Univerzitet u Nišu
Elektronski fakultet

dr Vladimir Ćirić

Uvod u programiranje i programski jezik C

```
1  #include <stdio.h>
2  void main()
3  {
4          int n,i,p,j,min,imin,a[30];
5          scanf ("%d",&n);
6          for ( i = 0; i < n; i++ )
7                  scanf ("%d",&a[i]);
8          min = A[0];
9          imin = n;
10         for ( i = 0; i < n; i++ )
11         {
12                 p=0;
13                 for ( j = 0; j < n; j++ )
14                         if ( a[i] == a[j] )
15                                 p++;
16                 if ( p < min )
17                 {
18                         min = p;
19                         imin = i;
20                 }
21         }
22         printf ("%d", a[imin]);
23 }
```

n,(a(i)i,0,n-1)
min=A(0)
imin=n
i=0,n-1
p=0
j=0,n-1
a(i) = a(j)
ne
da
p++
p<min
ne
da
min=p
imin=i
a(imin)

УНИВЕРЗИТЕТ У НИШУ
ЕЛЕКТРОНСКИ ФАКУЛТЕТ У НИШУ

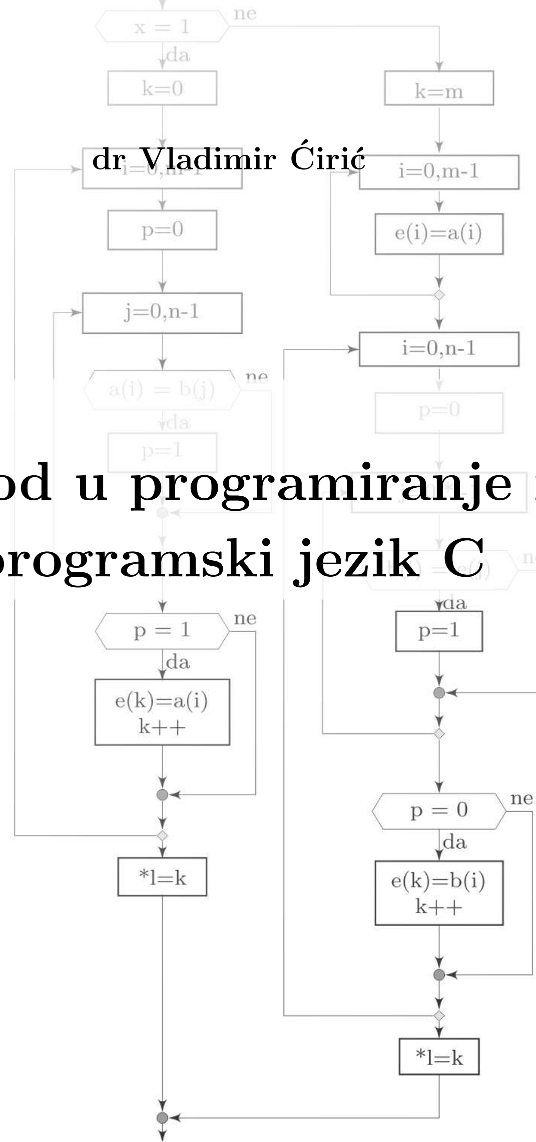
Edicija: **Osnovni udžbenici**

Niš 2014.

Univerzitet u Nišu
Elektronski fakultet

dr Vladimir Ćirić

Uvod u programiranje i programski jezik C



Niš, 2014.

UVOD U PROGRAMIRANJE I PROGRAMSKI JEZIK C

Autor Doc. dr Vladimir Ćirić,

Izdavač Elektronski fakultet u Nišu
P.fah 73, 18000 Niš
http://www.elfak.ni.ac.rs/

Recenzenti Prof. dr Dragan Janković, Elektronski fakultet u Nišu
Prof. dr Emina Milovanović, Elektronski fakultet u Nišu
Prof. dr Jelica Protić, Elektrotehnički fakultet u Beogradu

Glavni i odgovorni urednik: Prof. dr Dragan Tasić

Odlukom Nastavno-naučnog veća Elektronskog fakulteta u Nišu, br. 07/05-024/14-006 od 26.12.2014. godine, rukopis je odobren za štampu kao udžbenik na Elektronskom fakultetu u Nišu.

