[i][j]-a[i+1][j]));
    else if(i==m-1 && j==0)
      return(min2(a[i][j]-a[i-1][j],a[i][j]-a[i][j+1]));
    else if(i==0)
     return(min3(a[i][j]-a[i][j-1],a[i][j]-a[i+1][j],
                                   a[i][j]-a[i][j+1]));
    else if(j==0)
     return(min3(a[i][j]-a[i-1][j],a[i][j]-a[i][j+1],
                                   a[i][j]-a[i+1][j]));
    else if(i==m-1)
     return(min3(a[i][j]-a[i-1][j],a[i][j]-a[i][j-1],
                                   a[i][j]-a[i][j+1]));
    else if(j==n-1)
     return(min3(a[i][j]-a[i-1][j],a[i][j]-a[i][j-1],
                                   a[i][j]-a[i+1][j]));
    else return(min4(a[i][j]-a[i-1][j],a[i][j]-a[i][j-1],
                     a[i][j]-a[i+1][j],a[i][j]-a[i][j+1]));
 }

void main()
  { int vrh(int i, int j, int m, int n, int a[10][10]);
     int visina(int i, int j, int m, int n, int a[10][10]);
     int i,j,a[10][10], vrhovi[25],visine[25], pom,k=0,m,n;
     scanf("%d%d", &m,&n);
     for(i=0;i<m;i++)
        for(j=0;j<n;j++)  scanf("%d",&a[i][j]);
     for(i=0;i<m;i++)
      for(j=0;j<n;j++)
       if(vrh(i,j,m,n,a))
         { vrhovi[k]=a[i][j]; visine[k]=visina(i,j,m,n,a); k++;}
```

```
   for(i=0;i<k-1;i++)
     for(j=i+1;j<k;j++)
       if(visine[i]<visine[j])
          { pom=visine[i]; visine[i]=visine[j]; visine[j]=pom;
            pom=vrhovi[i]; vrhovi[i]=vrhovi[j]; vrhovi[j]=pom;
          }
   printf("\nNiz vrhova i visina:\n");
   for(i=0;i<k;i++)printf("Vrh: %d  visina: %d\n",vrhovi[i],visine[i]);
 }
```

**Primer.** Dat je niz $a_0,\ldots, a_{k-1}$ celih brojeva. Napisati program kojim se formira kvadratna matrica reda $n$ takva da je niz $a_i$, $i = 0,\ldots, k\text{-}1$ upisan spiralno u tu matricu u smeru kretanja kazaljke na satu. Ukoliko niz ima manje od $n^2$ elemenata, posle svakih $k$ upisanih elemenata početi upisivanje od prvog.

```
#include <stdio.h>
int dodi[4]={0,1,0,-1};
int dođ[4]={1,0,-1,0};  /* desno, dole, levo, gore */

void main()
 { int a[10], k, mat[20][20], n, n1, smer, i,j,br,pombr,broj;
     printf("Broj elemenata u nizu? "); scanf("%d", &k);
     printf("elementi niza?\n"); for(i=0;i<k;i++) scanf("%d",&a[i]);
     printf("Dimenzija matrice? "); scanf("%d", &n);
     n1=n;  i=j=br=smer=0;
     for(broj=1; broj<=n*n; broj++)
      {   /* ciklus za postavljanje elemenata u matricu */
         mat[i][j]=a[br];
         /* i, j su indeksi matrice, a br je indeks niza */
         br++;
         if(br==k)br=0;     /* kad br dođe do k, vraca se na 0 */

         i+=dodi[smer]; j+=dođ[smer];
         if(i==n1 || j==n1 || (i==n-n1&&smer>0) || j==n-n1-1)
            /* sada se menja smer dodavanja */
           { if(i==n1) i=n1-1;  if(j==n1) j=n1-1;
             if(i==n-n1 && smer>0) i=n-n1+1;
             if(j==n-n1-1)j=n-n1;
             smer++;     /* sledeci smer */
             if(smer==4) /* upotrebljeni su svi smerovi */
               { smer=0;  /* ponovo na desno */
                 n1--;
               }
             i+=dodi[smer]; j+=dođ[smer];
           }
      }
    for(i=0;i<n;i++)
     {for(j=0;j<n;j++) printf("%d  ",mat[i][j]); printf("\n"); }
 }
```

**Primer.** Neka je dato $n$ koncentričnih krugova takvih da svaki od njih ima $m$ otvorenih vrata. Prolaskom kroz bilo koja vrata dobija se izvestan broj nenegativnih poena. Ako je data matrica $A$ dimenzije $n\times m$ čiji element $a[i,j]$ označava broj poena koji se osvaja prolaskom kroz $j$-ta vrata $i$-tog kruga, napisati program koji trasira put kroz $n$ vrata tako da se sakupi dati broj poena $s$.

Kroz svaka vrata se prolazi tačno jedanput.

```
program krugovi;
  type opseg=0..10;
       niz=array[opseg] of integer;
       matrica=array[opseg,opseg] of integer;
  var a:matrica;
```

```
      x:niz;
      i,j,m,n,s:integer;
      q:boolean;
  function suma(x:niz; k:opseg):integer;
    var i,j,s1:integer;
    begin
      s1:=0;
      for i:=1 to k do s1:=s1+a[i,x[i]];
      suma:=s1;
    end;

  procedure pisi(x:niz; n:integer);
    var i:integer;
    begin
     for i:=1 to n do
      begin
       write('Na ',i,'. krugu vrata ',x[i],' poeni ',a[i,x[i]],' ');
       writeln;
      end;
    end;

  procedure trazi(k:opseg);
    var i:opseg;
    begin
      i:=1;
      while (i<=n) and not q do
        begin
          x[k]:=i;
          if suma(x,k)<=s then
             begin
               if (k=n) and (suma(x,k)=s) then
                 begin
                   pisi(x,n); q:=true;
                 end
               else trazi(k+1);
             end;
          i:=i+1;
        end;
    end;

  begin
    write('Uneti broj krugova i broj vrata na svakom krugu ');
    readln(n,m);    write('Unesi sumu '); readln(s);
    writeln('Unesi broj poena na vratima ');
    for i:=1 to n do
      begin
        for j:=1 to m do read(a[i,j]);      readln;
      end;
    q:=false;
    trazi(0);     if not q then writeln('Nema resenja ');
  end.
```

**Primer.** Problem *n* dama.

```
#include <stdio.h>

int a[50][50],n;
void cepaj(int);
int radi(int,int);
void ispis();

void main()
```

```
  { scanf("%d",&n);  radi(0); }

void ispis()
  {
    int i,j;
    for (i=0; i<n; i++)
      { for (j=0; j<n; j++) printf("%d ",a[i][j]); printf("\n");  }
    printf("\n\n\n");
  }

void radi(int j)
  { for (int i=0; i<n; i++)
      if (cepa(i,j))
        { a[i][j]=1;
          if (j<n-1) dalje(j+1);
            else ispis();
          a[i][j]=0;
        }
  }

int dalje(int i, int j)
  { for (int k=0; k<j; k++)
      { if ((a[i][k]==1) || (a[i-j+k][k]==1) || (a[i+j-k][k]==1))
        return(0);     }
    return(1);
  }
```

**Primer.** Data je sahovska tabla dimenzija $n \times m$. Na polju sa koordinatama $(i,j)$ je data figura. jedan potez se sastoji u pomeranju figure za jedno polje gore, dole, levo ili desno. Napisati program koji za date $n,m,i,j$ određuje najmanji broj poteza kojim se figura dovodi u ugao table.

```
#include<stdio.h>
#include<math.h>
void plusOne(int a[], int n, int k)
{     int i,j;
      if(a[k-1]<n-1) a[k-1]++;
      else
      {     i=k-2; j=1;
            while((a[i]>=n-1-j) && (i>0))
            { i--; j++;       }
            a[i]++;
            for(j=i+1;j<k;j++)a[j]=a[j-1]+1;
      }
}

int jedanRed(int x[], int n)
{     int i,j,min=2000000000,rast;
      for(i=0;i<n;i++)
      {     rast=0;
            for(j=0;j<n;j++)
                  if(i!=j) rast+=abs(x[i]-x[j]);
            if(rast<min) min=rast;
      }
      return min;
}

int main()
{     int n,potezi[50],x[50],y[50],pomx[50],pomy[50],nizIndexa[50],i,j;
      scanf("%d",&n);
      for(i=0;i<n;i++)scanf("%d%d",&x[i],&y[i]);
      potezi[0]=0; potezi[1]=0;
      int min,rez;
      for(i=2;i<=n;i++)
```

```
      {     min=2000000000;
            nizIndexa[0]=0;
            for(j=1;j<i;j++)
                  nizIndexa[j]=nizIndexa[j-1]+1;
            while(nizIndexa[0]<=(n-i))
            {     rez=0;
                  for(j=0;j<i;j++)
                  {pomx[j]=x[nizIndexa[j]]; pomy[j]=y[nizIndexa[j]]; }
                  rez+=jedanRed(pomx,i)+jedanRed(pomy,i);
                  if(rez<min) min=rez;
                  plusOne(nizIndexa,n,i);
            }
            potezi[i]=min;
      }
      for(i=0;i<=n;i++)printf("%4d", potezi[i]);
      printf("\n");
      return 0;
}
```

**Primer.** Na obali nekog ostrva nalazi se *n* gradova označenih brojevima od 0 do *n*-1. Oko celog ostrva je izgrađen autoput koji prolazi kroz svaki od gradova. U nizu *d* su data rastojanja između gradova, tako da *d*[*i*] predstavlja dužinu autoputa između gradova *i* i *i*+1 (*d*[*n*-1] predstavlja dužinu puta između grada *n*-1 i 0). Napisati program koji za sve gradove određuje niz najkraćih rastojanja do ostalih gradova.

```
#include<stdio.h>
float rastojanje(int i, int j, int n, float d[10])
  { float rl,rd;
    int k;
     rl=rd=0;
     for(k=i; k<=j-1;k++)rl=rl+d[k];
     for(k=j; k<=n-1; k++)rd=rd+d[k];
     for(k=0;k<=i-1;k++)rd=rd+d[k];
     if(rl<rd) return(rl);
           else return(rd);
  }

void main()
  { float rastojanje(int i, int j, int n, float d[10]);
     int k,n,i,j;
     float d[10], rastojanja[10][10];
     printf("Broj gradova? "); scanf("%d",&n);
     printf("Rastojanja?\n");     for(k=0;k<n;k++) scanf("%f",&d[k]);
     for(i=0; i<n-1;i++)
       for(j=i+1;j<n;j++)
         { rastojanja[i][j]=rastojanje(i,j,n,d);
           rastojanja[j][i]=rastojanja[i][j];
         }
     for(j=0;j<n;j++)rastojanja[j][j]=0;
     for(i=0;i<=n-1;i++)
      {for(j=0;j<=n-1; j++)
         printf("Rastojanje između %d i %d = %f\n",i,j,rastojanja[i][j]);
       printf("\n");
      }
  }
```

**51. (PASCAL)** Napisati funkciju kojom se utvrdjuje da li je zadata matrica *A* dimenzija *m* x *n* ortonormirana. Matrica je ortonormirana ako je skalarni proizvod svakog para različitih vrsta jednak 0, a skalarni proizvod vrste sa samom sobom 1.

**Ulaz:** Dva pozitivna cela broja *m* i *n* koji predstavljaju dimenzije matrice.
**Izlaz:** Tekst *Matrica je ortonormirana*, ako je matrica ortonormirana, a inače tekst *Matrica nije ortonormirana.*

```
program ortomormirana;
type opseg=1..10;
     niz=array[opseg]of real;
     matrica=array[opseg]of niz;
var a:matrica; m,n:opseg;ort:boolean;

function skalpr(n:opseg; v1,v2:niz):real;
  var i:opseg;
      s:real;
  begin
     s:=0;
     for i:=1 to n do s:=s+v1[i]*v2[i];
     skalpr:=s
  end;

function isort(m,n:opseg; a:matrica):boolean;
 var lg:boolean;
     i,j:opseg;
     sp:real;
 begin
   lg:=true;
   for i:=1 to m do
      for j:=i to m do
        begin
           sp:=skalpr(n,a[i],a[j]);
           if ((i=j)and(sp<>1))or((i<>j)and(sp<>0)) then
             lg:=false;
        end;
   isort:=lg;
 end;

procedure ucitaj(m,n:opseg; var x:matrica);
  var i,j:opseg;
  begin
    for i:=1 to m do
      begin
        for j:=1 to n do read(x[i,j]);
        readln
      end;
  end;

begin
  readln(m,n);
  ucitaj(m,n,a);
  ort:=isort(m,n,a);
  if ort then writeln('Matrica je ortonormirana ')
         else writeln('Matrica nije ortonormirana ');
end.
```

**Test primeri:**

| | |
|---|---|
| 3 3<br>1 0 0<br>0 1 0<br>0 0 1<br>Matrica je ortonormirana | 3 3<br>1 1 0<br>0 1 0<br>0 0 1<br>Matrica nije ortonormirana |

**54. (PASCAL)** Data je matrica $A(n \times n)$ čiji su elementi:

$$a[i,j] = \begin{cases} \texttt{true}, & \text{ako postoji direktan jednosmeran put između gradova } i \text{ i } j, \\ \texttt{false}, & \text{u suprotnom.} \end{cases}$$

Napisati program kojim se formira i ispisuje matrica iz koje se vidi koji gradovi imaju vezu sa najmanje jednim presedanjem.

**Ulaz**: Matrica $A(n \times n)$.

**Izlaz**: Matrica veze $C(n \times n)$.

```
program gradovi;
  const g=10;
  type matrica=array[1..g,1..g] of boolean;
  var n:integer;veza,jednopres:matrica;
  procedure citajveze(n:integer; var veza:matrica);
    var i,j,v:integer;
    begin
      writeln('Unesi veze (1-da,0-ne) izmedju gradova');
      for i:=1 to n do
        begin
          for j:=1 to n do
            begin
                read(v);
                if v=1 then veza[i,j]:=true
                else veza[i,j]:=false;
            end;
          readln;
        end;
    end;

procedure pisiveze(brgradova:integer;a:matrica);
    var i,j:integer;
    begin
      writeln('Matrica veze (1-da, 0-ne)');
      for i:=1 to brgradova do
        begin
          for j:=1 to brgradova do
            if a[i,j] then write('1':2)
            else write('0':2);
          writeln;
        end;
    end;

procedure pro(brgradova:integer;a,b:matrica;var c:matrica);
   var log:boolean;
       i,j,k:integer;
   begin
     for i:=1 to brgradova do
       for j:=1 to brgradova do
         begin
             log:=false;
             for k:=1 to brgradova do
               log:=log or (a[i,k] and b[k,j]);
             c[i,j]:=log;
         end;
   end;
```

```
begin
     write('Broj gradova? '); readln(n);
     writeln('Veze između gradova?');
     citajveze(n,veza);
     pro(n,veza,veza,jednopres);
```

```
     pisiveze(n,jednopres);
   end.
```

**Test primer:**

```
Broj gradova? 3
Unesi veze (1-da,0-ne) izmedju gradova
1 0 1
0 1 0
0 1 1
Matrica veze (1-da,0-ne)
1 1 1
0 1 0
0 1 1
```

**116. (PASCAL)** Raspoređeno je 15 kuglica na sledeći način: u prvoj vrsti se nalazi 5 kuglica, u drugoj 4, u trećoj 3, u četvrtoj 2 i u petoj 1. Obeležiti svaku kuglicu jednim od brojeva 1,...,15, tako da svaka kuglica ima jedinstven broj i da važi jednakost

$$a_{ij}=|a_{i-1,j}-a_{i-1,j+1}|,\ i=2,3,4,5,\ \ j=1,5-i+1.$$

```
program kuglice;
 type  niz=array[1..15] of integer;
       matrica =array[1..5,1..5] of integer;
  var i,j,k,l,m,br,indu :1..15;
      upotrebljen:niz;
      a:matrica;
  procedure ispis(a:matrica);
    var i,j:1..5;
    begin
      for i:=1 to 5 do
        begin
          for j:=1 to 5-i+1 do write(a[i,j]:3);writeln
        end;
    end;

  function razliciti(k:integer; a:niz):boolean;
    var i,j:1..15;
        bol:boolean;
    begin
      bol := true;
      for i:=1 to k do
        for j:=i+1 to k do
           if a[i]=a[j] then bol:=false;
      razliciti:=bol;
    end;

  begin
    for i:=1 to 15 do
      for j:=1 to 15 do
        for k:=1 to 15 do
          for l:=1 to 15 do
            for m:=1 to 15 do
              begin
                 a[1,1]:=i;a[1,2]:=j;
                 a[1,3]:=k;a[1,4]:=l;a[1,5]:=m;

                 for br:=1 to 5 do
                   upotrebljen[br]:=a[1,br];
                 indu:=5;
                 for br:=1 to 4 do
                   begin
                     a[2,br]:=abs(a[1,br]-a[1,br+1]);
                     indu:=indu+1;
```

```
                    upotrebljen[indu]:=a[2,br];
                  end;
                for br:=1 to 3 do
                  begin
                    a[3,br]:=abs(a[2,br]-a[2,br+1]);
                    indu:=indu+1;
                    upotrebljen[indu]:=a[3,br];
                  end;
                for br:=1 to 2 do
                  begin
                    a[4,br]:=abs(a[3,br]-a[3,br+1]);
                    indu:=indu+1;
                    upotrebljen[indu]:=a[4,br];
                   end;
                a[5,1]:=abs(a[4,1]-a[4,2]);
                indu:=indu+1;
                upotrebljen[indu]:=a[5,1];
                if razliciti(indu,upotrebljen) then
                   begin
                     writeln;writeln;
                     ispis(a);readln;
                   end;
            end;
  end.
```

**Test primer:**

| Resenje 1 | Resenje 2 |
|---|---|
| 6 14 15 3 13<br>8 1 12 10<br>7 11 2<br>4 9<br>5 | 13 3 15 14 6<br>10 12 1 8<br>2 11 7<br>9 4<br>5 |

**117. (PASCAL)** Zadata je crno-bela matrica (0 definiše belu boju, a 1 crnu). Napisati program kojim se bele oblasti te matrice boje različitim bojama (boje se definišu sa 2,3,...). Ispisati obojenu matricu.

```
program bojenje;
   const k=20; l=20;
   type matrica = array[1..k,1..l] of integer;
   var m,n,i,j,boja:integer;
       tabla:matrica;
       prx,pry:array[1..4] of integer;
   procedure ispis(m,n:integer; a:matrica);
     var i,j:integer;
     begin
       for i:=1 to m do
         begin
           for j:=1 to n do write(a[i,j]:3);
           writeln;
         end;
     end;
   procedure upis(m,n:integer; var a:matrica);
     var i,j:integer;
     begin
       for i:=1 to m do
         begin
           for j:=1 to n do read(a[i,j]);readln;
         end;
     end;
   function moze(x,y:integer; a:matrica):boolean;
     begin
```

```
      if(x>=1)and(x<=m)and(y>=1)and(y<=n) then
             moze:=a[x,y]=0
      else   moze:=false;
    end;
  procedure oboji(x,y,boja:integer; var a:matrica);
    var i:integer;
    begin
      a[x,y]:=boja;
      for i:=1 to 4 do
        if moze(x+prx[i],y+pry[i],a) then
           oboji(x+prx[i],y+pry[i],boja,a)
    end;

  begin
    write('Dimenzije? '); readln(m,n);
    upis(m,n,tabla);
    prx[1]:=-1; pry[1]:=0;   { gore }
    prx[2]:=0; pry[2]:=1;    { desno }
    prx[3]:=1; pry[3]:=0;    { dole }
    prx[4]:=0; pry[4]:=-1;   { levo }
    boja:=1;
    for i:=1 to m do
      for j:=1 to n do
        if tabla[i,j]=0 then
          begin
            boja:=boja+1; oboji(i,j,boja,tabla);
          end;
    ispis(m,n,tabla);
  end.
```

**Test primer:**

```
Dimenzije? 3 4
0 0 1 0
0 1 1 0
0 0 0 1
 2 2 1 3
 2 1 1 3
 2 2 2 1
```

```
Dimenzije? 4 5
0 0 0 0 1
0 0 1 1 0
0 0 0 0 0
1 0 0 0 1
 2 2 2 2 1
 2 2 1 1 2
 2 2 2 2 2
 1 2 2 2 1
```

**118. (PASCAL)** Napisati program kojim se određuje minimalan broj (*P*,*Q*) konja kojima se mogu kontrolisati sva polja šahovske table dimenzije m×n. (*P*,*Q*) konj je figura koja se u jednom skoku premešta za *P* polja po horizontali i *Q* polja po vertikali. Običan šahovski konj je (1,2) konj.