ISBN 978-86-6125-119-1

CIP - Каталогизација у публикацији - Народна
библиотека Србије, Београд

004.42(075.8)
004.432.2C(075.8)

ЋИРИЋ, Владимир, 1977-
Uvod u programiranje i programski jezik C /
Vladimir Ćirić. - Niš :
Elektronski fakultet, 2014 (Niš : Unigraf X-
Copy). - VI, 350 str. : graf.
prikazi ; 24 cm

Na vrhu nasl. str.: Univerzitet u Nišu. - Tiraž
500. - Bibliografija: str. 347. - Registar.

ISBN 978-86-6125-119-1

a) Програмирање b) Програмски језик "C"
COBISS.SR-ID 212405516

Preštampavanje ili umnožavanje ove knjige nije dozvoljeno bez pismene dozvole izdavača.

Tiraž: 500 primeraka
Štampa: SVEN d.o.o. Niš

Predgovor

Knjiga dr Vladimira Ćirića, Uvod u programiranje i programski jezik C, vredna je udžbenička literatura koja će biti od koristi, u prvom redu, studentima kojima je osnovna linija obrazovanja vezana za neki od računarskih pravaca, ali i onim studentima koji studiraju različite tehničke discipline, a neophodna su im programerska znanja.

Autor je rad na knjizi zasnovao na svom bogatom, dugogodišnjem programerskom i pedagoškom iskustvu. On pažljivo uvodi čitaoca u svet programiranja kroz poglavlja ove knjige. Početno poglavlje posvećeno je rešavanju problema uz pomoć računara i sadrži pažljivo selektovane činjenice iz istorije računarstva, kao i klasifikaciju programskih jezika. Potom se fokus prebacuje na algoritme, uz demonstraciju načina predstavljanja, opis dijagrama toka, kao i osnovnih i složenih algoritamskih struktura. Treće poglavlje započinje formalnim opisom sintakse programskih jezika, čime je urađena dobra priprema za prezentaciju osnovnih elemenata programskog jezka C, što je zapravo centralni predmet razmatranja u ovom delu. U posebnom poglavlju predstavljene su promenljive, tipovi podataka i operatori. Kako bi student, koji počinje sa programiranjem, imao jasniju sliku o tome sa kakvim podacima se radi i kako podaci mogu biti organizovani, autor je peto poglavlje posvetio osnovnim strukturama podataka. Poslednje, šesto poglavlje veoma sveobuhvatno razmatra funkcije i pomaže studentu da ovlada ovim ključnim sredstvom u programiranju.

Materijal sadrži veliki broj preciznih i dobro postavljenih defincija, objašnjenja su jezgrovita i orijentisana ka studentu, a posebnu vrednost predstavljaju pažljivo kreirani originalni primeri koji ilustruju svaku od tema. Kontrolna pitanja na kraju svakog poglavlja nemaju samo funkciju da provere znanje, već vrlo jasno ukazuju šta je to što student treba da prepozna i kao značajno usvoji iz prethodno prezentovanog materijala. Nadam se da će generacije studenata koje dolaze sa ovom knjigom dobiti dobar uvod u programiranje i da će sigurnim koracima savladati programski jezik C.

U Nišu,
27.10.2014. godine

Prof. dr Ivan Milentijević

2π

Uvod u programiranje i programski jezik C

Sadržaj

1 Rešavanje problema uz pomoć računara **1**
1.1 Mašinska obrada podataka 1
1.1.1 Tjuringova mašina 3
1.1.2 Fon Nojmanova arhitektura 6
1.1.3 Mašinsko i asemblersko programiranje 9
1.2 Klasifikacija programskih jezika 10
1.3 Faze u rešavanju problema uz pomoć računara 14