```
program konj;
   const k=30; l=30;
 type matrica =array[1..k,1..l] of boolean;
 var m,n,p,q,x,y,mink:integer;
     tabla:matrica;
     skokpox,skokpoy:array[1..8] of integer;
 function moze(x,y:integer):boolean;
   begin
     if (x>=1)and(x<=m)and(y>=1)and(y<=n) then
           moze := not tabla[x,y]
      else moze:=false
    end;

 procedure konjana(x,y:integer);
     var i:integer;
     begin
       tabla[x,y]:=true;
```

```
       for i:=1 to 8 do
            if moze(x+skokpox[i],y+skokpoy[i]) then
               konjana(x+skokpox[i],y+skokpoy[i]);
     end;

 begin
   write('p,q= ? ');readln(p,q); write('m,n= ? ');
   readln(m,n);
   for x:=1 to m do
   for y:=1 to n do tabla[x,y]:=false;
   skokpox[1]:=-q;skokpoy[1]:=p;skokpox[2]:=-p;
   skokpoy[2]:=q;skokpox[3]:=p; skokpoy[3]:=q;
   skokpox[4]:=q; skokpoy[4]:=p;
   skokpox[5]:=q; skokpoy[5]:=-p;
   skokpox[6]:=p; skokpoy[6]:=-q;
   skokpox[7]:=-p; skokpoy[7]:=-q;
   skokpox[8]:=-q; skokpoy[8]:=-p;
     mink:=0;
     for x:=1 to m do
       for y:=1 to n do
         if not tabla[x,y] then
           begin
             konjana(x,y); mink:=mink+1;
           end;
     writeln('Minimalni broj konja je ',mink);
end.
```

**Test primer:**

| | |
|---|---|
| p,q= ? 1 1<br>m,n= ? 8 8<br>minimalan broj konja je 2<br>p,q= ? 1 3<br>m,n= ? 8 8<br>minimalan broj konja je 2 | p,q= ? 3 3<br>m,n= ? 8 8<br>minimalan broj konja je 18<br>p,q= ? 4 4<br>m,n= ? 8 8<br>minimalan broj konja je 32 |

**119. (PASCAL)** Data je realna matrica $X$ dimenzije $m \times n$ i celi brojevi $I_p$ i $J_p$, $(1 <= I_p <= m, 1 <= J_p <= n)$. Svakom elementu matrice pridružuje se jedno polje na tačno određen način. Polje je određeno sa tri elementa: koordinatama $(i,j)$ i realnim brojem koji predstavlja njegovu visinu. Broj $X[i,j]$ određuje visinu polja sa koordinatama $(i,j)$. U polju sa koordinatama $(I_p,J_p)$ nalazi se loptica. Loptica prelazi sa polja $A$ na polje $B$ ako važe sledeći uslovi:
- polja $A$ i $B$ su susedna, što znači da imaju zajedničku stranicu;
- visina polja $B$ je strogo manja od visine polja $A$, i
- polje $B$ ima najmanju visinu u odnosu na sva polja susedna u odnosu na $A$.

Loptica se zaustavlja u polju u odnosu na koje ne postoji polje na koje može da pređe. Takvo polje se naziva završno. Odrediti polje na kome će se loptica zaustaviti kao i put od početnog polja $(I_p, J_p)$ do završnog.

```
program lopta1;
 type matrica=array[1..30,1..30] of integer;
 var x:matrica;
     m,n,i,j,ip,jp:integer;
 function moze(a,b:integer):boolean;
  begin
     moze:= (a>=1)and(a<=m)and(b>=1)and(b<=n)
  end;
 procedure uradi(ip,jp:integer; x:matrica; m,n:integer);
  var q:boolean;
      poz,poz1,a,b,min,min1:integer;
  begin
    q:=true; poz:=ip; poz1:=jp;
```

```
   writeln('Put: ');
   while q do
    begin
      min:=x[poz,poz1]; a:=poz; b:=poz1; min1:=min;
      if moze(a-1,b)and(x[a-1,b]<x[poz,poz1])
                    and(x[a-1,b]<min) then
         begin
           if poz1<>b then
               poz1:=b; poz:=a-1;
               min:=x[a-1,b];
         end;
      if moze(a+1,b)and(x[a+1,b]<x[poz,poz1])
                    and(x[a+1,b]<min)then
         begin
           if poz1<>b then
               poz1:=b; poz:=a+1;
               min:=x[a+1,b];
         end;
      if moze(a,b-1)and(x[a,b-1]<x[poz,poz1])
                    and(x[a,b-1]<min) then
         begin
           if poz<>a then
               poz:=a; poz1:=b-1;
               min:=x[a,b-1];
         end;
      if moze(a,b+1)and(x[a,b+1]<x[poz,poz1])
                    and(x[a,b+1]<min) then
         begin
           if poz<>a then
               poz:=a; poz1:=b+1;
               min:=x[a,b+1];
         end;
      q:=min<>min1;min:=min1;
      if q then
        writeln(poz:6,poz1:6,'   ',x[poz,poz1]);
    end;
    writeln('Zavrsno polje: ');
    writeln(poz:6,poz1:6,'   ',x[poz,poz1]);
 end;

 begin
   writeln('Unesite dimenzije matrice');readln(m,n);
   writeln('Unesite elemente');
   for i:=1 to m do
     begin
       for j:=1 to n do   read(x[i,j]);
       readln;
     end;
   writeln('Unesite ip i jp:');
   readln(ip,jp);    uradi(ip,jp,x,m,n);
end.
```

**Test primer:**

| | |
|---|---|
| Unesite dimenzije matrice<br>3 3<br>Unesite elemente<br>1 2 3<br>6 5 4<br>7 8 9 | Unesite ip i jp:<br>1 3<br>Put:<br>1 2 2<br>1 1 1<br>Zavrsno polje:<br>1 1 1 |

**120. (PASCAL)** Napisati funkcije (rekurzivnu i iterativnu verziju) koje za datu relaciju nad

maksimalno devetočlanim skupom ispituju refleksivnost, simetričnost i tranzivnost. Relacija je opisana matricom istinitosnih vrednosti: ako je element $a_{ij}$ matrice *A* jednak `true`, to znači da je *i*-ti element skupa u relaciji sa j-tim elementom. Data je tekstualna datoteka. Svaki red datoteke ima oblik

$$n \quad i_1\ j_1\ i_2\ j_2\ \ldots\ i_k\ j_k,$$

gde je *n* dimenzija matrice, a $i_1$ i $j_1$ $(l=1,\ldots,k)$ indeksi elemenata koji su u relaciji. Dakle, jedan red datoteke opisuje jednu relaciju.
Napisati program koji prebrojava nekorektne redove u datoteci, redove koji opisuju relaciju ekvivalencije, kao i redove koji opisuju relaciju poretka (uređenja).

```
program pmfokt22;
  const maxn=9;
  type relacija = array[1..maxn,1..maxn] of boolean;

  function ucitaj(var dat:text;var n:integer;
                  var r:relacija):boolean;
    var i,j:integer;
        ind:boolean;
    begin
      ind:=true;
      read(dat,n);
      for i:=1 to n do
        for j:= 1 to n do r[i,j]:=false;
      while not eoln(dat) do
        begin
          read(dat,i);
          if(i<1) or (i>n) or eoln(dat) then ind:=false;
          if ind then
            begin
              read(dat,j);
              if (j<1) or (j>n) then ind:=false;
            end;
          if ind then r[i,j]:=true;
        end;
      ucitaj:=ind;
    end;

  (* Iterativne funkcije *)
  function refleks(n:integer; r:relacija):boolean;
    var i:integer;
        ind:boolean;
    begin
      ind:=true;
      for i:=1 to n do ind:=ind and r[i,i];
      refleks:=ind;
    end;

  function simet(n:integer; r:relacija):boolean;
    var i,j:integer;
        ind:boolean;
    begin
      ind:=true;
      for i:=1 to n-1 do
        for j:=i+1 to n do
            ind:=ind and r[i,j]=r[j,i];
        (* if r[i,j]<>r[j,i] then ind:=false *)
      simet:=ind;
    end;

  function antisimet(n:integer; r:relacija):boolean;
    var i,j:integer;
```

```
      ind:boolean;
  begin
    ind:=true;
    for i:=1 to n-1 do
      for j:=i+1 to n do
        if (i<>j) and r[i,j] and r[j,i] then ind:=false;
    antisimet:=ind;
  end;

function tranzit(n:integer; r:relacija):boolean;
  var i,j,k:integer;
      ind:boolean;
  begin
    ind:=true;
    for i:=1 to n do
      for j:=1 to n do
        for k:=1 to n do
            if r[i,j] andr [j,k] and (not r[i,k]) then
               ind:=false;
    tranzit:=ind;
  end;

(* Rekurzivne verzije funkcija *)
function refleksr(n:integer; r:relacija):boolean;
  begin
    if n=1 then
      refleksr:= r[1,1]
    else refleksr:=refleksr(n-1,r)and r[n,n];
  end;

function simetr(n:integer; r:relacija):boolean;
  var i:integer;
      ind:boolean;
  begin
    if n<=1 then simetr:=true
    else begin
           ind:=true;
           for i:=1 to n do ind:=ind and (r[i,n]=r[n,i]);
           simetr:=simetr(n-1,r) and ind;
         end;
    simetr:=ind;
  end;

function antisimetr(n:integer; r:relacija):boolean;
  var i,j:integer;
      ind:boolean;
  begin
    if n<=1 then antisimetr:=true
    else begin
          ind:=true;
          for i:=1 to n-1 do
             if r[i,n] and r[n,i] and (i<>n) then
              ind:=false;
          antisimetr:=ind and antisimetr(n-1,r);
         end;
  end;

function tranzitr(n:integer; r:relacija):boolean;
  var i,j:integer;
      ind:boolean;
  begin
```

```
    if n<=2 then tranzitr:=true
    else
      begin
        ind:=true;
        for i:=1 to n do
         for j:=1 to n do
                if r[i,j] and r[j,n] and(not r[i,n]) then
                  ind:=false;
         tranzitr:=ind and tranzitr(n-1,r);
       end;
  end;

procedure uradi;
  var brekv, brpor, brnekor:integer;
      f:text;
      r:relacija;
      n:integer;
      ref,sim,asim,tr,refr,simr,asimr,trr:boolean;
  begin
    brekv:=0; brpor:=0; brnekor:=0;
    assign(f,'c:\relacije');  reset(f);
    while not eof(f) do
      begin
        if not ucitaj(f,n,r) then
           brnekor:=brnekor+1
        else
         begin
           ref:=refleks(n,r); (* ref:=refleksr(n,r *)
           sim:=simet(n,r); (* sim:=simetr(n,r); *)
           asim:=antisimet(n,r);
           (* asim:=antisimetr(n,r);  *)
            tr:=tranzit(n,r); (* tr:=tranzitr(n,r); *)
            if ref and sim and tr then brekv:=brekv+1;
            if ref and asim and tr then brpor:=brpor+1;
          end;
      end;
    writeln('Iterativna verzija: ');
    writeln('Bilo je ',brekv,' relacija ekvivalencije');
    writeln('Bilo je ',brpor,' relacija poretka');
    writeln('Bilo je ',brnekor,' nekorektnih relacija');
    close(f);
  end;

begin
 uradi;
end.
```

**156. (PASCAL)** Mali Đurica je počeo da skija. Đurica je egzibicionista i ne skija po stazama, već skija kako stigne. Zbog toga, služba spasavanja je rešila da napravi program koji će izračunati sva moguća mesta na kojima se Đurica može naći. Mapa terena se zadaje kao matrica dimenzija m × n, gde svaki element matrice $h[i, j]$, predstavlja visinu mesta sa koordinatama $(i, j)$. Đurica se spusta uvek u pravcu samo jedne koorditane (ne dijagonalno), i to isključivo sa mesta koje je više, na mesto koje je niže ($h[istart, jstart] > \ldots > h[i, j] > \ldots > h[ikraj, jkraj]$). Za zadato startno mesto i mapu terena, treba odštampati broj mesta na kojima se Đurica može naći (uključujući i startno mesto).

Ulazni podaci nalaze se u fajlu ZAD2.DAT. U prvom redu ulaznog fajla nalaze se celi brojevi $m$ i $n$ ($1 \le m \le 100$, $1 \le n \le 100$), gde je $m$ broj redova, a $n$ broj kolona mape terena. U sledećih $m$ redova nalazi se po $n$ celih brojeva koji predstavljaju elemente matrice $h[i, j]$ ($0 \le h[i, j] \le 10000$). U sledećem redu nalaze se brojevi $r$ i $k$ koji predstavljaju red i kolonu startnog mesta ($1 \le r \le m$, $1 \le k \le n$).

Izlazne podatke treba upisati u fajl ZAD2.RES. Fajl treba da sadrži samo jedan broj *x*, koji predstavlja broj mesta na kojima se Đurica može nalaziti:

```
{A+,B-,D+,E+,F+,G+,I+,L+,N-,O+,P-,Q+,R+,S+,T-,V+,X+,Y+}
{M 65520,300000,600000}

const inp='zad2.dat';
      out='zad2.res';

var a:array[0..101,0..101] of 0..10000;
    mark:array[0..101,0..101] of boolean;
    n,m,i,j,r,k,br:integer;
    f:text;

procedure radi(x,y:byte);
  begin
    inc(br);
    mark[x,y]:=true;
    if x<m then
      if (a[x+1,y]<a[x,y]) and not mark[x+1,y] then
         radi(x+1,y);
    if y<n then
      if (a[x,y+1]<a[x,y]) and not mark[x,y+1] then
          radi (x,y+1);
    if x>1 then
      if (a[x-1,y]<a[x,y]) and not mark[x-1,y] then
          radi(x-1,y);
    if y>1 then
      if (a[x,y-1]<a[x,y]) and not mark[x,y-1] then
          radi(x,y-1);
  end;

begin
  assign(f,inp); reset(f);
  read(f,m,n);
  for i:=1 to m do
    for j:=1 to n do read(f,a[i,j]);
  read(f,r,k); br:=0; close(f);
  radi (r,k);
  assign(f,out); rewrite(f);
  write(f,br); close(f);
end.
```

**Test primeri:**

| ZAD2.DAT | ZAD2.RES | Objašnjenje: Sva mesta na kojima se Đurica može naći su podvučena. |
|---|---|---|
| 6 7<br>0 0 0 10 10 10 10<br>0 0 0 10 10 10 10<br>10 2 10 6 10 0 1<br>10 3 10 7 7 4 2<br>10 4 5 7 8 10 10<br>10 3 10 10 9 10 10<br>6 5 | 14 | 0 0 0 10 10 10 10<br>0 0 0 10 10 10 10<br>10 2 10 6 10 0 1<br>10 3 10 7 7 4 2<br>10 4 5 7 8 10 10<br>10 3 10 10 9 10 10 |

**60. (C)** Napisati proceduru kojom se učitavaju elementi pravougaone matrice *x* dimenzija *m*×*n*, gde je *m* i *n* <=10. Napisati funkciju koja određuje najdužu rastuću seriju brojeva u nizu celih brojeva. Koristeći ovu funkciju, napisati funkciju koja određuje indeks vrste u matrici *x* koja sadrži najdužu rastuću seriju.

**Ulaz:** Pozitivni celi brojevi *m* i *n* koji predstavljaju dimenzije matrice *x*, a zatim *m*× *n* elemenata matrice *x*.
**Izlaz:** Redni broj vrste u matrici *x* koja sadrži najdužu rastuću seriju brojeva.

```
#include<stdio.h>
int naj(int x[], int n)
  { int i;
    float s, max;
    s=max=1;
    for(i=1;i<n; i++)
       if(x[i]>x[i-1]) s++;
       else
         {
          if (s>max) max=s;
          s=1;
         }
    if(s>max) max=s;
    return(max);
  }
void ucitaj(int x[][10], int m, int n)
  { int i,j;
     for(i=0; i<m; i++)
       for(j=0; j<n; j++) scanf("%d", &x[i][j]);
  }
int najduza(int a[][10], int m, int n)
  { int i,max=0, k, ind;
    for(i=0; i<m; i++)
      { k=naj(a[i],n);
        if(k>max)
          { max=k; ind=i;  }
      }
    return(ind+1);
  }

void main()
 { int x[10][10];
   int mx, m, n;
   scanf("%d%d", &m,&n);
   ucitaj(x,m,n);
   mx=najduza(x,m,n);
   printf("Vrsta sa najduzom serijom je %d.\n",mx);
 }
```

**Test primeri:**

```
3 5
2  7  -2  3  4
1  1  0  -2  3
2  3  4   5  0

Vrsta sa najduzom serijom je 3.
```

```
2 3
2  3  4
2  1  3


Vrsta sa najduzom serijom je 1.
```

**152. (C)** Napisati program koji kvadratnu matricu dimenzije dim×dim popunjava prirodnim brojevima od 1 do $dim^2$ po spirali. Očigledno je da će maksimalni upisan broj biti $dim^2$. Suština zadatka je naći uslov koji treba da ispunjavaju vrste i kolone da bi došlo do skretanja u određenom pravcu. Za skretanje udesno uslov je da je i+j<dim+1. Za skretanje naniže uslov je da je i<j. Za skretanje ulevo uslov je da je i+j<>dim+1. Za skretanje naviše uslov je da je i>j+1.

```
void main()
 { int i,j,dim,brojac,kraj;int spirala[21][21];
    do  { printf("\nUnesite dimenziju matrice: ");
          scanf("%d",&dim);    }
    while (dim<1 || dim>20);
    brojac=0;i=1;j=0;kraj=dim*dim;
    do
      {
```

```
       while ((brojac < kraj) && (i+j<dim+1))
             { j++;brojac++; spirala[i][j]=brojac;}
       while ((brojac < kraj) && (i<j))
             { i++;brojac++; spirala[i][j]=brojac;}
       while ((brojac < kraj) && (i+j!=dim+1))
             { j--;brojac++; spirala[i][j]=brojac;}
       while ((brojac < kraj) && (i>j+1))
             { i--;brojac++; spirala[i][j]=brojac;}
       }
    while (brojac!=kraj);

    printf("\n");
    for (i=1;i<=dim;i++)
     {  for (j=1;j<=dim;j++)
            printf("%3d ",spirala[i][j]);
         printf("\n");
     }
 }
```

Rešenje 2.

```
#include <stdio.h>
void main()
 { int a[100][100];    int i,j,p,q,b,n,z,m,k,e,k2,e2,r;
    scanf("%d%d",&n,&m);
    for (i=1;i<=n;i++)
       for (j=1;j<=m;j++) a[i][j]=0;
    i=1;j=1;b=1;k=n;e=m;p=0;q=1;k2=1;e2=1;z=n*m;
    while (b<=z)
     { a[i][j]=b;
        if((j==e)&&(i==k2)) { p=1; q=0; };
        if((i==k)&&(j==e)) { q=-1; p=0; };
        if((i==k)&&(j==e2)) { p=-1; q=0; };
        if((j==e2)&&(i==k2)) { p=0; q=1;};
        if((j==e2)&&(i==k2+1))
         { p=0;q=1;k2++;e2++;e--;k--; };
       b++; i=i+p; j=j+q;
     }
   for (i=1;i<=n;i++)
   { printf("\n");
      for (j=1;j<=m;j++)  printf("%4d",a[i][j]);
   }
 }
```

**Test primer:**

```
Unesite dimenziju matrice:15
 1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
56  57  58  59  60  61  62  63  64  65  66  67  68  69  16
55 104 105 106 107 108 109 110 111 112 113 114 115  70  17
54 103 144 145 146 147 148 149 150 151 152 153 116  71  18
53 102 143 176 177 178 179 180 181 182 183 154 117  72  19
52 101 142 175 200 201 202 203 204 205 184 155 118  73  20
51 100 141 174 199 216 217 218 219 206 185 156 119  74  21
50  99 140 173 198 215 224 225 220 207 186 157 120  75  22
49  98 139 172 197 214 223 222 221 208 187 158 121  76  23
48  97 138 171 196 213 212 211 210 209 188 159 122  77  24
47  96 137 170 195 194 193 192 191 190 189 160 123  78  25
```

**153. (C)** Napisati program kojim se u matrici dimenzija m×n upisuju brojevi do zadatog broja, prateći put kuglice koja polazi od elementa sa koordinatama (0,0) i odbija se od zidova matrice. Ostali elementi matrice su 0.

```
#include<stdio.h>
main()
 { int a[100][100];
   int i,j,p,q,n,m,b,k;
   scanf("%d",&n); scanf("%d",&m);
   for (i=0;i<n;i++)
      for (j=0;j<m;j++) a[i][j]=0;
   scanf("%d",&k);
   i=0;j=0; p=q=1; b=2;
   a[i][j]=1;i=j=1;
   while (b<=k)
      { a[i][j]=b; b++;
        if ((i==(n-1))||(i==0)) p=p*(-1);
        if ((j==(m-1))||(j==0)) q=q*(-1);
        i=i+p;  j=j+q;
      }
   for (i=0;i<n;i++)
    { printf("\n");
       for (j=0;j<m;j++)
          printf("%d\t",a[i][j]);
    }
}
```

**Test primeri:**

| | |
|---|---|
| 2 4 4<br><br>1 0 3 0<br>0 2 0 4 | 3 3 4<br><br>1 0 0<br>0 4 0<br>0 0 3 |

Još jedno rešenje

```
#include <stdio.h>

typedef int mat[100][100];

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

void main()
{ int n,m;
      mat a = {{0,0}};
      int cnt;
      scanf("%d %d %d", &m, &n, &cnt);
      int dx=1;
      int dy=1;
      int cx=0;
      int cy=0;
      int curr=0;
      while (cnt--)
      {
            a[cx][cy] = ++curr;
            cx+=dx;
            cy+=dy;
            if ((cx>=m)||(cx<0))
            {
                  dx*=-1;
```

```
                cx+=2*dx;
            }
            if ((cy>=n)||(cy<0))
            {
                dy*=-1;
                cy+=2*dy;
            }
      }

      ispis(m, n, a);

}
```

**154. (C)** Napisati program koji učitava dimenziju šahovske table (broj kvadrata) i veličinu svakog kvadrata na tabli, a zatim generiše matricu koja odgovara uokvirenoj šahovskoj tabli. U gornjem levom uglu šahovske table uvek je crni kvadrat. Crnom polju odgovara znak '*', a belom znak ' '.

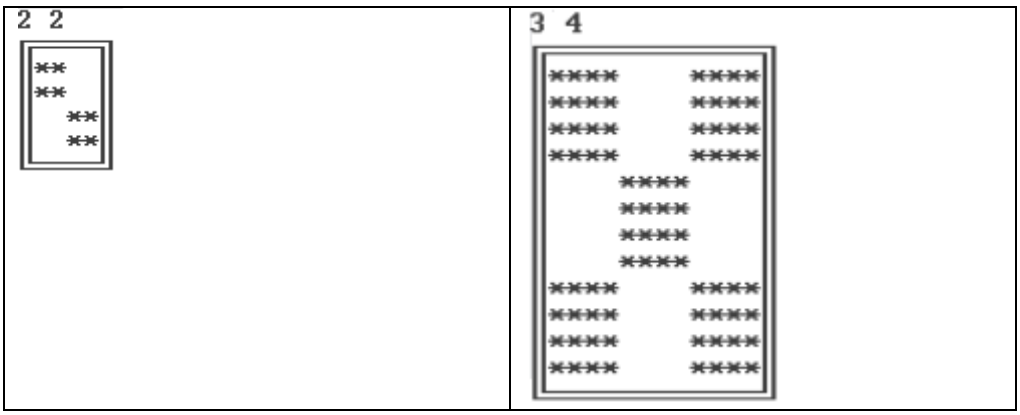
```
#include<stdio.h>

  void promeni(char *ch)
    { if((*ch)== '*') *ch=' ';
      else *ch='*';
    }

  void generisi(int d,int v, char a[80][80])
   { int i,j,k,l,p,q;
    char znak,znak1;
      p=1; q=1; znak1='*';
      for(i=1; i<=d; i++)
       { for(k=1; k<=v; k++)
          { znak=znak1;
            for(j=1; j<=d; j++)
              { for(l=1; l<=v; l++)
                 { if(q>d*v) q=1;
                   a[p][q]=znak;    q=q+1;
                 }
                promeni(&znak);
              }
              p=p+1;
            }
          promeni(&znak1);
        }
        a[0][0]=(char)201;
        a[0][d*v+1]=(char)187;
        a[d*v+1][0]=(char)200;
        a[d*v+1][d*v+1]=(char)188;
        for(j=1; j<=d*v; j++)
          { a[0][j]=(char)205;a[d*v+1][j]=(char)205; }
        for(i=1; i<=d*v; i++)
          { a[i][0]=(char)186;a[i][d*v+1]=(char)186; }
    }

 void main()
  { char a[80][80];
    int i,j,dim,vel;
    clrscr();
    scanf("%d%d",&dim,&vel);
    generisi(dim,vel,a);
    for(i=0; i<=dim*vel+1; i++)
      { for(j=0; j<=dim*vel+1; j++) putchar(a[i][j]);
         printf("\n");
      }
```

```
}
```

Rešenje 2.

```
#include<stdio.h>

 void generisi(int d,int v, char a[80][80])
  { int i,j;
       for(i=1; i<=d*v; i++)
        for(j=1; j<=d*v; j++)
          if( ((i-1)/v+(j-1)/v)%2==0)
               a[i][j]='*';
          else a[i][j]=' ';
       a[0][0]=(char)201;
       a[0][d*v+1]=(char)187;
       a[d*v+1][0]=(char)200;
       a[d*v+1][d*v+1]=(char)188;
       for(j=1; j<=d*v; j++)
         { a[0][j]=(char)205;
           a[d*v+1][j]=(char)205;
           a[j][0]=(char)186;
           a[j][d*v+1]=(char)186; }
   }

void main()
 { char a[80][80];
   int i,j,dim,vel;
   clrscr();
   scanf("%d%d",&dim,&vel);
   generisi(dim,vel,a);
   for(i=0; i<=dim*vel+1; i++)
     { for(j=0; j<=dim*vel+1; j++)
       putchar(a[i][j]);
       printf("\n");
     }
 }
```

**Test primeri:**




**Primer. (C)** U svakom od tri kontejnera se nalazi izvesna količina smeđih, zelenih i belih boca koje treba da se recikliraju. Da bi staklo moglo da se reciklira u svakom kontejneru moraju da se nalaze boce iste boje. Napisati program koji određuje minimalni broj premeštanja boca tako da posle premestanja u svakom kontejneru budu boce iste boje. Pri jednom premeštanju se prebacuje jedna boca iz bilo kog u neki drugi kontejner, a kontejneri su dovoljno veliki da mogu da sadrže sve boce.

U tekstualnom fajlu **zad2.dat** nalaze se 3 reda sa po 3 broja u svakom, koji opisuju početno stanje: broj smeđih, zelenih i belih boca u prvom kontejneru, zatim u drugom i na kraju u trećem. U

izlazni fajl **zad2.res** ispisati minimalan broj premeštanja koji je potreban da se postigne to završno stanje.

Test Primer:

| **zad2.dat** | **zad2.res** |
|---|---|
| 1 2 3<br>4 5 6<br>7 8 9 | 30 |

```
#include<stdio.h>
int main(){
    FILE *f,*g;
    int i,j,k,m[3][3],r,a;
    f=fopen("zad2.dat","r");
    g=fopen("zad2.res","w");
    for(i=0;i<3;i++)
          for(j=0;j<3;j++)
                fscanf(f,"%d",&m[i][j]);
    r=-1;
    for(i=0;i<3;i++)
          for(j=0;j<3;j++)
                for(k=0;k<3;k++)
                      if ((i!=j)&&(i!=k)&&(j!=k)){

    a=m[0][i]+m[1][i]+m[0][j]+m[2][j]+m[1][k]+m[2][k];
                            if(a<r) r=a;
                            else if(r==-1) r=a;
                      }
    fprintf(g,"%d",r);
    fclose(f);
    fclose(g);
    return 0;
}
```

**Kratko objašnjenje rešenja**

Za rešavanje ovog zadatka dovoljne su nam 2. Prva for petlja koja se sastoji iz dve for petlje služi za učitavanje broja flaša koje se nalaze u kontejnerima i za smeštanje tih brojeva u matricu *m* dimenzije 3×3.

Druga složena petlja, koja se sastoji od tri for petlje ispituje sva moguća premestanja flaša tako da novodobijeni raspored zadovoljava postavljeni uslov. Od svih premestanja bira se premestanje sa najmanjim brojem poteza i broj premeštanja se smesta u promenljivu r, koja se zatim i upisuje u izlaznu datoteku kao krajnji i tačni rezultat našeg programa.

## 6.5.3. Matrice i dinamička alokacija memorije

**1.** Dinamička matrica se predstavlja strukturom koja sadrži dimenzije i pokazivač na elemente matrice. Sastaviti na jeziku C program za transponovanje matrice celih brojeva. Polaznu matricu *a* i transponovanu matricu *b* smestiti u dinamičku zonu memorije i deklarisati ih sa

```
int **a, **b.
```

Program bi trebalo da u beskonačnom ciklusu obrađuje proizvoljan broj početnih matrica. Beskonačni ciklus prekinuti kada se za dimenzije ulazne matrice zada vrednost koja je ≤ 0.

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

void main()
{ int **a, **b, m,n,i,j;
```

```
  while(1)
  {  printf("\nBroj vrsta i kolona? "); scanf("%d%d", &m,&n);
     if(m<=0 || n<=0)break;
       a=(int **)malloc(m*sizeof(int *));
       for(i=0; i<m; i++)
       { a[i]=(int *)malloc(n*sizeof(int));
         printf("Elementi %2d. vrste:\n",i+1);
         for(j=0; j<n; scanf("%d",&a[i][j++]));
       }
       b=(int **)malloc(n*sizeof(int *));
       for(i=0; i<n; i++)
       { b[i]=(int *)malloc(m*sizeof(int));
       for(j=0; j<m; j++)b[i][j]=a[j][i];
       }
     printf("Transponovana matrica:\n");
       for(i=0;i<n;i++)
       { for(j=0;j<m;printf("%d  ",b[i][j++]));
         printf("\n");
       }
  }
  for(i=0; i<m; free(a[i++])); free(a);
  for(i=0; i<n; free(b[i++])); free(b);
}
```

U programu koji sledi rešen je isti problem, samo što su napisane funkcije za učitavanje elemenata matrice kao i za transponovanje matrice.

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

void Citaj(int*** a, int* n, int* m)
{     int i, j;
      scanf("%d%d", n, m);

      *a = (int**) malloc(*n*sizeof(int*));
      for (i = 0; i < *n; i++)
            (*a)[i] = (int*) malloc(*m*sizeof(int));
      for (i = 0; i < *n; i++)
            for (j = 0; j < *m; j++)
                  scanf("%d", *(*a+i)+j); // ili &((**a)[i][j])

}

void Transponuj(int*** at, int** a, int n, int m)
{     int i, j;
      *at = (int**) malloc(m*sizeof(int*));
      for (i = 0; i < m; i++)
            (*at)[i] = (int*) malloc(n*sizeof(int));
      for (i = 0; i < m; i++)
            for (j = 0; j < n; j++)
                  (*at)[i][j] = a[j][i];
}

int main()
{     int n, m, i, j;
      int** a;    int** at;
      Citaj(&a, &n, &m);
      for (i = 0; i < n; i++)
      {  for (j = 0; j < m; j++)printf("%d ", a[i][j]);
         printf("\n");
      }
```

```
      printf("\n");

      Transponuj(&at, a, n, m);

      for (i = 0; i < m; i++)
      {  for (j = 0; j < n; j++)  printf("%d ", at[i][j]);
         printf("\n");
      }

      for (i = 0; i < n; i++) free(a[i]);
      free(a);
      for (i = 0; i < m; i++) free(at[i]);
      free(at);
      return 0;
}
```

---

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

void Citaj(int*** a, int* n, int* m)
{     int i, j;
      scanf("%d%d", n, m);

      *a = (int**) malloc(*n*sizeof(int*));
      for (i = 0; i < *n; i++)
            (*a)[i] = (int*) malloc(*m*sizeof(int));
      for (i = 0; i < *n; i++)
            for (j = 0; j < *m; j++)
                  scanf("%d", *(*a+i)+j); // ili &((**a)[i][j])

}

void Transponuj(int** at, int** a, int n, int m)
{     int i, j;
      for (i = 0; i < m; i++)
            for (j = 0; j < n; j++)
                  at[i][j] = a[j][i];
}

int main()
{     int n, m, i, j;
      int** a;    int** at;
      Citaj(&a, &n, &m);
      at = (int**) malloc(m*sizeof(int*));
      for (i = 0; i < m; i++)
            at[i] = (int*) malloc(n*sizeof(int));

      for (i = 0; i < n; i++)
      {  for (j = 0; j < m; j++)printf("%d ", a[i][j]);
         printf("\n");
      }
      printf("\n");

      Transponuj(&at, a, n, m);

      for (i = 0; i < m; i++)
      {  for (j = 0; j < n; j++)  printf("%d ", at[i][j]);
         printf("\n");
      }
```

```
        for (i = 0; i < n; i++) free(a[i]);
        free(a);
        for (i = 0; i < m; i++) free(at[i]);
        free(at);
        return 0;
}
```

# 6.6. STRINGOVI

U programskim jezicima, string je sekvenca karaktera. String konstanta može da ima proizvoljan broj karaktera, uključujući i string bez karaktera. Broj karaktera u stringu se naziva dužina stringa. Jedinstveni string koji ne sadrži karaktere naziva se *prazan string*.

Stringovi su podržani u svim modernim programskim jezicima. Tipične operacije nad stringovima su:

- dužina (length);
- upoređenje na jednakost (equality comparison);
- leksikografsko poređenje (lexicographic comparison);
- selekcija karaktera u stringu (character selection);
- selekcija podstringa (substring selection);
- nadovezivanje (concatenation);
- konverzija stringa u numeričku vrednost i obratno.

## 6.6.1. Stringovi u C

U jeziku C, string je jednodimenzionalni niz elemenata tipa *char* koji se završava karakterom '\0'. String se može posmatrati i kao pointer na *char*.

### Inicijalizacija i obrada stringova

Karakteru u stringu može da se pristupi ili kao elementu niza ili pomoću pointera na *char*. Svaki string se završava karakterom '\0'. String konstanta je proizvoljan niz znakova između navodnika. Svi znaci zahvaćeni znacima navoda, kao i *null* znak '\0' smeštaju se u uzastopnim memorijskim lokacijama.

Na primer, string *s* = "*ABC*" može da se zapamti u memoriju kao sekvenca od 4 karaktera *s*[0] = '*A*', *s*[1] = '*B*', *s*[2] = 'C', *s*[3] = '\0', od kojih je poslednji null karakter '\0'. Bez poslednjeg karaktera '\0' niz nije kompletiran, te predstavlja samo niz karaktera.

**Primer.** Dodeljivanje početne vrednosti string promenljivoj.

```
#define MAXREC 100
void main()
 { char w[MAXREC];
        …
   w[0]='A'; w[1]='B'; w[2]='C'; w[3]='\0';
  }
```

Sada je jasno da su karakter '*a*' i string "*a*" dva različita objekta. String "*a*" je niz koji sadrži dva karaktera: '*a*' i '\0'.

Stringovi su nizovi karaktera, pa se mogu inicijalizovati na mestu deklaracije, kao i nizovi brojeva. Osim toga, praktičnije je da se string inicijalizuje konstantnom vrednošću tipa string umesto "znak po znak".

**Primer.** Možemo pisati

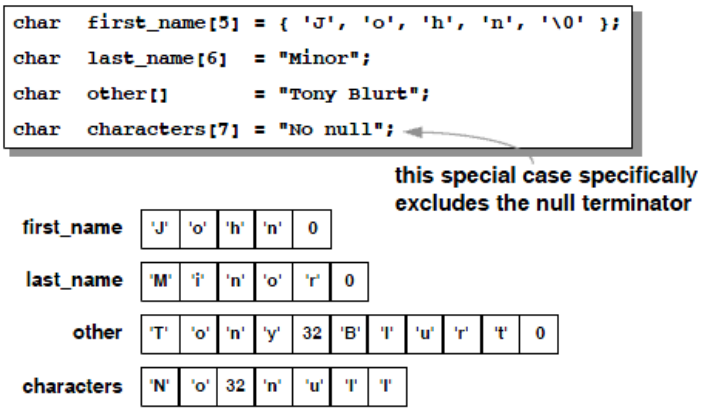
`char s[]="abc";` umesto `char s[]={'a', 'b', 'c', '\0'};`

Takođe, može se inicijalizovati pointer na char kao konstantni string:

```
char *p="abc";
```

Efekat ove naredbe da se string "*abc*" smešta u memoriju, a pointer *p* se inicijalizuje baznom adresom tog stringa.

U slučaju kada se niz znakova zadate dužine inicijalizuje string konstantom, tada se neiskorišćeni elementi niza inicijalizuju null znakom '\0'.




**Primer.** Transformacija stringa: u stringu koji se učitava zameniti svaki znak '*e*' znakom '*E*', dok se svaka praznina zamenjuje prelazom u novi red i tabulatorom. Učitavanje znakova u stringu se završava prelazom u novi red.

```
#include <stdio.h>
#define MAX 50
void main()
 { char line[MAX], *change(char *);
    void read(char *);
    printf("Unesi jedan red:  ");
    read(line);
    printf("\n%s\n\n%s\n\n","Posle promene red je: ", change(line));
 }

void read(char *s)
 { int c, i=0;
    while ((c=getchar()) !='\n') s[i++]=c;
    s[i]='\0';
 }

char *change(char *s)
 { static char new_string[MAX];
    char *p=new_string;
    /* pointer p je inicijalizovan na  baznu adresu od new_string */
    for (; *s != '\0'; ++s)
       if(*s=='e') *p++='E';
       else if(*s==' ')  { *p++='\n'; *p++='\t'; }
       else *p++=*s;
    *p='\0';
    return(new_string);
 }


#include <stdio.h>

char s[105];
```

```
int main()
{
	//scanf("%[^\n]s",s);
	gets(s);
	int p=0;
	int cnt=0;
	while (s[p])
		cnt+=s[p++]==' ';
	p=0;
	while (s[p])			p++;
	int pl=p+cnt-1;
	while (p)
	{
		if (s[p-1]=='e')
			s[p-1]='E';
		if (s[p-1]==' ')
		{
			p--;
			s[pl]='\t';
			pl--;
			s[pl]='\n';
			pl--;
		}
		else
		{
			p--;
			s[pl]=s[p];
			pl--;
		}
	}
	puts(s);
}
```

## Testiranje i konverzija znakova

Standardna biblioteka <ctype.h> sadrži deklaracije funkcija za testiranje znakova. Sve funkcije iz te biblioteke imaju jedan argument tipa *int*, čija vrednost je ili *EOF* ili je predstavljena kao unsigned char. Rezultat svake od ovih funkcija je tipa *int*. Funkcije vraćaju rezultat različit od nule (tačno), ako argument zadovoljava opisani uslov, a 0 ako ne zadovoljava. Najčešće korišćene funkcije iz ove biblioteke su:

`isdigit(c)` ispituje da li je *c* decimalna cifra;
`islower(c)` ispituje da li je *c* malo slovo;
`isupper(c)` ispituje da li je *c* veliko slovo;
`isalpha(c)` ispituje da li je *islower*(c) ili *isupper*(c) tačno;
`isalnum(c)` ispituje da li je *isalpha*`(c)` ili *isdigit*(c) tačno;
`iscntrl(c)` ispituje da li je *c* kontrolni znak (kontrolni znaci imaju kodove 0... 31);
`isspace(c)` razmak, novi red, povratnik, vertikalni ili horizontalni tabulator.

Takođe, postoje i dve funkcije za promenu veličine slova.

`int tolower(int c)` konvertuje *c* u malo slovo;
`int toupper(int c)` konvertuje *c* u veliko slovo.

Ako je karakter *c* veliko slovo, tada funkcija *tolower*(*c*) vraća odgovarajuće malo slovo, a inače vraća *c*. Ako je karakter *c* malo slovo, tada funkcija *toupper*(*c*) vraća odgovarajuće veliko slovo, a inače vraća nepromenjeno *c*.