2 Algoritmi **27**
2.1 Tekstualni opis algoritma 28
2.2 Dijagrami toka 32
2.2.1 Početak i kraj programa 33
2.2.2 Unos i prikaz rezultata 34
2.2.3 Obrada podataka 35
2.2.4 Kontrola toka izvršenja programa 36
2.3 Osnovne i složene algoritamske strukture 37
2.3.1 Sekvenca 39
2.3.2 Alternacija 40
2.3.3 Petlje 41
2.3.4 Složene algoritamske strukture 64
2.4 Strukturogrami 74
2.5 Pseudokod 76

3 Osnovni elementi programskog jezika C **83**
3.1 Formalni opis sintakse programskih jezika 83
3.1.1 Bekus-Naurova forma 84
3.1.2 Proširena Bekus-Naurova forma 85
3.1.3 Sintaksni dijagrami 87
3.2 Programski jezik C 89
3.2.1 Razvoj i verzije C kompajlera 89
3.2.2 Karakteristike C-a 90
3.2.3 Prevođenje programa 91
3.3 Azbuka i tokeni C-a 92

3.3.1 Ključne reči 93
3.3.2 Identifikatori 94
3.3.3 Separatori 95
3.3.4 Konstante 96
3.3.5 Literali 99
3.3.6 Operatori i izrazi 100
3.4 Osnovna struktura C programa 101
3.5 Deklaracija promenljivih i konstanti 104
3.6 Standardni ulaz i izlaz 107
3.6.1 Standardni ulaz 108
3.6.2 Standardni izlaz 111
3.7 Osnovne algoritamske strukture C-a 114
3.7.1 *if-then* i *if-then-else* 116
3.7.2 *switch* 120
3.7.3 *while* 123
3.7.4 *do-while* 126
3.7.5 *for* 128
3.8 Naredbe bezuslovnog skoka 132
3.8.1 Naredba *continue* 133
3.8.2 Naredba *break* 134
3.8.3 Naredba *goto* 135
3.8.4 Primena naredbi bezuslovnog skoka 136
3.9 Preprocesorske direktive 144
3.9.1 Simboličke konstante i makroi 145
3.9.2 Uključivanje biblioteka u program 148
3.9.3 Uslovno prevođenje programa 149

4 Promenljive, tipovi podataka i operatori 155
4.1 Osnovni tipovi podataka i operatori 155
4.1.1 Numerički tipovi podataka 157
4.1.2 Znakovni podaci 166
4.1.3 Aritmetički operatori i operatori za rad sa bitovima 169
4.1.4 Logički podaci 174
4.1.5 Složeni operatori i operatori transformacije podataka 178
4.1.6 Prioritet operatora i prioritet tipova podataka 187
4.2 Izvedeni tipovi podataka 190
4.2.1 Pokazivači 191
4.2.2 Pokazivačka algebra 193
4.2.3 Strukture podataka 196
4.2.4 Koncept objektno-orjentisanog programiranja 200
4.2.5 Ugnježdene i samoreferencirajuće strukture podataka 201
4.2.6 Unije 205
4.2.7 Definisanje novih tipova 207

5 Osnovne strukture podataka 213
5.1 Linearne i nelinearne strukture podataka . . . 213
5.2 Polja . . . 214
5.2.1 Statička deklaracija polja . . . 216
5.2.2 Pristup elementima polja . . . 217
5.2.3 Linearizacija polja . . . 219
5.2.4 Osnovne operacije sa nizovima i matricama . . . 220
5.2.5 Sortiranje nizova . . . 227
5.2.6 Karakteristični delovi matrice . . . 235
5.2.7 Nizovi i pokazivači . . . 239
5.3 Stringovi . . . 241
5.3.1 Deklaracija i inicijalizacija stringova . . . 241
5.3.2 *Null-terminated* stringovi . . . 242
5.3.3 Unos i prikaz stringova . . . 243
5.3.4 Osnovne operacije nad stringovima . . . 246
5.3.5 Nizovi stringova . . . 253
5.4 Magacin i red . . . 254
5.4.1 Magacin . . . 254
5.4.2 Red . . . 256
5.5 Nelinearne strukture . . . 257
5.5.1 Lančane liste . . . 257
5.5.2 Stabla i grafovi . . . 258