**Primer.** Prebrojavanje reči unutar stringa. String se završava prelazom u novi red ili karakterom *EOF*. Reči unutar stringa su razdvojene praznina.

```
#include <stdio.h>
#include <ctype.h>
#define MAX 30
void main()
 { char line[MAX];
    void read(char *);
    int cnt(char *);
    printf("\n Unesi string : "); read(line);
    printf("\n Broj reci = %d\n", cnt(line));
 }

void read(char *s)
 { int c,i=0;
    while ((c=getchar()) !=EOF && c!='\n')      s[i++]=c;
    s[i]='\0';
 }

int cnt(char *s)
 { int br=0;
    while (*s!='\0')
     { while (isspace(*s)) ++s;
        /* preskoci praznine izmedju reci na pocetku */
        if(*s!='\0') /* naci rec */
         { ++br;
           while(!isspace(*s)&& *s!='\0')     /*preskoci rec*/
               ++s;
         }
     }
    return(br);
 }
```

**Primer.** String je deklarisan kao statički niz karaktera a zatim je inicijalizovan nekom string konstantom. Napisati program koji ispisuje string u direktnom i inverznom poretku.

```
#include<stdio.h>
void main()
 { static char s[]="Konstantni string";
   char *p;
    p=s;   printf("\nIspis stringa:\n");
    while(*p)      putchar(*p++);
    printf("\nIspis u inverznom poretku:\n");
    while(--p>=s)  putchar(*p);
    putchar('\n');
 }
```

## Učitavanje i ispis stringova

Vrednosti stringova se mogu i učitavati. Pri učitavanju stringova bitno je da se rezerviše dovoljno memorijskog prostora za smeštanje stringova. Program ne može da predvidi maksimalnu dužinu stringa i da rezerviše potreban memorijski prostor. Zbog toga je programer obavezan da predvidi maksimalnu dužinu stringova koji se učitavaju. String može da dobije vrednost pomoću funkcije *scanf*(), koristeći format %*s*. Na primer, možemo pisati:

```
scanf("%s", w);
```

Svi znaci, do prvog znaka *EOF* ili do prve praznine se uzimaju kao elementi stringa *w*. Posle toga se *null* karakter stavlja na kraj stringa. Napomenimo da nije korišćena adresa *&w* u drugom argumentu funkcije *scanf*(). Kako je ime niza u stvari bazna adresa niza, očigledno je *w* ekvivalentno sa *&w*[0]. Stoga je operator adresiranja ispred imena niza *w* nepotreban.

Za učitavanje stringova najčešće se koristi funkcija *gets*(). Ovom funkcijom se prihvataju svi znaci sa tastature, sve dok se ne unese znak za novi red '\n'. Znak za prelaz u novi red se ignoriše, dok se svi

ostali karakteri dodeljuju stringu koji se učitava. Kao poslednji znak učitanog stringa uvek se postavlja *null* karakter '\0'.

Prototip funkcije *gets*() je oblika `char *gets(char *)`

Ukoliko je učitavanje završeno korektno, vrednost ove funkcije je pokazivač na adresu u kojoj je smešten prvi znak stringa *s*. U suprotnom slučaju, vraćena vrednost je NULL (nulta adresa, koja je u datoteci *stdio.h* definisana sa 0). To znači da se kontrola pravilnosti ulaza može kontrolisati izrazom

```
while(gets(ime)==NULL);
```

## Printing Strings

- Strings may be printed by hand
- Alternatively printf supports "%s"

```
char   other[] = "Tony Blurt";
```

```
char   *p;

p = other;
while(*p != '\0')
      printf("%c", *p++);
printf("\n");
```

```
int    i = 0;

while(other[i] != '\0')
      printf("%c", other[i++]);
printf("\n");
```

```
printf("%s\n", other);
```

**Primer.** Učitavanje stringa i njegovo prikazivanje na ekran.

```
#include<stdio.h>
void main()
 { char ime[100];     char *pok;
   pok=gets(ime);
   printf("1. nacin: %s\n",ime);    printf("2. nacin: %s\n",pok);
 }
```

## Assigning to Strings

Strings may be initialised with "=", but not assigned to with "="

Remember the name of an array is a CONSTANT pointer to the zeroth element

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

int    main(void)
{
       char   who[] = "Tony Blurt";
       who = "John Minor";
       strcpy(who, "John Minor");
       return 0;
}
```

**Primer.** Reč je data kao niz od *n* karaktera (*n*<300). Napisati program koji nalazi najduži palindrom u toj reči. Palindrom je podniz od uzastopnih elemenata koji se jednako čita s leva i s desna.

```
#include <stdio.h>
void main()
 { char a[300];
   int i,j,n,maxi,maxl,maxj,l;
   int pal(char *, int, int);
   printf("Unesi broj karaktera reci\n"); scanf("%d",&n);
   printf("Unesi rec:\n");   scanf("%c", &a[0]);
   for (i=0; i<n; i++) scanf("%c", &a[i]);
   a[i]='\0';
   maxl=0;
   for (i=0; i<n; i++)
      for (j=i; j<n; j++)
         { if(pal(a,i,j))
            { l=(j-i)+1; if (l>maxl){ maxl=l; maxj=j; maxi=i; }
            }
         }
   printf("\nNajduzi palindrom je od pozicije %d do %d\n", maxi,maxj);
   for (i=maxi; i<=maxj; i++) printf("%c",a[i]);
   printf("\n");
 }

 int pal (char *a, int i, int j)
  { int p;    int l,k,m;    char b[300];
    l=(j-i)+1;    k=0;
    for (m=i; m<=j; m++)     b[++k]=a[m];
    p=1; k=1;
    while(p && k<=l/2)
     if(b[k] != b[l-k+1]) p=0;
       else k++;
    return(p);
 }
```

Još jedno rešenje za funkciju *pal* dato je u sledećem kodu

```
int pal (char *a, int i, int j)
  { int m;    char *p, *q;
    p=q=a;
    for(m=0; m<i; m++){ p++; q++; }
    for(m=i; m<j; m++)q++;
    while(q>=p)
       if(*q != *p) return 0;
       else { p++; q--; }
    return 1;
 }
```

**Primer.** U beskonačnom ciklusu se učitavaju stringovi i ispituje da li su palindromi. Stringovi se tretiraju kao pointeri na karaktere.

```
void main()
 { char s[30];
    printf("\n Stringovi u beskonačnom ciklusu\n");
    while(1)
      { printf("Zadati string\n");        gets(s);
        if(palindrom(s)) printf("Jeste palindrom\n");
        else printf("Nije palindrom\n");
      }
  }

 int palindrom(char *strpok)
    { char *strpok1=strpok;
       while(*strpok1)++strpok1;
```

```
        --strpok1;
        while(strpok < strpok1) if(*strpok++ != *strpok1--) return(0);
        return(1);
    }
```

**Primer.** Ispitati da li je uneti string palindrom

```
#include<cstdio>
#include<cstring>
#include<algorithm>

void main()
{
      char s[100], s2[100];
      int i,j,n,b;
      scanf("%s",s);
      n=strlen(s);
      i=0; b=0;
      j=n-1;
      while ((i<n/2)&&(j>n/2)&&(b==0))
      {
            if (s[i]!=s[j]) b++;
            i++;
            j--;
      }
      if (b==0) printf("Palindrom je.\n");
      else printf("Nije palindrom\n");

      char *a,*c;
      a=s;
      c=s;
      while (*c) c++;
      c--;
      b=1;
      while ((a<c)&&b)
      {
      if (*a!=*c) b=0;
      else {a++; c--;}
      }
      if (b) printf("Palindrom je.\n");
      else printf("Nije palindrom\n");

      strcpy(s2,s);
      std::reverse(s2,s2+strlen(s2)-1);
      printf("%s\n",
           (!strcmp(s, s2)) ? "Nije palindrom." : "Jeste palindrom.");
}
```

**Primer.** Učitava se tekst, sve do znaka *ENTER*. Štampati izveštaj, u kome se nalaze dužine reči sadržanih u tekstu, kao i broj reči date dužine.

```
#include<stdio.h>
#define MAX 80
int separator(char ch)
 { return(ch==','||ch==' '||ch==';'||ch=='('||ch==')'||
          ch==' '||ch=='!'||ch=='?')||ch=='\n'; }
void main()
 { char ch;
   int i,l=0;
   int b[MAX];
   for(i=1; i<=MAX; i++)b[i]=0;
   while((ch=getchar()) != '\n')
     {  if(!separator(ch)) l++;
```

```
        else if(l>0)       { b[l]++; l=0; }
      }
   ch=getchar();
   if(l>0)       { b[l]++; l=0; }
   printf("%s   %s\n","Duzina reci","Broj ponavljanja");
   for(i=1;i<=MAX;i++)
     if(b[i]>0)   printf("%d                %d\n",i,b[i]);
 }
```

Već je pokazano da se vrednosti stringova mogu ispisivati pomoću funkcije *printf*, koristeći format %*s*. Takođe, za ispis stringova se može koristiti standardna funkcija *puts*. Funkcija *puts* ima jedan argument, koji predstavlja pokazivač na početnu adresu stringa koji se ispisuje. Funkcija *puts* ispisuje sve karaktere počev od pozicije određene argumentom funkcije do završnog karaktera \0'.

**Primer.** Zadati string *s* koristeći funkciju *gets*(). Napisati funkciju za izbacivanje svih nula sa kraja stringa, kao i funkciju za izbacivanje nula sa početka učitanog stringa. U glavnom programu izbaciti sve početne i završne nule u zadatom stringu.

```
#include <stdio.h>
void main()
 { char s[30];
   void ukloni_kraj(char *);
   void ukloni_pocetak(char *);
    printf("\n Zadati string:\n");    gets(s);
    ukloni_kraj(s);
    printf("\nString bez zadnjih nula = ");    puts(s);
    ukloni_pocetak(s);
    printf("\nString bez pocetnih i zadnjih nula = ");  puts(s);
 }

 void ukloni_pocetak(char *s)
  { char *t=s;
     while(*t && *t=='0') t++;
     /* Nađen je prvi karakter razlicit od \0 */
     while(*t) *s++=*t++;
     /* Kopira preostale karaktere na pocetak stringa */
     *s='\0';
  }

 void ukloni_kraj(char *s)
 { char *t=s;
    while(*t)t++;
    t--;
    while(*t=='0')t--;
    t++;   *t='\0';
 }
```

**Primer.** Napisati program kojim se u jednoj liniji unosi korektno zapisan postfiksni izraz, a zatim izračunava njegova vrednost. Operandi u izrazu su prirodni brojevi razdvojeni blanko simbolom, a operacije su iz skupa {+, -, *, /}, pri čemu je / celobrojno deljenje.

Na primer, postfiksnom izrazu 3 2 7 + - 4 * odgovara infiksni izraz (3-(2+7))*4=-24.

```
#include<stdio.h>
#include<string.h>
#include<stdio.h>
void main()
 { char izraz[80], *s;
   int a[41],n,greska,x,st;
   while(1)
    { gets(izraz);   n=0;   greska=0; s=izraz;
      while(*s && !greska)
```

```
        { if(*s>='0' && *s<='9')
           { x=0; st=1;
             while (*s>='0' && *s<='9')
               { x+=(*s-'0')*st; st*=10;  s++; }
             a[n++]=x;
           }
          else if(*s!=' ')
           { switch(*s)
               { case '+': a[n-2]+=a[n-1]; break;
                 case '-': a[n-2]-=a[n-1]; break;
                 case '*': a[n-2]*=a[n-1]; break;
                 case '/': if(a[n-1]) a[n-2]/=a[n-1];
                                      else greska=1;
               }
              n--;
           }
         s++;
        }
      if(!greska) printf("%d\n",a[0]);
      else printf("greska, deljenje nulom\n");
    }
}
```

## Standardne funkcije za rad sa stringovima u C

Standardna biblioteka sadrži veći broj funkcija za manipulaciju stringovima. Ove funkcije nisu deo C jezika, ali su napisane u jeziku C. U njima su često promenljive deklarisane u memorijskoj klasi register, da bi se obezbedila njihovo brže izvršavanje. Sve te funkcije zahtevaju da se string završava null karakterom, a njihov rezultat je integer ili pointer na char. Prototipovi ovih funkcija su dati u header fajlu `<string.h>`.

| | |
|---|---|
| unsigned strlen(char *s) | Prebrojava sve karaktere pre '\0' u *s*, i vraća nađeni broj. |
| int strcmp(char *s1, char *s2) | Rezultat je ceo broj <0, =0 ili >0, zavisno od toga da li je *s*1 u leksikografskom poretku manje, jednako ili veće od *s*2. |
| int strncmp(char *s1, char *s2, int n) | Slična je funkciji strcmp, osim što ova funkcija upoređuje najviše *n* karaktera niske *s*1 sa odgovarajućim karakterima niske *s*2. |
| char *strcat(char *s1, char *s2) | Stringovi *s*1 i *s*2 se spajaju, a rezultujući string se smešta u *s*1. Rezultat je pointer na *s*1. U stvari, povezuje nisku *s*2 sa krajem niske *s*1. Mora da se alocira dovoljno memorije za *s*1. |
| char *strncat(char *s1, char *s2, int n) | Nadovezuje najviše *n* znakova niske *s*2 sa krajem niske *s*1. Nisku *s*1 završava karakterom \0 i vraća *s*1. |
| char *strcpy(char *s1, char *s2) | String *s*2 se kopira u memoriju, počev od bazne adrese na koju ukazuje *s*1. Sadržaj od *s*1 se gubi. Rezultat je pointer *s*1. |
| char *strncpy(char *s1, char *s2, int n) | Kopira najviše *n* znakova stringa *s*2 u memoriju, počev od bazne adrese na koju ukazuje *s*1. Rezultat je pointer *s*1. |
| char *strchr(char *s, char c) | Rezultat primene ove funkcije je pointer na prvo pojavljivanje karaktera *c* u stringu *s*, ili *NULL* ako se *c* ne sadrži u *s*. |
| char *strstr(char *s1, char *s2) | Ova funkcija vraća pokazivač na prvi znak u *s*1 počev od koga se *s*2 sadrži u *s*1. |
| int atoi(char *s) | Ova funkcija prevodi string *s* sastavljen od ASCII znakova u ekvivalentan ceo broj. Ispred decimalnog broja koji je sadržan u *s* može da stoji proizvoljan broj nula, praznina ili |

| | |
|---|---|
| | znakova tabulacije, i ono se ignorišu. String *s* može počinjati znakom minus (-), i tada je rezultat konverzije negativni ceo broj. Broj se završava null karakterom '\0' ili bilo kojim znakom koji nije cifra. Ova funkcija se nalazi u standardnoj biblioteci *stdlib.h*. |
| float atof(char *s) | Ova funkcija prevodi string *s* sastavljen od ASCII znakova u ekvivalentan realni broj. Ova funkcija se nalazi u standardnoj biblioteci *stdlib.h*. |

Funkcija `strlen` može da se implementira na sledeći način:

```
unsigned strlen(register char *s)
{ register unsigned n;
    for (n=0; *s!='\0'; ++s) ++n;
    return(n);
 }
```

**Primer.**

```
char s1[100], s2[100], t[100];
strcpy(s1, "recenica 1"); strcpy(s2, "rec 2");
strlen(s1);      /* 10 */
strlen(s1+9);    /* 1 */
strcmp(s1, s2);  /* pozitivan broj */
s3=strcpy(t,s1+9);   /* t="1" */
strcat(t," ");   /* t="1 " */
strcat(t,s1+9);  /* t="1 1"*/
printf("%s",s3); /* 1 1*/
printf("%s",t); /* 1 1*/
```

**Primer.** Funkcija `strcpy` se može implementirati na sledeći način:

```
char *strpy(char *s1, char *s2)
 { char u;
     u=s2;
     while(s1++=s2++);
     return(u);
 }
```

**Primer.** Kažemo da se string *U* pojavljuje u stringu *T* sa pomakom *s*, ili da se *U* pojavljuje u *T* počev od pozicije s ako su ispunjeni sledeći uslovi:

$$0 <= s <= T_n - U_n,$$

$$T[s+j] = U[j] \text{ za } 0 <= j < U_n,$$

gde je $T_n$ broj elemenata u stringu *T* a $U_n$ dužina stringa *U*.

Odrediti sva pojavljivanja stringa *U* u stringu *T*.

```
#include<string.h>
void sekv_sm (char *t, char *u)
{ int s, j;
  int tn, un;
  tn = strlen(t);   un = strlen(u);
  for (s = 0; s <= tn-un; s++)
  { for (j = 0; j < un; j++)     if (t[s+j] != u[j]) break;
    if(j == un) printf("Uzorak se pojavljuje sa pocetkom %d.\n", s);
  }
}

void main()
 { char *s1, *s2;
```

```
    printf("Prvi string? "); gets(s1);
    printf("Drugi string? "); gets(s2);
    sekv_sm(s1,s2);
  }
```

**Primer.** Napisati C program za uređivanje niza imena po abecednom redosledu. Broj imena nije unapred poznat. Svako ime se učitava u posebnom redu. Niz imena se završava imenom koje je jednako "**...**".

```
#include<stdio.h>
#include<string.h>
#define N 100
#define D 40

void main()
{ char ljudi[N][D+1], osoba[D+1];
  int i,j,m,n=0;
  //Citanje neuredjenog niza imena
  do
    gets(ljudi[n]);
  while(strcmp(ljudi[n++],"...")!=0);
  n--;

  //Uredjivanje niza imena
  for(i=0;i<n-1;i++)
  { m=i;
    for(j=i+1; j<n; j++)
        if(strcmp(ljudi[j],ljudi[m])<0)m=j;
    if(m!=i)
      { strcpy(osoba,ljudi[i]); strcpy(ljudi[i],ljudi[m]);
        strcpy(ljudi[m],osoba);
      }
  }

  //Ispisivanje uredjenog niza imena
  for(i=0; i<n; puts(ljudi[i++]));
}
```

II način

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#define N 100
#define D 40

void main()
{ char **ljudi, osoba[D+1];
  int i,j,m,n=0;
  ljudi=(char **)malloc(N*sizeof(char *));
  //Citanje neuredjenog niza imena
  do
  { ljudi[n]=(char *)malloc((D+1)*sizeof(char));
      gets(ljudi[n]);
  }
  while(strcmp(*(ljudi +n++),"...")!=0);
  n--;

//Uredjivanje niza imena
for(i=0;i<n-1;i++)
{ m=i;
  for(j=i+1; j<n; j++)
        if(strcmp(ljudi[j],ljudi[m])<0)m=j;  // *(ljudi+i)=ljudi[i]
```

```
  if(m!=i)
  { strcpy(osoba,ljudi[i]); strcpy(ljudi[i],ljudi[m]);
    strcpy(ljudi[m],osoba);
  }
}

  //Ispisivanje uredjenog niza imena
  for(i=0; i<n; puts(ljudi[i++]));
}
```

**Primer.** Na kružnom putu dužine *mn* kilometara postoji *n* benzinskjh stanica ($1 < n < 100$, $1 < m < 1000$). Na stanici *i* ima ukupno $s_i$ litara benzina. Između svake dve stanice je rastojanje *m* kilometara. Automobil polazi iz jedne od stanica, a može da započne kretanje u bilo kom od dva smera. Na početku mu je rezervoar prazan, a pri prolazu kroz svaku od stanica uzima celokupnu količinu goriva iz stanice (rezervoar ima neograničenu zapreminu). Automobil troši *b* ($1 \le b \le 10$) litara po kilometru. Odrediti da li postoji stanica iz koje automobil može da započne kretanje i obiđe celu stazu.

U ulaznoj datoteci 'zad3.in' dati su, redom, brojevi *n, m, b,* $s_1$, ..., $s_n$. U izlaznu datoteku 'zad3.out' upisati reč 'MOGUĆE' ako postoji tražena stanica, a u suprotmom reč 'NEMOGUĆE'.

| zad3.in | zad3.out | Objašnjenje |
|---|---|---|
| 3 1 1<br>2 2 0 | MOGUĆE | Na putu su tri stanice. Automobil može da krene iz bilo koje od prve dve. |
| 3 1 2<br>4 0 1 | NEMOGUĆE | Pošto automobil troši 21 na kilometar, iz koje god stanice da pode, ne može da obiđe celu stazu. |

# 6.7. Strukture i nabrojivi tipovi u c

Strukture su složeni tipovi podataka koje se, za razliku od nizova, sastoje od komponenti (segmenata) različitog tipa. Komponente sloga se obično nazivaju polja. Svako polje poseduje ime i tip. Imena polja se grade kao i drugi identifikatori.