6 Funkcije 263
6.1 Sintaksa funkcija u C-u . . . 264
6.2 Prenos parametara . . . 270
6.2.1 Prenos po vrednosti . . . 271
6.2.2 Prenos po referenci . . . 273
6.2.3 Nizovi i matrice kao parametari funkcije . . . 275
6.3 Funkcija *main* . . . 281
6.4 Rekurzivne funkcije . . . 283
6.5 Memorijske klase promenljivih . . . 286
6.5.1 Automatska klasa i lokalne promenljive . . . 286
6.5.2 Eksterna klasa i globalne promenljive . . . 288
6.5.3 Statičke promenljive . . . 290
6.5.4 Registarske promenljive . . . 291
6.6 Standardne funkcije C-a . . . 291
6.6.1 Funkcije za matematička izračunavanja . . . 292
6.6.2 Funkcije za rad sa stringovima . . . 294
6.6.3 Funkcije za dinamičku alokaciju memorije . . . 309
6.7 Standardni ulaz/izlaz i rad sa fajlovima . . . 316
6.7.1 Standardni tokovi podataka . . . 317
6.7.2 Fajlovi i korisnički tokovi podataka . . . 322
6.7.3 Tekstualni fajlovi . . . 328
6.7.4 Binarni fajlovi . . . 337

1

Rešavanje problema uz pomoć računara

Prva upotreba reči "računar", odnosno "kompjuter" (eng. *computer*), zabeležena je 1613. godine u kontekstu osobe koja vrši izračunavanja. Ovo značenje je zadržano do sredine 20. veka, mada je već u drugoj polovini 19. veka reč počela da dobija današnje značenje.

Moderno računarstvo razvijalo se kroz konstantno usavršavanje računarskih arhitektura i tehnika programiranja. Počev od relativno jednostavnih uređaja pomoću kojih je bilo moguće izvršiti jednostavne računske operacije, uređaji su evoluirali, a prava ekspanzija različitih programabilnih elektronskih kola desila se sa pojavom integrisanih kola.

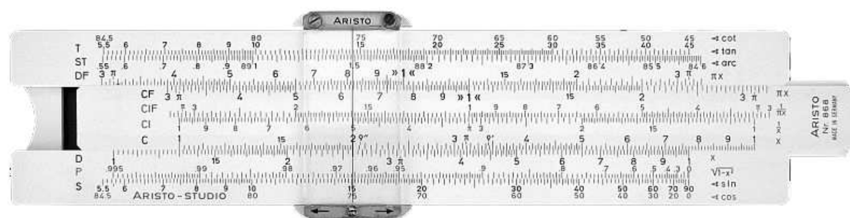
Iako je veoma teško identifikovati neki uređaj kao "prvi računar", vredi pomenuti neke od uređaja koji su uticali na razvoj računarstva.

1.1 Mašinska obrada podataka

Još od najranijih vremena ljudi su se trudili da konstruišu sprave koje bi im pomagale u izvođenju računskih operacija. Jedna od rasprostranjenijih i češće sretanih sprava kroz istoriju bio je abakus (računaljka), konstruisan pre oko 5.000 godina. Razvojem novih tehnologija usavršavale su se i računarske sprave. Poznate su mehaničke mašine za sabiranje koje su konstruisali Leonardo da Vinči (1425-1519) i Blez Paskal (1623-1662), kao i Lajbnicova mašina iz 1671. (Gottfried Wilhelm von Leibniz), koja je pored sabiranja i oduzimanja obavljala i operacije množenja i deljenja. Tek prvih decenija ovog veka pojavljuju se prve električne računarske mašine sastavljene od elektromagnetnih releja. Ove mašine su pravile veliku buku pri radu, jer je svaka tranzicija logičkih nivoa bila propraćena mehaničkim pomeranjem kontakata uz pomoć elektromagneta. Amerikanci su tokom Drugog svetskog rata razvili prvi potpuno elektronski računar ENIAC (*Electronic Numerical Integrator and Computer*), koji je radio na principu elektronskih cevi i bio prevashodno