## 6.7.1. Članovi strukture

Slično slogovima u Pascalu, strukture dozvoljavaju da se različite komponente ujedine u pojedinačnu strukturu. Komponente strukture su takođe imenovane i nazivaju se članovi (odnosno elementi ili polja). Elementi struktura su različitih tipova generalno, i mogu se prilagođavati problemu. Strukture se definišu pomoću ključne reči *struct*. U najopštijem obliku, strukture se opisuju na sledeći način:

```
struct [oznaka]
 { tip ime_elementa1[, ime_elementa2...];
     …
 }[<ime_promenljive 1>[, <ime_ promenljive 2>...]];
```

Rezervisana reč *struct* služi kao informacija kompajleru da neposredno iza nje sledi opis neke strukture. Zatim sledi neobavezni identifikator, označen sa *oznaka*, koji predstavlja ime strukture. Ime dodeljeno strukturi se može kasnije koristiti pri deklaraciji promenljivih strukturnog tipa. Iza imena strukture, između velikih zagrada, deklarišu se pojedini delovi strukture. Elementi strukture mogu biti proizvoljnih tipova, pa i nove strukture. Iza poslednje zagrade piše se znak ';'. Između zatvorene zagrade } i znaka ; opciono se mogu navesti imena promenljivih strukturnog tipa.

**Primer.** Sledećom definicijom strukture opisani su osnovni podaci o studentu: ime, broj indeksa i upisana godina studija.

```
struct student { char *ime;
                 int indeks;
```

```
        int godina;
    } s1,s2,s3;
```

Osim toga, promenljive *s*1, *s*2, *s*3 se deklarišu kao promenljive koje mogu da sadrže konkretne vrednosti saglasno tipu *struct student*, tj. podatke o studentima. Svaki slog o studentu sadrži tri komponente: *ime, indeks* i *godina*. Komponenta *ime* je string, dok su komponente *indeks* i *godina* celobrojnog tipa.

Struktura može da se koristi kao šablon za deklaraciju tipova podataka. Na primer, neka je deklarisana struktura student na sledeći način:

```
struct student { char *ime;
                 int indeks;
                 int godina;
       };
```

Sada se mogu deklarisati strukturne promenljive strukturnog tipa *struct student*. Na primer, možemo pisati

```
struct student pom, razred[100];
```

Tek posle ovakvih deklaracija se alocira memorija za promenljivu *pom* i niz *razred*.

Objedinjavanje opisa strukture čije je ime izostavljeno sa deklaracijama promenljivih koristi se kada se šablon strukture ne koristi na drugim mestima u programu. Na primer, možemo pisati

```
struct
 { char *ime;
   int indeks;
   int godina;
 } s1,s2,s3;
```

Strukturna promenljiva se može inicijalizovati na mestu svoje deklaracije:

```
static struct student pom= { "Milan", 1799, 3};
```

Članovima strukture se direktno pristupa pomoću operatora '.'.

Na primer, vrednosti članovima strukture student mogu se dodeljivati na sledeći način:

```
strcpy(pom.ime, "Milan");
pom.indeks= 1799;
pom.godina= 3;
```

Takođe, članovima strukture se mogu zadati željene vrednosti pomoću funkcija *scanf*() i *gets*():

```
gets(pom.ime);
scanf("%d",&pom.indeks);
scanf("%d",&pom.godina);
```

Ako se opis strukture navede van svih funkcija, ta struktura se može koristiti u svim funkcijama koje su definisane posle opisa strukture. Uobičajeno je da se definicija strukture navede na početku izvornog programa, pre opisa promenljivih i funkcija. U velikim programima opisi struktura se obično pišu u posebnom fajlu.

## Structure templates are created by using the `struct` keyword

```
struct Date
{
    int  day;
    int  month;
    int  year;
};
```

```
struct Library_member
{
    char         name[80];
    char         address[200];
    long         member_number;
    float        fines[10];
    struct Date  dob;
    struct Date  enrolled;
};
```

```
struct Book
{
    char  title[80];
    char  author[80];
    float price;
    char  isbn[20];
};
```

```
struct Library_book
{
    struct Book           b;
    struct Date           due;
    struct Library_member *who;
};
```

## Creating Instances

**Having created the template, an instance (or instances) of the structure may be declared**

```
struct Date
{
    int  day;
    int  month;
    int  year;
} today, tomorrow;

struct Date next_monday;

struct Date next_week[7];
```

instances must be declared before the ';' ...

... or "struct Date" has to be repeated

an array of 7 date instances

# Initialising Instances

## Structure instances may be initialised using braces (as with arrays)

```
int  primes[7] = { 1, 2, 3, 5, 7, 11, 13 };

struct  Date   bug_day = { 1, 1, 2000 };

struct  Book   k_and_r = {
      "The C Programming Language 2nd edition",
      "Brian W. Kernighan and Dennis M. Ritchie",
      31.95,
      "0-13-110362-8"
};
```

```
struct Book
{
    char   title[80];
    char   author[80];
    float  price;
    char   isbn[20];
};
```

**Primer.** Prebrojati studente treće godine.

Može se prvo napisati fajl *cl.h*:

```
#define N 10
struct student { char ime[30];
                 int indeks;
                 int godina;
               };
```

Ovaj header fajl može se koristiti kao informacija u modulima koje čine program.

```
#include "cl.h"
#include<stdio.h>
#include<string.h>
void main()
 { int broji(student *);
   student s[10];
   int i;
   for(i=0; i<N; i++)
     { printf("%d ti student : ",i);
       gets(s[i].ime);
       scanf("%d", &s[i].indeks); scanf("%d", &s[i].godina);
       getchar();
      }
  printf("Studenata trece godine ima %d\n", broji(s));
}

int broji(student *sts)
 { int i, cnt=0;
   for (i=0; i<N; ++i)
            cnt += sts[i].godina==3;
   return(cnt);
 }
```

Drugi način:

```
#include "cl.h"
#include<stdio.h>
#include<string.h>
void main()
 { int broji(student *);
   void ucitaj(student *s);
   student s[10];
   ucitaj(s);
  printf("Studenata trece godine ima %d\n", broji(s));
}

void ucitaj(student *s)
{ int i;
   for(i=0; i<N; i++)
   { printf("%d ti student : ",i);
       gets(s[i].ime);
       scanf("%d", &s[i].indeks); scanf("%d", &s[i].godina);
       getchar();
   }
}

int broji(student *sts)
 { int i, cnt=0;
   for (i=0; i<N; ++i)
            cnt += sts[i].godina==3;
   return(cnt);
 }
```

Treći način:

```
#include "cl.h"
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
void main()
 { int broji(student *);
   void ucitaj(student *);
   student *s=(student *)malloc(N*sizeof(student));
   ucitaj(s);
  printf("Studenata trece godine ima %d\n", broji(s));
}

void ucitaj(student *s)
{ int i;
   for(i=0; i<N; i++)
   { printf("%d ti student : ",i);
       gets(s[i].ime);
       scanf("%d", &s[i].indeks); scanf("%d", &s[i].godina);
       getchar();
   }
}

int broji(student *sts)
 { int i, cnt=0;
   for (i=0; i<N; ++i)
            cnt += sts[i].godina==3;
   return(cnt);
 }
```

Strukture se mogu inicijalizovati od strane programera na mestu svoje deklaracije.

**Primer.** U sledećem kôdu je definisan nabrojivi tip *Month* i strukturni tip *Date*.

```
enum Month {jan,feb,mar,apr,may,jun,jul,aug,sep,oct,nov,dec};
struct Date {Month m; byte d;};
```

Ova struktura ima skup vrednosti

```
Date = Month×Byte = {jan,feb, ..., dec}×{0, ..., 255}.
```

```
#include<stdio.h>

enum Month {jan,feb,mar,apr,may,jun,jul,aug,sep,oct,nov,dec};
struct Date {
   Month m;
   unsigned short d;
};

void main()
{  struct Date someday = {jan, 1}; // deklaracija i incijalizacija
  //Sledeci kod ilustruje selekciju elemenata strukture
   printf("%d/%d\n", someday.m + 1, someday.d);
   someday.d = 29; someday.m = feb;
   printf("%d/%d\n", someday.m+1, someday.d);
}
```

**Primer.** Primeri inicijalizacije struktura.

1. Struktura *datum*:

```
struct datum
{ int dan, mesec, godina; };
struct datum d={11, 4, 1996};
```

1. Struktura *karta*:

```
enum tip {'p','h','k','t'};
struct karta
{ int vrednost;
  tip boja;
};
struct karta k={12, 't'};
```

**Primer.** Deklaracija strukture datum i inicijalizacija strukturne (i statičke) promenljive danas.

```
#include <stdio.h>
void main()
   { struct datum
        { int dan;
          int mesec;
          int godina;
        } danas={9, 4, 1996};
      printf("%d.%d.%d.\n",danas.dan,danas.mesec,danas.godina);
   }
```

**Primer.** Program za korekciju tekućeg vremena u sekundama.

```
void main()
 { struct vreme
     { int sat;
        int minut;
        int sekund;
     } tekuce,naredno;
 printf("Unesi tekuce vreme [cc:mm:ss]:");
 scanf("%d:%d:%d",&tekuce.sat,&tekuce.minut,&tekuce.sekund);
 naredno=tekuce;
 if (++naredno.sekund==60)
      { naredno.sekund=0;
         if(++naredno.minut==60)
            { naredno.minut=0;
               if(++naredno.sat==24) naredno.sat=0;
            }
      }
 printf("Naredno vreme: %d:%d:%d\n", naredno.sat,
                 naredno.minut, naredno.sekund);
 }
```

## 6.7.2 Strukturni tipovi i pokazivači u C

Mogu se definisati pokazivačke promenljive koje će pokazivati na strukturne promenljive. Vrednosti ovakvih pokazivačkih promenljivih jesu adrese strukturnih promenljivih na koje pokazuju.

Na primer, za definisani strukturni tip podataka student može se definisati pokazivačka promenljiva *pstudent*:

```
struct student *pstudent;
```

Sada se elementima strukture student može pristupiti koristeći operator indirekcije '*' i operator '.':

```
(*pstudent).ime
(*pstudent).indeks
(*pstudent).godina
```

U cilju pristupa elementima strukturne promenljive pomoću pokazivača na tu promenljivu uveden je operator "strelica u desno" ->. Generalno pravilo je sledeće: ako je pokazivačkoj promenljivoj *pointer_strukture* dodeljena adresa neke strukture, tada je izraz

```
pointer_strukture ->ime_člana
```

ekvivalentan izrazu

```
(*pointer_strukture).ime_člana
```

U poslednjem izrazu zagrade nisu neophodne.

Operatori -> i '.' imaju asocijativnost sleva udesno.

Na primer elementima strukture pstudent pristupa se na sledeći način:

```
pstudent->ime
pstudent->indeks
pstudent->godina
```

Sve eksterne i statičke promenljive, uključujući strukturne promenljive, koje nisu eksplicitno inicijalizovane, automatski se inicijalizuju na vrednost nula.

**Primer.** Neka je definisana struktura *licnost* na sledeći način:

```
struct licnost
{  char ime[30];
   char adresa[50];
   unsigned starost;
}
```

Takođe, neka je definisana pokazivačka promenljiva osoba:

```
struct licnost *osoba;
```

Elementima strukture *osoba* može se pristupiti pomoću operatora indirekcije '*' i operatora '.':

```
(*osoba).ime
(*osoba).adresa
(*osoba).starost
```

Takođe, elementima strukturne promenljive na koju ukazuje pointer osoba, može se pristupiti i na sledeći način:

```
osoba->ime
osoba->adresa
osoba->starost
```

Formalni parametar funkcije može biti promenljiva nekog strukturnog tipa *S* kao i pokazivač na strukturu *S*. Ukoliko je formalni parametar strukturna promenljiva, tada se kao stvarni parametar navodi strukturna promenljiva ili neka konstantna vrednost strukturnog tipa *S*. U pozivu te funkcije, kao stvarni parametar, predaje se kopija vrednosti te strukturne promenljive. Na taj način, ne menja se vrednost strukturne promenljive koja je predata kao stvarni parametar. U slučaju kada je formalni parametar funkcije pointer na strukturu, tada se kao stvarni parametar predaje adresa strukturne promenljive koja se koristi kao stvarni parametar. Time je omogućeno ne samo korišćenje vrednosti pokazivačke promenljive (pomoću operatora indirekcije), već i promena tih vrednosti.

Takođe, strukture se mogu koristiti kao elementi nizova. Nizovi struktura se deklarišu analogno ostalim nizovima. Na primer, za definisane strukture *licnos*t i *student* mogu se koristiti nizovi

```
struct licnost osobe[20]
struct student studenti[50]
```

Ovakvim deklaracijama opisan je niz struktura osobe od najviše 20 elemenata, kao i niz struktura studenti sa najviše 50 elemenata. Svaki element niza osobe predstavlja jednu strukturnu promenljivu tipa *licnost*.

**Primer.** U ovom primeru je definisana matrica kompleksnih brojeva i dodeljene su početne vrednosti njenim elementima. Svaki element matrice predstavljen je parom realnih brojeva.

```
static struct complex
   { double real;
     double imag;
   } m[3][3]=
           { { {1.0,-0.5}, {2.5,1.0}, {0.7,0.7} },
             { {7.0,-6.5}, {-0.5,1.0}, {45.7,8.0} },
```

```
                    };
```

Elementi strukture mogu biti nove strukture. Struktura koja sadrži bar jedan element strukturnog tipa naziva se hijerarhijska struktura.

**Primer.** Definisana je struktura licnost koja sadrži element *datumrodjena* strukturnog tipa *datum*:

```
struct datum
      { unsigned dan;
        unsigned mesec;
        unsigned godina;
      }
 struct licnost
    { char *ime;
      datum datumrodjenja;
    }
```

Strukturna promenljiva *danas* može se definisati pomoću izraza
`struct datum danas;`
Pointer *p* na ovu strukturu deklariše se iskazom
`struct datum *p;`

Postavljanje pointera *p* na adresu strukturne promenljive *danas* ostvaruje se naredbom
`p=&danas;`
Sada se strukturnoj promenljivoj danas može pristupiti indirektno, pomoću pointera p na jedan od sledeća dva ekvivalentna načina:

```
(*p).godina=1996;  ⇔  p->godina=1996;
(*p).mesec=4;      ⇔   p->mesec=4;
(*p).dan=11;       ⇔  p->dan=11;
```

Takođe, pointeri mogu biti članovi strukture.

**Primer.** Izraz

```
struct pointeri
    {  int *n1;
       int *n2;
    };
```

definiše strukturu pointeri sa dva ukazatelja *n*1 i *n*2 na tip *int*. Sada se može deklarisati strukturna promenljiva *ukaz*:

```
struct pointeri ukaz;
void main()
 { int i1, i2;
    struct { int *n1;
             int *n2;
           }ukaz;
    ukaz.n1=&i1;  ukaz.n2=&i2;
    *ukaz.n1=111; *ukaz.n2=222;
    /* indirektno dodeljivanje vrednosti promenljivim i1, i2 */
    printf("i1=%d *ukaz.n1=%d\n", i1, *ukaz.n1);
    printf("i2=%d *ukaz.n2=%d\n", i2, *ukaz.n2);
 }
```

**Primer.** Dva ekvivalentna pristupa elementima strukture.

```
struct Simple { int a; };
int main() {
  Simple so, *sp = &so;
  sp->a;
  so.a;
}
```

## Unusual Properties

§ Structures have some very "un-C-like" properties, certainly when considering how arrays are handled

| | Arrays | Structures |
|---|---|---|
| Name is | pointer to zeroth element | the structure itself |
| Passed to functions by | pointer | value or pointer |
| Returned from functions | no way | by value or pointer |
| May be assigned with "=" | no way | yes |

## Returning Structure Instances

Structure instances may be returned by value from functions

This can be as inefficient as with pass by value

Sometimes it is convenient!

```
struct Complex  add(struct Complex a, struct Complex b)
{
   struct Complex result = a;

   result.real_part += b.real_part;
   result.imag_part += b.imag_part;

   return result;
}
```

```
struct Complex  c1 = { 1.0, 1.1 };
struct Complex  c2 = { 2.0, 2.1 };
struct Complex  c3;

c3 = add(c1, c2);      /* c3 = c1 + c2 */
```

**Primer.** Prebrojavanje studenata treće godine.

```
#include "c1.h"
#include<stdio.h>
#include<string.h>
void main()
 { int i;
   student s[10];
   int broji(student *);
   void citaj(student *,int);
   for(i=0; i<N; i++)
       citaj(&s[i],i);
  printf("Studenata trece godine ima %d\n", broji(s));
}

int broji(student *sts)
 { int i, cnt=0;
   for (i=0; i<N; ++i)
            cnt += sts[i].godina==3;
   return(cnt);
 }

void citaj(student *st, int i)
{  printf("%d ti student : ",i);
       gets(st->ime); scanf("%d", &(st->indeks));
       scanf("%d", &(st->godina));
       getchar();
}
```

## 6.7.3. Definicija strukturnih tipova pomoću typedef

Mogu se uvesti i strukturni tipovi podataka pomoću operatora *typedef*. Novi strukturni tipovi se definišu izrazom oblika

```
typedef struct
    {  <tip polja>  <ime polja>;
       ...
     } <ime>;
```

**Primer.** Definicija strukturnog tipa *datum*.

```
typedef struct
 {  int dan;
    int mesec;
    int godina;
 } datum;
```

Sada se identifikator *datum* može koristiti kao samostalan tip podataka. Na primer, iskazom

```
datum rodjendan[10];
```

deklariše se vektor *rodjendan* kao jednodimenzionalni vektor sa elementima tipa datum.

**Primer.** Ime string kao tip za pokazivače na tip *char* uvodi se pomoću operatora typedef na sledeći način:

```
typedef char *string;
```

Sada je

```
string str1, str2;
```

ekvivalentno sa

```
char *str1, *str2;
```

**Primer.** Implementacija osnovnih aritmetičkih operacija nad kompleksnim brojevima. Kompleksni brojevi su predstavljeni strukturom *kom*.

```
#include <stdio.h>
#include <math.h>
/* UVEDENI TIPOVI   */
struct kom
    { float prvi,drugi;    };

/* NAJAVE FUNKCIJA   */
void upis(struct kom *p);
void ispis(struct kom p);
struct kom zbir(struct kom p,struct kom q);
struct kom proiz(struct kom p,struct kom q);

void main()
 {
   struct kom a,b,c;
   upis(&a);upis(&b);
   c=zbir(a,b);
   printf("Njihov zbir je  ");ispis(c);
   c=proiz(a,b);
   printf("Njihov proizvod je  ");ispis(c);
 }

void upis(struct kom *p)
   /* Da bi upisani kompleksan broj bio zapamcen
      koriste se pointeri. Prema tome, koriste se oznake
      (*p).prvi=p->prvi, (*p).drugi=p->drugi   */
  {
    float x,y;
    scanf("%f%f",&x,&y);  p->prvi=x;p->drugi=y;
 }
```

```
void ispis(struct kom p)
{ if(p.drugi>0) printf("\n %f + i*%f\n",p.prvi,p.drugi);
  else printf("\n %f – i*%f\n",p.prvi,fabs(p.drugi));
}

struct kom zbir(struct kom p,struct kom q)
  /* U C-jeziku vrednost funkcije MOZE DA BUDE struct tipa */
 {  struct kom priv;
     priv.prvi=p.prvi+q.prvi;priv.drugi=p.drugi+q.drugi;
     return(priv);
 }

struct kom proiz(struct kom p,struct kom q)
 { struct kom priv;
    priv.prvi=p.prvi*q.prvi-p.drugi*q.drugi;
    priv.drugi=p.prvi*q.drugi+p.drugi*q.prvi;
    return(priv);
 }
```

Ovaj zadatak se može uraditi uz korišćenje *typedef* izraza. Kompleksni brojevi se reprezentuju novim tipom *kompl*.

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

typedef struct
 {   float prvi,drugi; } kompl;

void upis(kompl *p); void ispis(kompl p);
kompl zbir(kompl p,kompl q); kompl proiz(kompl p,kompl q);

void main()
 { kompl a,b,c;
    upis(&a); upis(&b); c=zbir(a,b); ispis(c);
    c=proiz(a,b); ispis(c);
 }

void upis(kompl *p)
 { float x,y;
    printf("  Daj dva broja  :\n");   scanf("%f%f",&x,&y);
    p->prvi=x;p->drugi=y;
 }

void ispis(kompl p)
 { printf("\n %f *i+ %f\n",p.prvi,p.drugi); }

kompl zbir(kompl p,kompl q)
 { kompl priv;
    priv.prvi=p.prvi+q.prvi;priv.drugi=p.drugi+q.drugi;
    return(priv);
 }

kompl proiz(kompl p,kompl q)
 { kompl priv;
    priv.prvi=p.prvi*q.prvi-p.drugi*q.drugi;
    priv.drugi=p.prvi*q.drugi+p.drugi*q.prvi;
    return(priv);
 }
```

**Primer.** Tačka u ravni je definisana kao strukturom `Tacka` koja sadrži dva broja tipa double. Krug je definisan strukturom koja sadrži centar tipa `Tacka` i poluprečnik tipa double. Napisati funkciju koja ispitatuje da li se dva kruga seku. U funkciji *main* je dato *n* krugova. Sa ulaza se učitava *n*, a u

narednih *n* redova su zadati centar i poluprečnik svakog kruga. Ispisati redne brojeve svaka dva kruga koji se seku.

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

typedef struct
{double x,y; } Tacka;

typedef struct
{ Tacka O; double R; } Krug;

int intersect(Krug a, Krug b)
{ double d;
  d=sqrt(pow(a.O.x-b.O.x,2)+ pow(a.O.y-b.O.y,2));
  return (d<a.R+b.R)&& (d>fabs(a.R-b.R));
}

void main()
{ int i,j,n;
  Krug a[100];
  scanf("%d",&n);
  for(i=1;i<=n;i++)
    scanf("%lf%lf%lf",&a[i].O.x,
              &a[i].O.y,&a[i].R);
  for(i=0;i<n-1;i++)
    for(j=i+1;j<n;j++)
        if(intersect(a[i],a[j]))
          printf("%d  %d\n",i,j);
}
```

```
#include<stdio.h>
#include<math.h>
struct Tacka{ double x,y;  };
struct Krug{  Tacka O; double R; };

void CitajKrug(Krug *K)
{ scanf("%lf%lf%lf", &K->O.x,
           &K->O.y,&K->R);
}

int Intersect(Krug K1, Krug K2)
{ double
    d=sqrt(pow(K1.O.x-K2.O.x,2)+
           pow(K1.O.y-K2.O.y,2));
  return ((d<K1.R+K2.R) &&
          (d>fabs(K1.R-K2.R)));
}

void main()
{ int i,j,n;
  Krug K[5];
  scanf("%d",&n);
  for(i=0; i<n; i++)  CitajKrug(&K[i]);
  for(i=0; i<n-1; i++)
    for(j=i+1; j<n; j++)
      if(Intersect(K[i],K[j])) printf("(%d,%d)\n",i+1,j+1);
}
```

**Primer.** Retka matrica se može predstaviti brojem ne-nula elemenata kao i sledećim informacijama za svaki ne-nula element:

- redni broj vrste tog elementa,
- redni broj kolone tog elementa, i
- realan broj koji predstavlja vrednost tog elementa.

Predstaviti retku matricu brojem ne-nula elemenata i nizom slogova, u kome svaki slog sadrži redni broj vrste i kolone kao i vrednost svakog ne-nula elementa.

Napisati procedure za formiranje niza slogova koji predstavljaju reprezentaciju retke matrice $A$.

Napisati proceduru za ispis matrice.

Napisati proceduru za formiranje sparse reprezentaciju predstavlja matricu $A^T$.

Napisati funkciju koja za zadate vrednosti $i$, $j$ preko tastature izračunava vrednost elementa $A[i,j]$.

```
#include<stdio.h>

 struct slog
   { int i,j;
     float aij;
   };

 void pravimatricu(struct slog x[10], int *n)
   { int p,q,i;
     float xpq;
      printf("Broj ne-nula elemenata? "); scanf("%d", n);
      printf("Pozicije i vrednosti ne-nula elemenata?\n");
      for(i=1; i<=*n; i++)
        { scanf("%d%d%f", &p,&q,&xpq);
          x[i].i=p; x[i].j=q; x[i].aij=xpq;
        }
   }

 void transponuj(struct slog x[10], int n, struct slog y[10])
   { int i;
      for(i=1; i<=n; i++)
        { y[i].i=x[i].j; y[i].j=x[i].i; y[i].aij=x[i].aij;  }
   }

 void pisimatricu(struct slog y[10], int n)
   { int i;
      printf("%d\n",n);
      for(i=1; i<=n; i++)
           printf("(%d %d): %10.3f\n",y[i].i, y[i].j,y[i].aij);
   }

 float vrednost(int n, struct slog x[10], int k, int l)
   { int i;
      for(i=1; i<=n; i++)
      if((x[i].i==k) && (x[i].j==l))
         return(x[i].aij);
      return(0);
   }

 void main()
  { int i,j,n;
    struct slog p[10], q[10];
     pravimatricu(p, &n);
     printf("Matrica je: \n");
     pisimatricu(p,n);
     transponuj(p,n,q);
     printf("Transponovana matrica: \n");
```

```
    pisimatricu(q,n);
    printf("i,j= ? "); scanf("%d%d", &i,&j);
    printf("%10.3f\n",vrednost(n,p,i,j));
}
```

**Primer.** Modifikacija prethodnog primera. Matrica se zadaje brojem vrsta, brojem kolona, brojem ne-nula elemenata kao i nizom slogova koji odgovaraju ne-nula elementima. Svaki takav slog je određen sledećim informacijama za ne-nula element:

- redni broj vrste tog elementa,
- redni broj kolone tog elementa, i
- realan broj koji predstavlja vrednost tog elementa.

Napisati funkciju za formiranje retke matrice *A*.

Napisati funkciju za ispis nenula elemenata matrice.

Napisati funkciju za uobičajeni ispis matrice.

Napisati funkciju za formiranje sparse reprezentaciju matrice $A^T$.

Napisati funkciju koja za zadate vrednosti *i, j* preko tastature izračunava vrednost elementa *A*[*i,j*].

```
#include<stdio.h>

struct slog
  { int i,j;
    float aij;
  };

struct matrica
{ int m,n,bn;
  slog sp[10];
};

void pravimatricu(matrica *a)
  {  int i;
     printf("Dimenzije matrice: "); scanf("%d%d",&a->m,&a->n);
     printf("Broj ne-nula elemenata? "); scanf("%d", &a->bn);
     printf("Pozicije i vrednosti ne-nula elemenata?\n");
     for(i=0; i<a->bn; i++)
          scanf("%d%d%f", &a->sp[i].i,&a->sp[i].j,&a->sp[i].aij);
     }

void transponuj(matrica x, matrica *y)
  { int i;
     y->m=x.n; y->n=x.n;
       y->bn=x.bn;
     for(i=0; i<x.bn; i++)
       { y->sp[i].i=x.sp[i].j; y->sp[i].j=x.sp[i].i;
         y->sp[i].aij=x.sp[i].aij; }
  }

void pisimatricu(matrica x)
  { int i;
     for(i=0; i<x.bn; i++)
          printf("(%d %d): %10.3f\n",x.sp[i].i, x.sp[i].j,x.sp[i].aij);
  }

float vrednost(matrica x, int k, int l)
  { int i;
    for(i=0; i<x.bn; i++)
      if((x.sp[i].i==k) && (x.sp[i].j==l))
           return(x.sp[i].aij);
    return(0);
  }
```

```
 void pisigusto(matrica x)
 { int i,j;
   for(i=0;i<x.m; i++)
   { for(j=0; j<x.n; j++)
          printf("%5.2f  ",vrednost(x,i,j));
     printf("\n");
   }
 }

 void main()
  { int i,j;
    matrica p, q;
     pravimatricu(&p);
     printf("Matrica je: \n");    pisimatricu(p);  pisigusto(p);
     transponuj(p,&q);
     printf("Transponovana matrica: \n");    pisimatricu(q); pisigusto(q);
     printf("i,j= ? "); scanf("%d%d", &i,&j);
     printf("%10.3f\n",vrednost(p,i,j));
 }
```

**Primer.** Definisana je struktura koja predstavlja datum:

```
type datum=record
            dan:1..31;
            mesec:1..12;
            godina:0..maxint;
          end;
```

Zatim je definisana je struktura koja sadrži jedno polje tipa *datum* i jedno polje tipa sting:

```
     praznik=record
               d:datum;
               naziv:string[30];
             end;
```

Ova struktura reprezentuje jedan praznik. U programu formirati niz datuma koji predstavljaju praznike.

```
program Praznici;

type datum=record
            dan:1..31;
            mesec:1..12;
            godina:0..maxint;
          end;
     praznik=record
               d:datum;
               naziv:string[30];
             end;
procedure CitajPraznik(var p:praznik);
 begin
   with p do
     begin
      with d do
        begin
          write('Dan = ? '); readln(dan);
          write('Mesec = ? '); readln(mesec);
          write('Godina = ? '); readln(godina);
        end;
        write('Naziv praznika ? '); readln(naziv);
     end; {with }
 end;

procedure PisiPraznik(p:praznik);
```

```
  begin
    writeln(p.naziv);   writeln;
    writeln(p.d.dan,'.',p.d.mesec,'.',p.d.godina,'.');   writeln;
  end;

var
  i,n:integer;
  a:array[1..30] of praznik;

begin
  readln(n);  writeln('Zadati podatke za ',n,' praznika:');
  for i:= 1 to n do CitajPraznik(a[i]);
  writeln; writeln;
  for i:=1 to n do
    begin
      writeln(i,'. praznik:'); PisiPraznik(a[i]);
    end;
end.
```

**Rešenje u jeziku C:**

```
#include<stdio.h>
#include<string.h>
typedef struct
{ int dan, mesec, godina;  }datum;
typedef struct
{ char naziv[30];
  datum d;
}praznik;

void CitajPraznik(praznik *p)
{ int d,m,g;
  printf("Dan = ? "); scanf("%d", &d);
  printf("Mesec = ? "); scanf("%d", &m);
  printf("Godina = ? "); scanf("%d", &g);
  p->d.dan=d;  p->d.mesec=m;  p->d.godina=g;
  printf("Naziv praznika ? ");
  gets(p->naziv); gets(p->naziv);
}
void PisiPraznik(praznik p)
{ puts(p.naziv);
  printf("\n%d.%d.%d.\n\n",p.d.dan,p.d.mesec,p.d.godina);
}
void main()
{ int i,n;
  praznik a[30];
  scanf("%d",&n);  printf("Zadati podatke za %d praznika:\n",n);
  for(i=0;i<n;i++)CitajPraznik(a+i);
  printf("\n\n");
  for(i=0;i<n;i++)
  { printf("%d. praznik:\n",i+1); PisiPraznik(a[i]); }
}
```

**Primer.** Polinom se može predstaviti strukturom koja sadrži red polinoma i niz njegovih koeficijenata. Napisati funkcije za izračunavanje zbira, razlike, proizvoda i količnika dva polinoma koji su zadati odgovarajućim strukturama. Napisati program za testiranje napisanih funkcija.

```
#include<stdio.h>

typedef struct
{ double a[30]; int n; } Poli;

Poli zbir(Poli p1, Poli p2)
{ Poli p;  int i;
```

```
  p.n=(p1.n>p2.n)? p1.n : p2.n;
  for(i=0;i<=p.n; i++)
        if(i>p2.n)p.a[i]=p1.a[i];
        else if(i>p1.n)p.a[i]=p2.a[i];
        else p.a[i]=p1.a[i]+p2.a[i];
  while(p.n>=0 && p.a[p.n]==0)p.n--;
  return p;
}

Poli razlika(Poli p1, Poli p2)
{ Poli p;  int i;
  p.n=(p1.n>p2.n)? p1.n : p2.n;
  for(i=0;i<=p.n; i++)
      if(i>p2.n)p.a[i]=p1.a[i];
      else if(i>p1.n)p.a[i]=-p2.a[i];
      else p.a[i]=p1.a[i]-p2.a[i];
  while(p.n>=0 && p.a[p.n]==0)p.n--;
  return p;
}

Poli proizvod(Poli p1, Poli p2)
{ Poli p;  int i,j;
  p.n=p1.n+p2.n;
  for(i=0;i<=p.n; p.a[i++]=0);
  for(i=0; i<=p1.n; i++)
        for(j=0; j<=p2.n; j++) p.a[i+j]+=p1.a[i]*p2.a[j];
  return p;
}

Poli kolicnik(Poli p1, Poli p2, Poli *ostatak)
{ Poli p; int i,j;
  p.n=p1.n-p2.n;
  for(i=p.n; i>=0; i--)
  { p.a[i]=p1.a[p2.n+i]/p2.a[p2.n];
    for(j=0; j<=p2.n; j++) p1.a[i+j]-= p.a[i]*p2.a[j];
  }
  while(p1.n>=0 && p1.a[p1.n]==0)p1.n--;
  *ostatak=p1;
  return p;
}

Poli citaj()
{ Poli p; int i;
  printf("Stepen = ? "); scanf("%d", &p.n);
  printf("Koeficijenti?\n"); for(i=p.n; i>=0; scanf("%lf", &p.a[i--]));
  return p;
}

void pisi(Poli p)
{ int i;
  putchar('{');
  for(i=p.n; i>=0; i--){ printf("%.2lf", p.a[i]); if(i>0)putchar(',');  }
  putchar('}');
}

void main()
{ Poli p1, p2, p3;
  while(1)
  {   p1=citaj(); p2=citaj();
      printf("P1 = "); pisi(p1); putchar('\n');
      printf("P2 = "); pisi(p2); putchar('\n');
      printf("P1+P2 = "); p3=zbir(p1,p2); pisi(p3); putchar('\n');
      printf("P1-P2 = "); pisi(razlika(p1,p2)); putchar('\n');
```

```
      printf("P1*P2 = "); pisi(proizvod(p1,p2)); putchar('\n');
      printf("P1/P2 = "); pisi(kolicnik(p1,p2, &p3));  putchar('\n');
      printf("P1%%P2 = "); pisi(p3); putchar('\n'); putchar('\n');
  }
}
```

## 6.7.4. Unije

Unija je tip podataka koji može da sadrži (u raznim situacijama) objekte različitih tipova. Unije određuju način manipulisanja različitim tipovima podataka u istoj memorijskoj oblasti. Svrha njihovog postojanja je ušteda memorije. One su analogne slogovima promenljivog tipa u Pascal-u, a sintaksa im je zasnovana na strukturama jezika C. Svrha unije je da postoji samo jedna promenljiva koja može da sadrži bilo koju od vrednosti različitih tipova. Unije se razlikuju od struktura po tome što elementi unije koriste isti memorijski prostor, i što se u svakom trenutku koristi samo jedan element unije.

**Primer.** Pretpostavimo da u tabeli simbola nekog kompajlera, konstanta može da bude int, float ili char. Najveća ušteda memorije se postiže ako je vrednost ma kog elementa tabele memorisana na istom mestu, bez obzira na tip. U našem slučaju je potrebna sledeća deklaracija

```
union tag
  {  int ival;
     float fval;
     char *sval;
  };
  union tag u;
```

Ekvivalentna deklaracija je data kako sledi:

```
union tag
 { int ival;
   float fval;
   char *sval;
 } u;
```

Promenljivoj koja je deklarisana kao unija prevodilac dodeljuje memorijski prostor dovoljan za memorisanje onog člana unije koji zauzima najveći memorijski prostor. Programer mora da vodi računa o tome koji tip se trenutno memoriše u uniji, jer je važeći tip onaj koji je najskorije memorisan.

Članovi unije se selektuju identično članovima strukture:

```
unija.clan
```

ili

```
pointer_unije->clan.
```

Ako je int tip tekućeg člana unije u, njegova vrednost se prikazuje izrazom

```
printf("%d\n",u.ival);
```

ako je aktivni član tipa float pišemo

```
printf("%f\n",u.fval);
```

a ako je tipa char

```
printf("%s\n",u.sval);
```

Elementima strukturne promenljive u vrednosti se mogu dodeliti, na primer, iskazima
u.ival=189; ili u.fval=0.3756; ili u.sval="string";

**Primer.** Jednostavna unija koja sadrži jedno celobrojno i jedno realno polje.

```
void main()
 { union { int i;
           float f;
         } x;
```

```
   x.i=123;   printf("x.i=%d x.f=%f\n",x.i,x.f);
   x.f=12.803;   printf("x.i=%d x.f=%f\n",x.i,x.f);
}
```

Unije mogu da se pojavljuju u strukturama i nizovima. Kombinacijom strukture i unije grade se promenljive strukture.

**Primer.** Tablica simbola

```
struct { char*name;
         int flags;
         int utype;
         union { int ival;
                 float fval;
                 char *sval;
               } u;
       } symtab[100];
```

Sada možemo pristupiti članu *ival i*-tog elementa tablice simbola pomoću izraza

```
symtab[i].u.ival;
```

Prvi znak stringa sval *i*-tog elementa tablice simbola se dobija jednim od sledeća dva ekvivalentna izraza:

```
*symtab[i].u.sval;
symtab[i].u.sval[0];
```

Unija se može inicijalizovati samo pomoću vrednosti koja odgovara tipu njenog prvog člana. Stoga se unija opisana u prethodnom primeru može inicijalizovati samo pomoću celobrojne vrednosti.

**Primer.** Osnovne geometrijske figure se mogu okarakterisati sledećom strukturom:

```
struct figura
{ float povrsina, obim;  /* Zajednicki elementi */
  int tip;               /* Oznaka aktivnog elementa */
  union
    { float r;           /* Poluprecnik kruga */
      float a[2];        /* Duzine strana pravougaonika */
      float b[3];        /* Duzine strana trougla */
   } geom_fig;
} fig;
```

**Primer.** Napisati program za izračunavanje rastojanja između dve tačke u ravni, pri čemu postoji mogućnost da svaka od tačaka bude definisana u Dekartovom ili polarnom koordinatnom sistemu. Obuhvaćeni su svi slučajevi: kada su obe tačke zadate Dekartovim koordinatama, jedna od tačaka Dekartovim, a druga polarnim koordinatama i slučaj kada su obe tačke definisane polarnim koordinatama.

```
#include<stdio.h>
#include<math.h>
typedef enum { Dekartove, Polarne }TipKor;

typedef struct
{ double x,y; } Dek;

typedef struct
{ double r,fi; } Pol;

typedef struct
{ TipKor tip;
   union
   {  Dek DK;
      Pol PK;
   };
```

```
} koordinate;

void ucitaj(koordinate *Q)
{ int ch;
  double u,v;
  printf("Tip koordinata? 1 za Dekartove, 2 za polarne: ");
  scanf("%d", &ch);
  printf("Koordinate? ");
  switch (ch)
  { case 1: Q->tip=Dekartove; scanf("%lf%lf", &u, &v);
            Q->DK.x=u; Q->DK.y=v;
            break;
    case 2:  Q->tip=Dekartove; scanf("%lf%ld", &u,&v);
            Q->PK.r=u, Q->PK.fi=v;
            break;
  }
}

void main()
{ koordinate A,B;
  double d;
  ucitaj(&A); ucitaj(&B);
  switch(A.tip)
  {
     case Dekartove:
           switch(B.tip)
           {  case Dekartove:
                  d=sqrt(pow(A.DK.x-B.DK.x,2)+pow(A.DK.y-B.DK.y,2));
                  break;
              case Polarne:
                    d=sqrt(pow(A.DK.x-B.PK.r*cos(B.PK.fi),2)+
                            pow(A.DK.y-B.PK.r*sin(B.PK.fi),2));
                     break;
           }
           break;
     case Polarne:
           switch(B.tip)
           {case Dekartove:
                     d=sqrt(pow(A.PK.r*cos(A.PK.fi)-B.DK.x,2)+
                            pow(A.PK.r*sin(A.PK.fi)-B.DK.y,2));
                     break;
            case Polarne:
                d=sqrt(pow(A.PK.r*cos(A.PK.fi)-B.PK.r*cos(B.PK.fi),2)+
                       pow(A.PK.r*sin(A.PK.fi)-B.PK.r*sin(B.PK.fi),2));
                    break;
           }
           break;
  }
  printf("%.4lf\n",d);
}
```

**Primer.** Napisati program za sumiranje površina *n* geometrijskih figura. Figura može biti krug (određen dužinom poluprečnika), pravougaonik (određen dužinama susednih stranica) ili trougao (određen dužinama svojih stranica).

```
#include<stdio.h>
#include<math.h>
void main()
{ struct pravougaonik
      { double a,b;};
  struct trougao
      { double a,b,c; };
```

```
  struct figura
      { int tip;
      union
        { double r;
          struct pravougaonik p;
          struct trougao t;
        };
      } figure[50];

  int i,n;
  double pov=0,s;
  scanf("%d",&n);
  for(i=0; i<n; i++)
  { printf("Tip figure (0 - krug, 1-pravougaonik, 2-trougao): ");
    scanf("%d",&figure[i].tip);
      switch(figure[i].tip)
      { case 0:
          scanf("%lf",&figure[i].r);break;
        case 1:
            scanf("%lf%lf",&figure[i].p.a,&figure[i].p.b);break;
        case 2:

      scanf("%lf%lf%lf",&figure[i].t.a,&figure[i].t.b,&figure[i].t.c);
break;
      }
  }
  for(i=0;i<n;i++)
  { switch(figure[i].tip)
      { case 0:
          pov+=3.14*figure[i].r*figure[i].r;break;
        case 1:
            pov+=figure[i].p.a*figure[i].p.b;break;
        case 2:
            s=(figure[i].t.a+figure[i].t.b+figure[i].t.c)/2;
            pov+=sqrt(s*(s-figure[i].t.a)*(s-figure[i].t.b)*(s-
figure[i].t.c));
      }
  }
  printf("Povrsina svih figura je %lf\n",pov);
}
```

# 6.8. Nabrojivi tip podataka u c

Pomoću ključne reči `enum` deklarišu se nabrojivi (enumerisani) tipovi podataka. Promenljive nabrojivog tipa mogu da imaju samo konačan skup vrednosti. Deklaracija promenljivih nabrojivog tipa se sastoji od ključne reči `enum`, imena enumerisanog tipa, eventualno liste dozvoljenih vrednosti i liste enumerisanih promenljivih koje se deklarišu.