namenjen za potrebe vojske. ENIAC je zauzimao prostor od 139 m^2, imao 19.000 termojonskih cevi i dioda i težio oko 30 tona. Revoluciju u razvoju elektronike, pa samim tim i računarstva, donosi pronalazak tranzistora, koji je omogućio da se relativno velike i nepouzdane elektronske cevi zamene daleko manjim i pouzdanijim ekvivalentima. Novu revoluciju u oblasti računarstva predstavljala je pojava tehnologije integrisanih kola, koja je omogućila da se na jednu silikonsku pločicu nevelikih dimenzija i potrošnje smeste hiljade, pa milioni, a u poslednje vreme i milijarde tranzistora. Zahvaljujući tome, računari su vremenom postajali sve manji i manji, a istovremeno i sve moćniji.

Jedan od mehaničkih uređaja za pomoć pri računanju, koji je bio u širokoj upotrebi do pre par decenija je logaritmar. Logaritmar (šiber), slika 1.1, je mehanički uređaj koji je konstruisao Edmund Ganter 1620. godine za računanje logaritma, koji može da obavlja različite računske operacije. Loragitmar je korišćen kao uređaj za računanje, umesto današnjih računara, čak na pet "Apolo" misija u svemiru, uključujući i misiju na Mesecu.[1]



Slika 1.1: Logaritmar (šiber): mehanički uređaj za računanje

Koncept koji nedostaje ovakvim uređajima da bi se mogli nazvati računarima u današnjem smislu te reči je mogućnost programiranja, odnosno zadavanja niza operacija i pamćenje rezultata i međurezultata.

Za prvog programera smatra se britanski matematičar i pisac, Ejda Bajron, grofica od Lavlejsa (Ada Augusta King, rođena Bajron, 1815-1852). Ejda je poznata po svom radu na prvom programabilnom mehaničkom računaru opšte namene koji je konstruisao Čarls Bebidž (Charles Babbage, 1791-1871), tzv. analitička mašina. Pomagala je Čarlsu Bebidžu u dokumentovanju rada ove mašine kao i u radu na njoj, ali i svojim predlozima, od kojih je najznačajniji bio prenos kontrole i rad sa ciklusima (petljama), tako da naredbe programa ne bi morale da se izvršavaju strogo u redosledu u kom su date, već u zavisnosti od toka programa. U svom radu je predviđala mogućnost upotrebe ove mašine i za opštije stvari, kao što je komponovanje muzike, ali i za šire naučne primene. Predložila je da se pomoću analitičke mašine računaju Bernulijevi brojevi. Ovaj plan se ujedno smatra i prvim

[1]Pokazna vežba o načinu upotrebe i principu rada logaritmara (eng. *Slide Rule*) može se naći na http://www.youtube.com/watch?v=waiprjueVpQ.

programom, a Ejda Lavlejs prvim programerom. U njenu čast jedan programski jezik dobio je njeno ime.

1.1.1 Tjuringova mašina

Za formalizaciju koncepta algoritama i programiranja, kao i savremenog računarstva, uzima se 1936. godina i Tjuringova mašina (Alan Turing, 1912-1954). Tjuring je ovu mašinu nazvao "a-mašina" (automatska mašina). Tjuringova mašina formalno objedinjuje koncept automatskog izračunavanja i programiranja. Ona predstavlja jednostavan uređaj za manipulaciju nad simbolima, zapisanih redom na beskonačnoj traci, na osnovu skupa pravila zadatih u vidu tabele konačne veličine. Mašina predstavlja konceptualni model koji se ne implementira kao takav, iako ga je tehnički moguće implementirati, već se zbog svojih osobina uglavnom koristi kao apstraktni pojam za definisanje drugih koncepata i ispitivanje granica mogućnosti mašinskog izračunavanja. Tjuringovu mašinu moguće je konceptualno prilagoditi tako da može simulirati bilo koji algoritam, a posebno je korisna u objašnjavanju funkcija procesora računara.