**Primer.** Iskazom

```
enum flag{true,false};
```

definisan je nabrojivi tip `enum flag`. Promenljive ovog tipa mogu da imaju samo vrednosti *true* ili *false*. Ključna reč `enum` označava da se radi o nabrojivom tipu podataka, a identifikator *flag* predstavlja ime definisanog nabrojivog tipa. Između zagrada se nalazi lista vrednosti.

Sada se mogu deklarisati promenljive nabrojivog tipa *enum flag*:

```
enum flag file_end, input_end;
```

Ime nabrojivog tipa se može izostaviti, čime se dobija neimenovani nabrojivni tip. Na primer, sledeća deklaracija je analogna prethodnoj:

```
enum {true,false} file_end, input_end;
```

Dodeljivanje vrednosti enumerisanoj promenljivoj je analogno dodeljivanju vrednosti promenljivima ostalih tipova. Sledeći iskazi su legalni:

```
file_end=false;
if(input_end == true)
```

U sledećem primeru se definiše nabrojivi tip *enum dani*:

```
enum dani {pon,uto,sre,cet,pet,sub,ned};
```

Vrednosti nabrojivog tipa se tretiraju kao celobrojne konstante. Preciznije, C kompajler dodeljuje sekvencijalne celobrojne vrednosti elementima liste vrednosti, startujući od prvog elementa liste, kome se dodeljuje vrednost 0. Na primer, izrazom

```
ovaj_dan=sreda;
```

dodeljena je vrednost 2 (ne ime *sreda*) promenljivoj *ovaj_dan* (tipa *enum dani*).

Sekvencijalni način dodeljivanja celobrojnih vrednosti elementima iz liste imena može se promeniti eksplicitnim dodeljivanjem željene celobrojne vrednosti nekom elementu iz liste imena. Pri tome se mogu koristiti i negativni celi brojevi.

**Primer.** Možemo pisati

```
enum dani{pon,uto,sre=10,cet,pet,sub=100,ned} ;
```

Sada je

```
pon=0, uto=1, sre=10, cet=11, pet=12, sub=100, ned=101.
```

Ako se želi eksplicitno dodeljivanje celobrojne vrednosti nabrojivog promenljivoj, mora da se koristi kast operator.

Na primer, izraz

```
ovaj_dan=pon
```

ekvivalentan je sa

```
ovaj dan=(enum dani)0;
```

**Primer.** Program određivanje sledećeg dana u nedelji

```
#include<stdio.h>
void main()
 { int k;
    enum dani{pon,uto,sre,cet,pet,sub,ned} sledeci;
    printf("unesite danasnji dan:\n");  scanf("%d",&k);
    k+=1; k%=7;
    sledeci=(enum dani)k;
    printf("\nSledeci dan je %d.dan u nedelji\n", (int)sledeci);
 }
```

Ista konstanta se ne može koristiti u deklaracijama različitih nabrojivih tipova podataka. Na primer, definicije tipova

```
enum radnidani {pon,uto,sre,cet,pet};
enum vikend {pet,sub,ned};
```

su nekorektne, jer se konstanta *pet* pojavljuje u definiciji oba tipa. Nabrojivi tip se može definisati koristeći operatore `typedef` i `enum`. Na primer, sledećim definicijama su uvedena dva nabrojiva tipa, samoglasnik i dani:

```
typedef enum {A,E,I,O,U} samoglasnik;
typedef enum {pon,uto,sre,cet,pet,sub,ned} dani;
```

Posle definicije ovih tipova mogu se deklarisati promenljive uvedenih tipova:

```
samoglasnik slovo;
dani d1,d2;
```

Promenljivoj nabrojivog tipa se može dodeliti bilo koja vrednost koja je sadržana u definiciji nabrojivog tipa:

```
slovo=U;
d1=sre;
```

Ako se vrednosti promenljivih nabrojivog tipa ispisuju pomoću funkcije printf(), ispisuju se celobrojne vrednosti koje su dobile na osnovu definicije nabrojivog tipa.

Za razliku od Pascala, promenljive nabrojivog tipa se ne mogu koristiti kao brojačke promenljive u operatorima *FOR* ciklusa.

## 6.9. Dinamičke matrice i strukture

**Primer.** Dinamička matrica se predstavlja strukturom koja sadrži dimenzije i pokazivač na elemente matrice. Sastaviti na jeziku *C* paket funkcija za rad sa dinamičkim matricama realnih brojeva koji sadrži funkcije za:

- dodeljivanje memorije matrici datih dimenzija,
- oslobađanje memorije koju zauzima matrica,
- kopiranje matrice,
- čitanje matrice preko glavnog ulaza,
- ispisivanje matrice preko glavnog izlaza,
- nalaženje transponovane matrice za datu matricu,
- nalaženje zbira, razlike i proizvoda dve matrice.

Sastaviti na jeziku *C* glavni program za proveru ispravnosti rada gornjih funkcija.

Rešenje:

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

typedef enum {MEM, DIM} Greska;  /* Šifre grešaka. */
typedef struct { float **a; int m, n; } Din_mat; /* Struktura matrice. */

Din_mat stvori (int m, int n);     /* Dodela memorije. */
void unisti (Din_mat dm);          /* Oslobadjanje memorije. */
Din_mat kopiraj (Din_mat dm) ;     /* Kopiranje matrice. */
Din_mat citaj (int m, int n) ;     /* Citanje matrice. */
void pisi (Din_mat dm, const char *frm, int max); /* Ispisivanje matrice.*/
Din_mat transpon (Din_mat dm);               /* Transponovana matrica*/
Din_mat zbir (Din_mat dm1, Din_mat dm2);     /* Zbir matrica. */
Din_mat razlika (Din_mat dm1, Din_mat dm2);  /* Razlika matrica. */
Din_mat proizvod (Din_mat dm1, Din_mat dm2); /* Proizvod matrica. */

/* Definicije paketa za rad sa dinamickim matricama. */

const char *poruke[] = /* Poruke o greškama.*/
   { "Neuspela dodela memorije", "Neusaglasene dimenzije matrica" };

static void greska (Greska g) { /* Ispisivanje poruke grešci. */
   printf("\n*** %s ! ***\n\a", poruke[g]);
   exit(g+1);
}

Din_mat stvori (int m, int n) { /* Dodela memorije. */
  int i; Din_mat dm;
  dm.m = m; dm.n = n;
  if((dm.a = (float **)malloc(m*sizeof(float*))) == NULL) greska (MEM);
  for(i=0; i<m; i++)
     if((dm.a[i] = (float *)malloc(n*sizeof(float))) == NULL) greska (MEM);
```

```
  return dm;
}

void unisti (Din_mat dm) {  /* Oslobadjanje memorije. */
  int i;
  for (i=0; i<dm.m; free(dm.a[i++]));
  free (dm.a);
}

Din_mat kopiraj (Din_mat dm) {   /* Kopiranje matrice. */
  int i, j;
  Din_mat dm2 = stvori (dm.m, dm.n);
  for (i=0; i<dm.m; i++)
      for (j=0; j<dm.n;j++)
        dm2.a[i][j] = dm.a[i][j];
  return dm2;
}

Din_mat citaj(int m, int n){ /* Citanje matrice. */
  int  i,j;
  Din_mat dm = stvori(m,n);
  for(i=0; i<m; i++)
    for(j = 0; j<n; scanf("%f",&dm.a[i][j++]));
  return dm;
}

void pisi(Din_mat dm, const char *frm, int max){ /* Ispis matrice.  */
  int i,j;
  for(i=0; i<dm.m; i++) {
     for(j=0; j<dm.n; j++) {
        printf(frm,dm.a[i][j]);
        putchar ((j%max==max-1 || j==dm.n-1) ? '\n' : ' ');
     }
     if(dm.n > max)putchar ('\n');
  }
}

Din_mat transpon(Din_mat dm)  { /*  Transponovana matrica*/
  int i, j;
  Din_mat dm2 = stvori(dm.n, dm.m);
  for (i=0; i<dm.m; i++)
     for(j=0; j<dm.n; j++) dm2.a[j][i] = dm.a[i][j];
  return dm2;
}

Din_mat zbir(Din_mat  dm1, Din_mat dm2)  { /* Zbir matrica. */
  int i,j;
  Din_mat dm3;
  if(dm1.m!=dm2.m || dm1.n != dm2.n) greska(DIM);
  dm3 = stvori(dm1.m, dm1.n);
  for(i=0; i<dm3.m;i++)
      for(j=0; j<dm3.n;j++)
        dm3.a[i][j]= dm1.a[i][j] + dm2.a[i][j];
  return dm3;
}

Din_mat razlika (Din_mat dm1, Din_mat dm2) { /* Razlika matrica.*/
  int i,j; Din_mat dm3;
  if(dm1.m!=dm2.m || dm1.n != dm2.n) greska (DIM);
  dm3 = stvori (dm1.m, dm1.n);
  for(i=0; i<dm3.m;i++)
      for(j=0; j<dm3.n;j++)
         dm3.a[i][j] = dm1.a[i][j] - dm2.a[i][j];
```

```
  return dm3;
}

Din_mat proizvod (Din_mat dm1, Din_mat dm2) { /*  Proizvod matrica. */
  int i, j, k; Din_mat dm3;
  if (dm1.n!=dm2.m) greska (DIM);
  dm3 = stvori(dm1.m, dm2.n);
  for(i=0; i<dm3.m; i++)
    for(k=0; k<dm3.n; k++)
      for(dm3.a[i][k]=j=0; j<dm2.m; j++)
         dm3.a[i][k] +=dm1.a[i][j] * dm2.a[j][k];
  return dm3;
}

int main() {
  while (1) {
    Din_mat dm1, dm2, dm3; int m, n;
    printf ("m1, n1? "); scanf ("%d%d", &m, &n);
    if (m<=0 || n<=0) break;
    printf("Matrl?\n"); dm1 = citaj(m,n);
    printf ("m2, n2? ");  scanf ("%d%d", &m, &n);
    if(m<=0 || n<=0) break;
    printf ("Matr2?\n"); dm2 = citaj(m,n);
    if(dm1.m==dm2.m && dm1.n==dm2.n) {
      dm3 = zbir(dm1, dm2); printf ("ZBIR:\n");
      pisi(dm3,"%8.2f", 8); unisti(dm3);
      dm3 = razlika(dm1, dm2); printf ("RAZLIKA:\n");
      pisi(dm3, "%8.2f", 8); unisti (dm3);
    }
    if(dm1.n == dm2.m) {
      dm3 = proizvod(dm1,dm2); printf("PROIZVOD:\n");
      pisi(dm3, "%8.2f", 8); unisti (dm3);
    }
    putchar('\n');
    unisti(dm1); unisti(dm2);
  }
  return 0;
}
```

# 6.10. Datoteke

Programi koje smo do sada koristili imali su jedan ozbiljan nedostatak. Naime, rezultati obrade ulaznih veličina prestankom izvršavanja programa su nepovratno nestajali. Jedan od načina da se sačuvaju podaci bio bi da se posredstvom štampača ispišu na hartiji. Međutim, ako postoji potreba da podaci budu pristupačni nekom programu, treba ih sačuvati na magnetnim nosiocima informacija, na primer na disku ili traci. Na taj način, na disku ili traci se mogu čuvati datoteke sa podacima koje možemo obrađivati koliko god hoćemo puta, a da ih ne moramo učitavati sa tastature.

Datoteke su sekvencijalne strukture jednorodnih zapisa čiji broj nije unapred poznat, i koje se zapisuju na nekom od spoljašnjih memorijskih medijuma. Ovakvi strukturni tipovi opisuju se sa file i postali su standardni u programskim jezicima posle pojave ovog koncepta u jeziku Pascal.

Datoteke su jednorodne strukture podataka, slično poljima ali za razliku od polja, broj elemenata datoteke se ne definiše unapred i podrazumeva se da one fizički postoje van programa na nekom od spoljašnjih memorijskih medijuma (disku, traci, disketi itd.).

## 6.10.1. Pristup datotekama u C

Datoteke omogućavaju da se ulazne veličine i rezultati obrade podataka trajno sačuvaju na disku. Datoteka se tretira kao sekvencijalni niz karaktera. Datotekama se pristupa korišćenjem ukazatelja na strukturu *FILE,* koja je definisana u biblioteci *stdio.h*. U datoteci *stdio.h* su definisane i konstante *EOF*

i *NULL*. Konstanta *EOF* označava kraj datoteke i ima vrednost -1. Konstanta *NULL* ima vrednost 0 i vraća se kao rezultat neuspešnog poziva nekih funkcija za upravljanje datotekama.

## Otvaranje i zatvaranje datoteka

Programski jezik C koristi dva osnovna tipa fajlova: **tekstualni** i **binarni** tip.

Rad sa fajlovima odvija se kroz nekoliko osnovnih aktivnosti: otvaranje fajla, čitanje i upis u fajl, kretanje po fajlu, zatvaranje fajla.

Datoteka mora da bude otvorena pre bilo kakvog procesiranja. Prilikom otvaranja datoteke mora se specificirati njeno ime i tip željene ulazno-izlazne operacije. Može se koristiti režim čitanja (*read* mode), režim upisa (*write* mode), odnosno režim dodavanja (*append* mode). Za otvaranje datoteka koristi se funkcija *fopen* iz C biblioteke. Zaglavlje ove funkcije je oblika

```
FILE *fopen(const char *filename, const char *mode);
```

Funkcija *fopen*() vraća pointer na otvoreni fajl. Vraćena vrednost NULL indicira grešku prilikom otvaranja fajla. Parametar *filename* ime fajla, a parametar *mode* predstavlja način na koji se fajl otvara. Mogući načini su opisani u nastavku:

"r" Otvara za čitanje. Ako fajl ne postoji ili ne može biti pronađen poziv ne uspeva.

"w" Otvara prazan fajl za pisanje. Ako dati fajl postoji njegov sadržaj se briše.

"a" Otvara fajl za pisanje na kraju fajla (appending) bez brisanja EOF markera pre pisanja novih podataka na kraju postojećeg fajla. Ukoliko fajl ne postoji, kreira se novi.

"r+" Otavara fajl za čitanje i pisanje. Fajl mora da postoji.

"w+" Otvara prazan fajl za čitanje i pisanje. Ako dati fajl postoji, njegov sadržaj je obrisan.

"a+" Otvara fajl za čitanje i appednovanje. Operacija apendovanja uključuje brisanje EOF markera pre nego se novi podaci upišu u fajl. Marker EOF se restaurira posle kompletiranja pisanja. Ukoliko fajl sa specificiranim imenom ne postoji, on se prvo kreira.

Kad je fajl otvoren sa pristupom "a" ili "a+" sve opracije upisa se dešavaju na kraju fajla. Fajl pointer može biti repozicioniran korišćenjem *fseek* ili *rewind*, ali se uvek vraća na kraj fajla pre bilo koje operacije upisa u fajl. Na taj način se obezbeđuje da preko postojećih podataka nema upisa.

Mod "a" ne briše EOF marker pre dodavanja podataka u fajl. Ako se to desi MSDOS komanda TYPE samo pokazuje podatke do originalnog EOF markera i ne pokazuje podatke dodate fajlu. Mod "a+" briše EOF marker pre dodavanja fajlu. Posle dodavanja MSDOS komanda TYPE pokazuje sve podatke dodate fajlu. Mod "a+" se zahteva za dodavanje stream fajlu koji je terminiran CTRL+Z EOF markerom. Kad je specificiran neki od "r+", "w+" ili "a+" i čitanje i pisanje su dozvoljeni, a kaže se da je fajl otovren za update. Potrebno je da se prilikom prebacivanja između čitanja i pisanja uradi intervencija u obliku poziva *fflush*, *fsetpos*, *fseek*, ili *rewind* operacije. Tekuća pozicija može biti specificirana *fsetpos* ili *fseek* operacijom.

Sledeći karakteri mogu biti uključeni u mod radi specifikacije prevođenja *newline* karaktera.

**t** Otvara fajl u tekstualnom modu. CTRL+Z se interpretira kao end-of-file karakter na ulazu. U fajlu otvorenom za čitanje - pisanje sa "a+" *fopen* proverava CTRL+Z na kraju fajla i briše ga ako je moguće. Ovo se radi jer korišćenje *fseek* i *ftell* za kretanje unutara fajla koji se završava sa CTRL+Z, može da uslovi nekorektno ponašanje *fseek* blizu kraja fajla. Takođe, u tekstualnom modu carriage *return* - *line feed* kombinacije na ulazu se prevode u jedan linefeeds na ulazu i obratno *linefeed* karakter na izlazu se prevode u carriage *return* – *line feed* kombinacije.

**b** Otvara fajl u binarnom neprevedenom modu. Prevođenja koja uključuju *carriage-return* i *linefeed* kraktere se ne vrše.

Ako je otvaranje neuspešno, tj. ako *fopen*() ne može da pronađe željenu datoteku, ona vraća vrednost *NULL*.

**Primer.** Niz iskaza

```
#include<stdlib.h>
FILE *infile,*fopen();
infile=fopen("file","r"); }
```

otvara datoteku pod imenom *file* u režimu čitanja.

Funckija *fopen*() vraća ukazatelj na strukturu *FILE*, koja se dodeljuje promenljivoj *infile* istog tipa.

Neuspešno otvaranje datoteke *infile* se može ispitati izrazom oblika

```
if(infile==NULL)
        printf("datoteka file ne moze se otvoriti\n");
```

Poziv funkcije *fopen*() i ispitivanje vraćenog rezultata se može ujediniti

```
if((infile=fopen("file","r"))==NULL)
        printf("datoteka file ne moze biti otvorena\n");
```

Funkcijom *fclose*() zatvara se datoteka otvorena za čitanje ili upis. Zatvorenoj datoteci se ne može pristupiti pre ponovnog otvaranja. Zaglavlje ove funkcije je

```
int fclose(FILE *file_pointer);
```

gde je *file_pointer* ukazatelj kojim se identifikuje datoteka.

Funkcija

```
fclose(file_pointer)
```

vraća vrednost *EOF* ako ukazatelj *file_pointer* nije povezan sa nekom datotekom.

Na primer, otvorena datoteka *infile* zatvara se izrazom

```
fclose(infile);
```

**Primer.**

```
/* FOPEN.C: This program opens files named "data"
   and "data2".It  uses fclose to close "data" and
   _fcloseall to close all remaining files.  */
#include <stdio.h>
FILE *stream, *stream2;
void main(void)
{ int numclosed;
  /*  Open  for  read  (will  fail  if  file  "data"  does  not  exist)  */
    if( (stream = fopen("data", "r")) == NULL )
           printf("The file 'data' was not opened\n" );
    else   printf( "The file 'data' was opened\n" );
   /* Open for write */
   if( (stream2 = fopen( "data2", "w+")) == NULL )
           printf("The file 'data2' was not opened\n");
   else printf("The file 'data2' was opened\n");
   /* Close stream */
   if(!fclose(stream))
     printf( "The file 'data2' was not closed\n" );
    /* All other files are closed: */
   numclosed = _fcloseall();
   printf("Number of files closed by _fcloseall: %u\n", numclosed);
 }
```

```
Output
The file 'data'  was opened
The file 'data'  was opened
Number of files closed by _fcloseall :  1
```

## Funkcije fgetc() i fputc()

Funkcija *fgetc*() učitava jedan po jedan karakter iz specificirane datoteke. Slična je funkciji *getchar*(). Ulazni argument funkcije *fgetc*() je ukazatelj na strukturu *FILE*, a rezultat je vrednost tipa *int*. Zaglavlje ove funkcije je oblika:

```
int fgetc(FILE *infile)
```

Pretpostavimo da promenljiva *infile* dobija vrednost pomoću funkcije *fopen*(), koristeći opciju "*r*". Ako je *c* promenljiva tipa *int*, tada se iskazom

```
c=fgetc(infile);
```

učitava jedan znak datoteke koja je povezana sa pointerom *infile*.

Ako je dostignut kraj datoteke, funkcija *getc*() vraća vrednost *EOF*.

Funkcija *fputc*() upisuje karakter u specificiranu datoteku. Poziva se izrazom oblika

```
int fputc(char c, FILE *outfile)
```

## fputc

Zaglavlje ove funkcije je oblika

```
int fputc( int c, FILE *stream );
```

i piše karakter *c* u *stream*.

Vrednost koju ova funkcija vraća je karakter koji se upisuje, ako je upisivanje uspešno, ili EOF ako je nesupešno upisivanje.

**Primer.** Izrazima

```
FILE *outfile;
outfile=fopen("file","w");
fputc('z', outfile);
```

u datoteku *outfile* upisuje se znak 'z'.

**Primer.** Napisati program kojim se sadržaj datoteke "ulaz" (formiran od velikih slova azbuke) šifrira i upisuje u datoteku "izlaz". šifriranje se sastoji u tome da se svaki znak različit od '*Z*' zamenjuje sledećim ASCII znakom, dok se znak '*Z*' zamenjuje znakom '*A*'.

```
#include<stdio.h>
void main()
 { FILE *infile, *outfile;
   int c;
   infile = fopen("ulaz","r");  outfile= fopen("izlaz","w");
   while((c=fgetc(infile)) != EOF)
     { if('A' <= c && c < 'Z')c++;
       else c='A';
       fputc(c,outfile);
     }
   fclose(infile);   fclose(outfile);
 }
```

**Primer:**

```
/* FGETC.C: This program uses getc to read the first
* 80 input characters (or until the end of input)
* and place them into a string named buffer.
*/
#include<stdio.h>
#include<stdlib.h>
void  main(void)
{ FILE  *stream;
```

```
    char buffer[81];
    int  i, ch;
    /* Open file to read line from: */
    if( (stream = fopen("fgetc.c", "r")) == NULL ) exit(0);
  /* Read in first 80 characters and place them in "buffer": */
    ch = fgetc(stream);
    for( i=0; (i<80) && (feof(stream)==0); i++)
     {   buffer[i] = (char)ch; ch = fgetc( stream };  }
     /* Add null to end string */
     buffer [i] = '\0' ;
     printf("%s\n", buffer); fclose(stream);
  }
```

**Primer.** Kopiranje sadržaja ulazne datoteke u izlaznu datoteku. Imena datoteka data su kao parametri funkcije *main*().

```
#include<stdio.h>
void main(int argc,char *argv[]);
 { int c;   FILE *infile,*outfile;
    if((infile=fopen(argv[1],"r"))==NULL)
      printf("datoteka %s ne moze biti otvorena\n",argv[1]);
    else   if((outfile=fopen(argv[2],"w"))==NULL)
               printf("datoteka %s ne moze biti otvorena\n",argv[2]);
    else
     { while((c=getc(infile))!=EOF) putc(c,outfile);
       printf("\n");
     }
     fclose(infile);    fclose(outfile);
  }
```

**Primer.**

```
/* FPUTC.C: This program uses fputc
* to send a character array to stdout.
*/
#include<stdio.h>
void main(void)
{  char strptrl[] = "This is a test of fputc!!\n";
   char *p;
   /* Print line to stream using fputc. */
   p = strptrl;
   while((*p != '\0') && fputc(*(p++), stdout)!= EOF);
}
```

## Funkcije fprintf() i fscanf()

Ove funkcije obavljaju analogne operacije kao i funkcije *printf*() i *scanf*(), s tim što zahtevaju dodatni argument za identifikaciju datoteke u koju se upisuju ili iz koje se čitaju podaci. Zaglavlje ovih funkcija je sledećeg oblika:

fprintf(file_pointer,konverzioni_niz,lista_argumenata)

fscanf(file_pointer,konverzioni_niz,lista_argumenata)

pri čemu je:

file_pointer ukazatelj na tip FILE,

konverzioni_niz je lista formata po kojima se upisuju ili čitaju podaci, dok je *lista_argumenata* lista vrednosti koje se upisuju u datoteku, odnosno čitaju iz datoteke koja je identifikovana pointerom *file_pointer*.

Funkcije *fprintf*() i *fscanf*() imaju promenljivi broj argumenata, zavisno od broja vrednosti koje se upisuju u datoteku, odnosno čitaju iz datoteke.

Na primer, izrazom

```
fprintf(outfile,"string u datoteku\n");
```

upisuje se string "*string u datoteku*\n" u datoteku koja je identifikovana pomoću pointera *outfile*.

Takođe, u istu datoteku se može upisati i sledeći sadržaj:

```
fprintf(outfile,"string sadrzaja %s u datoteku\n",x);
```

**Primer.** Izrazom oblika

```
fscanf(infile,"%f",&x);
```

iz datoteke koja je identifikovana sa *infile* učitava se vrednost u formatu %*f* i dodeljuje promenljivoj *x*.

**Primer.** Procedure za učitavanje celog broja iz datoteke "*zad*2.*dat*" i upis istog broja u datoteku "*zad*2.*res*".

```
#include<stdio.h>
void citaj(long *n)
  { FILE *f;
     f=fopen("d:\\zad2.dat","r");  fscanf(f,"%ld",n);  fclose(f);
  }

void upis(long n)
  { FILE *f;
     f=fopen("d:\\zad2.res","w");  fprintf(f,"%ld",n); fclose(f);
  }

void main()
  { void citaj(long *);    void upis(long);
     long n;
     citaj(&n); upis(n);
  }
```

**Primer.** Iz prvog reda datoteke učitati broj timova. U svakom od sledećih redova datoteke zadati sledeće podatke za svaki tim:

- naziv tima,
- broj pobeda,
- broj nerešenih rezultata, i
- broj poraza.

Sortirati tabelu prema broju osvojenih bodova. Ako dva tima imaju isti broj osvojenih bodova, sortirati ih prema broju pobeda.

```
#include<stdio.h>
#include<string.h>
#define broj 10
struct klub
  { char naziv[30];
    int izgubljene;
    int neresene;
    int dobijene;
  } tabela[broj];

void ucitaj(int *n, struct klub a[broj])
  { int i;    FILE *f;
     f=fopen("klubovi.dat","r");
     fscanf(f,"%d",n); fgetc(f);
     for(i=0; i<*n; i++)
       { fgets(a[i].naziv,30,f);

fscanf(f,"%d%d%d",&a[i].dobijene,&a[i].neresene,&a[i].izgubljene);
          fgetc(f);
      }
```

```
      fclose(f);
   }

void ispis(int n, struct klub a[broj])
 { int i;
   for(i=0; i<n; i++)
    {  printf("%-20s %d %d %d\n",a[i].naziv, a[i].dobijene,
                        a[i].neresene,a[i].izgubljene);
    }
 }

void main()
  { int i,j,n, bodovi[broj]; klub as; int pb;
    ucitaj(&n,tabela);
    for(i=0;i<n;i++)
            tabela[i].naziv[(int)strlen(tabela[i].naziv)-1]='\0';

    for(i=0;i<n;i++)
      bodovi[i]=tabela[i].neresene+3*tabela[i].dobijene;
    for(i=0; i<n-1;i++)
      for(j=i+1;j<n;j++)
        if((bodovi[i]<bodovi[j])||
           ((bodovi[i]==bodovi[j]) &&
           (tabela[i].dobijene<tabela[j].dobijene))
              )
          {
                as=tabela[i]; tabela[i]=tabela[j];tabela[j]=as;
                pb=bodovi[i];bodovi[i]=bodovi[j];bodovi[j]=pb;
          }
    ispis(n,tabela);
  }
```

## Funkcija feof()

Ova funkcija ispituje da li je dostignut broj datoteke. Njen opšti oblik je

```
int feof(FILE *file pointer)
```

Funkcija *feof*() vraća nenultu vrednost ako je dostignut kraj datoteke.

**Primer.** Funkcija *feof* se koristi za ispitivanje kraja datoteke.

```
if(feof(infile))  printf("kraj datoteke\n");
```

**Primer.** Retka matrica se može predstaviti brojem ne-nula elemenata kao i sledećim informacijama za svaki ne-nula element:

- redni broj vrste tog elementa,
- redni broj kolone tog elementa, i
- realan broj koji predstavlja vrednost tog elementa.

Predstaviti retku matricu brojem ne-nula elemenata i nizom slogova, u kome svaki slog sadrži redni broj vrste i kolone kao i vrednost svakog ne-nula elementa.

Napisati procedure za formiranje niza slogova koji predstavljaju reprezentaciju retke matrice *A*. Podaci za matricu *A* se učitavaju iz datoteke 'sparse.dat'.

Napisati proceduru za ispis matrice.

Napisati proceduru za formiranje sparse reprezentaciju predstavlja matricu $A^T$.

Napisati funkciju koja za zadate vrednosti *i, j* preko tastature izračunava vrednost elementa *A*[*i,j*].

```
#include<stdio.h>

struct slog
  { int i,j;
```

```
    float aij;
  };

void pravimatricu(struct slog x[10], int *n)
  { int p,q,i;
    float xpq;
    FILE *infile;
     infile=fopen("sparse.dat","r");
     fscanf(infile,"%d", n);
     for(i=1; i<=*n; i++)
     { fscanf(infile,"%d%d%f", &p,&q,&xpq);
       x[i].i=p; x[i].j=q; x[i].aij=xpq;
     }
     fclose(infile);
  }

void transponuj(struct slog x[10], int n, struct slog y[10])
  { int p,q,i;
    float xpq;
     for(i=1; i<=n; i++)
     { y[i].i=x[i].j; y[i].j=x[i].i; y[i].aij=x[i].aij;
     }
  }

void pisimatricu(struct slog y[10], int n)
  { int p,q,i;
    float xpq;
     printf("%d\n",n);
     for(i=1; i<=n; i++)
     printf("(%d %d): %10.3f\n",y[i].i, y[i].j,y[i].aij);
  }

float vrednost(int n, struct slog x[10], int k, int l)
  { int i;
     for(i=1; i<=n; i++)
     if((x[i].i==k) && (x[i].j==l))
        return(x[i].aij);
     return(0);
  }

void main()
 { int i,j,n;
   struct slog p[10], q[10];
    pravimatricu(p, &n);
    printf("Matrica je : \n");  pisimatricu(p,n);
    transponuj(p,n,q);
    printf("Transponovana matrica: \n"); pisimatricu(q,n);
    printf("i,j= ? "); scanf("%d%d", &i,&j);
    printf("%10.3f\n",vrednost(n,p,i,j));
}
```

## Funkcije fgets() i fputs()

Funkcija *fgets*() ima sledeće zaglavlje:

```
char *fgets(char *string, int maxl, FILE *infile);
```

Ovom funkcijom se, iz datoteke na koju ukazuje pointer *infile*, učitava string koji se završava prelazom u novi red ili sadrži *maxl*-1 znakova. Prema tome, drugi argument, označen sa *maxl*, predstavlja maksimalnu dužinu učitanog stringa. Na kraj učitanog stringa postavlja se znak '\0'. U slučaju da linija iz koje se učitava sadrži više od *maxl* karaktera, linija se ne ignoriše, već se preostali znakovi učitavaju sledećim pozivom funkcije *fgets*().

Razlika između funkcija *gets*() i *fgets*() je u tome da *gets*() zamenjuje znak za prelaz u novi red null karakterom '\0', dok *fgets*() ostavlja znak za prelaz u novi red, pa tek onda postavlja znak '\0'.

Funkcija *fgets*() vraća vrednost *NULL* kada naiđe na znak *EOF*.

Funkcija *fputs*() poziva se izrazom

```
int fputs(char *string, FILE *outfile);
```

Pozivom ove funkcije upisuje se niz karaktera, sadržani u vrednosti parametra string, u datoteku na koju ukazuje pointer *outfile*. U slučaju I/O greške ova funkcija vraća rezultat *EOF*. Funkcije *puts*() i *fputs*() ne dopisuju karakter '\0' na kraj upisanog stringa. Za razliku od funkcije *puts*(), funkcija *fputs*() ne dopisuje znak za prelaz u novi red na kraj stringa koji se upisuje.

**fgets**

Čita string iz fajla

**char *fgets( char** **string*, **int** *n*, **FILE** **stream* );

Vraća string inače NULL ako je greška ili je došlo do kraja fajla.

Koristiti **feof** *ili* **ferror** za određivanje da li je došlo do greške.

Parametri:

| | |
|---|---|
| *string* | Mesto za smeštanje podataka |
| *n* | Maksimalni broj karaktera za čitanje |
| *stream* | Pointer na FILE strukturu |

Funkcija čita string sa ulaza zadatog argumentom stream i patmi ga u stringu. Čita karaktere od tekuće pozicije u fajlu do i uključujući prvi newline karakter, do kraja fajla ili do broja karaktera jednakog *n*-1, šta se prvo desi. Rezultat zapamćen u stringu je završen *null* karakterom. *Newline* karakter ako je pročitan je uključen u string. Funkcija *fgets* je slična *gets* funkciji, mada *gets* zamenjuje *newline* karakter sa NULL.

```
/* FGETS.C: This program uses fgets to display
* a line from a file on the screen.n*/
#include <stdio.h> void main( void )
{ FILE *stream;
  char line[100];
  if((stream = fopen("fgets.c", "r")) != NULL)
  { if(fgets(line, 100, stream) == NULL)
       printf("fgets error\n");
    else
       printf("%s" , line);
    fclose( stream );
  }
}
```

## fclose, _fcloseall

Zatvara fajl ili sve otvorene fajlove. **int fclose( FILE** **stream* ); **int _fcloseall( void );**

## feof

Testira end-of-file na streamu:

**int feof( FILE** **stream* );

The feof routine (implemented both as a function and as a macro) determines whether the end *ofstream* has been reached. When end of file is reached, read operations return an end-of-file indicator until the stream is closed or until **rewind, fsetpos, fseek,** or **clearerr** is called against it.

## 6.10.2. Binarne datoteke

### fread

Čita podatke iz *streama*.

**size_t fread(void **buffer*, **size_t** *size*, **size_t** *count*, **FILE** **stream* );**

Funkcija **fread** vraća broj zaista pročitanih itema, što može biti manje ili jednako parametru count ako se desi greška ili se stigne do kraja fajla. Koristiti feof ili ferror da bi se razlikovao uzrok. Parametri:

*buffer* Loakcija za podatke
*size* Veličina jednog itema u bajtovima
*count* Maximalni broj itema za čitanje *stream* pointer na **FILE** strukturu.

Funkcija čita do *count* itema a svaki je *size* bajtova sa ulaznog strima u buffer. File pointer povezan sa streamom je povećan za broj bajtova koji je zaista pročitan. Ako je stream otvoren u tekstualnom modu carriage return-linefeed par je zamenjen ejdnom linefeed characterom. File pointer nije određen ako se desila greška.

### fwrite

Upisuje podatke u *stream*.

**size_t fwrite( const void** **buffer*, **size_t** *size*, **size_t** *count*, **FILE** **stream* );

**fwrite** vraća broj potopunih itema koji su upisani, a sve ostalo je slično funkciji fread.

**Primer.**

```
/* FREAD.C: This program opens a file named FREAD.OUT and
* writes 25 characters to the file. It then tries to open
* FREAD.OUT and read in 25 characters. If the attempt succeeds,
* the program displays the number of actual items read.
*/
#include <stdio.h>
void  main()
{ FILE *stream;
  char list[30];
  int i, numread, numwritten;
  /* Open file in text mode: */
  if((stream = fopen("fread.out" , "w+t")) != NULL)
  { for ( i=0; i<25; i++ )
       list[i] = (char)('z' - i);
    /* Write 25 characters to stream */
    numwritten = fwrite(list, sizeof(char), 25, stream);
    printf("Wrote %d items\n", numwritten);
    fclose(stream);
   }
   else printf("Problem opening the file\n");
   if((stream = fopen("fread.out", "r+t" )) != NULL)
   { /* Attempt to read in 25 characters */
     numread = fread(list, sizeof(char), 25, stream );
     printf("Number of items read = %d\n", numread);
     printf("Contents of buffer = %.25s\n", list);
     fclose(stream);
   }
   else printf("File could not be opened\n");
}
```

### fseek

Premešta file pointer na specificiranu lokaciju.

**int fseek( FILE** **stream*, **long** *offset*, **int** *origin* );

Ako uspe fseek vraća inače vraća vrednost različitu od 0.

Parametri:

*stream* Pointer na strukturu FILE.
*offset* Broj bajtova od izvorišta (Number of bytes from *origin*).
*origin* Startna pozicija.

**fseek** funkcija pomera file pointer asociran sa streamom na novu lokaciju koja je *offset* bajtova od *ohgin.*

Tako sledeća operacija na streamu uzima poyiciju od nove lokacije.

Argument origin mora biti jeadn od:

SEEK CUR Current position of file pointer
SEEK END End of file
SEEK_SET Beginning of file

**ftell**

Vraća tekuću poziciju fajl pointera. **long ftell( FILE** **stream* );

Za rad sa tekstualnim fajlovima na raspolaganju su i funkcije pandani poznatim funkcijama prontf i scanf.

```
int fprintf(
   FILE *stream,
   const char * format[,
   argument ] . . .
)
```

```
int fscanf(
  FILE *stream,
  const char *format [,
  argument ]...
)
```

## Direktni pristup datotekama

**Primer.** U binarnoj datoteci je uređen niz od *m* celih brojeva, dodati prirodan broj *n* na odgovarajuće mesto. Koristiti sekvencijalno pretraživanje.

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

void main()
{ int niz[21], m,n,i;
  FILE *f=fopen("file.txt", "w+b");
  niz[0]=0;
  printf("Koliko brojeva u nizu?\n"); scanf("%d",&m);
  for(i=1; i<m; i++)  niz[i]=niz[i-1]+2;
  fwrite(niz, sizeof(int), m, f);
  fclose(f);

  printf("Formirana je sledeca datoteka:\n");
  f=fopen("file.txt", "r+b");
  fread(niz, sizeof(int), m, f);
  for(i=0; i<m; i++) printf("%d  ", niz[i]); printf("\n");
  printf("Broj koji se umece? "); scanf("%d",&n);
  i=0;   while((niz[i]<n)&&(i<m)) i++;
  fseek(f, i*sizeof(int), SEEK_SET);

  fwrite(&n, sizeof(int), 1, f);
  fwrite(niz+i, sizeof(int),m-i, f);
```

```
    fclose(f);  f=fopen("file.txt", "r+b");
    printf("Datoteka sa umetnutim brojem %d:\n",n);
    fread(niz, sizeof(int), m+1, f);
    for(i=0; i<=m; i++) printf("%d  ", niz[i]);
    fclose(f);
}
```

# Literatura


[1] Literatura sa predavanja.

[2] M. Čabarkapa, *C, Osnovi programiranja*, Biblioteka Algoritam, 2000.

[3] M. Čabarkapa, S. Matković, *C/C++*, Biblioteka Algoritam, Krug, Beograd, 2005.

[4] S. Stojković, *Programski jezik C sa rešenim zadacima*, Elektronski fakultet u Nišu, Edicija pomoćni udžbenici, Niš, 2005.

[5] V. Vujičić, *Uvod u C jezik*, Univerzitet u Beogradu, Institut za nuklearne nauke "Boris Kidrič", Vinča, 1996.

[6] L. Kraus, *Programski jezik C sa rešenim zadacima*, Akademska Misao, Beograd, 2004.

[7] D. Urošević, *Algoritmi u programskom jeziku C*, Mikro Knjiga, Beograd, 1996.

[8] M. Vugdelija, *Programiranje & Programiranje*,

[9] M. Stanković, *Programski jezici*, Elektronski fakultet u Nišu, Edicija: osnovni udžbenici, 2000.

[10] J.J. Dujmović, *Programski jezici i metode programiranja*, Akademska misao, Beograd, 2003.

[11] Mike Grant, Christian Skalka, Scott Smith, *Programming Languages*, Version 1.01, http://www.cs.jhu.edu/plbook, 2003.

[12] D.A. Watt, William Findlay, *Programming language design concept*, John Wiley & Sons Ltd., 2004.