Tjuring je u eseju pod nazivom "Inteligentna mašinerija", u kom objašnjava svoj eksperiment, napisao:

"... memorija beskonačnog kapaciteta, formirana u vidu beskonačne trake, sadrži simbole odštampane jedan za drugim. U svakom momentu samo je jedan simbol u mašini. Ovaj simbol se naziva skenirani simbol. Mašina može da u zavisnosti od skeniranog simbola promeni svoje ponašanje, ali drugi simboli na traci ne utiču na tu promenu, već samo taj jedan skenirani simbol. Jedna od elementarnih operacija definisana simbolima je pomeranje trake napred i nazad..."

Američki matematičar Alonzo Čerč (Alonzo Church, 1903-1995), je u svom radu koji se oslanja na Tjuringovu formalnu teoriju računarstva, poznatom kao Čerč-Tjuringova teza, naveo da Tjuringova mašina predstavlja efikasan metod u matematici i logici i daje preciznu definiciju algoritma u vidu "mehaničke procedure".

Tjuringova mašina je *matematički model* mašine kojom se upravlja sadržajem trake. Na traci su simboli koje mašina može da piše i čita, jedan u datom trenutku, korišćenjem glave za čitanje/pisanje. Sve operacije su u potpunosti određene konačnim skupom elementarnih instrukcija kao na primer:

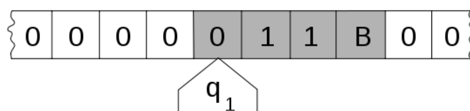
– "Instrukcija x: ukoliko je mašina u stanju S_y i pročitani simbol je i, upiši j, premotaj traku za 1 mesto u desno i pređi u stanje S_z."

Tjuringova mašina se sastoji od:

1. trake – traka je podeljena na ćelije, tako da svaka ćelija sadrži jedan simbol i ima samo dve susedne ćelije - jednu ispred i jednu iza;
2. glave – glava može da čita i piše simbole, kao i da pomera traku napred i nazad;

3. registar za pamćenje stanja – elementarna memorija veoma malog kapaciteta u kojoj se pamti trenutno stanje u kome se mašina nalazi. Ovaj registar Tjuring naziva "stanjem uma", po analogiji sa čovekom koji bi pamtio parcijalne rezultate dok računa složeniji matematički izraz sa više operacija.
4. konačna tabela instrukcija – tabela prelaska, ili drugačije, tabela akcija, koristi se za pamćenje promene stanja u zavisnosti od trenutnog stanja i pročitanog simbola.

Model Tjuringove mašine prikazan je na slici 1.2.



Slika 1.2: Primer stanja Tjuringove mašine

Tjuringova mašina sa slike 1.2 nalazi se u stanju q_1. Nule označavaju inicijalno stanje beskonačne trake, a sekvenca "11B" u ovom slučaju ilustruje primer nekog programa koji će se izvršiti. Značenje konkretnih simbola definiše se za svaku konkretnu mašinu.

Primer 1.1 (Tjuringova mašina za nadovezivanje nizova)**. Zadatak**: Tjuringova mašina treba da na niz od M simbola '1', koji je već upisan na traci, nadoveže niz od N takvih simbola, takođe već upisanih na traci, ukoliko se između ovih nizova nalazi simbol "+". Pre i posle nizova simbola '1' na traci se nalazi beskonačno simbola 'a'. Na primer, od niza "..aa1111+111aa.." treba da napravi niz "..aaa1111111aa..".

Rešenje: Azbuka (skup simbola) nad kojom ova Tjuringova mašina radi je

$$\mathbf{A} \in \{'1',' +',' a'\}.$$

Neka je početno stanje (S_1), i završno stanje (S_k); početno stanje je stanje u kome se mašina nalazi na početku rada, a kada mašina dode u završno stanje, prestaje sa radom. Glava može da se pomera za jedno polje ulevo (L), za jedno polje udesno (D), ili da ostane u mestu (M). U zavisnosti od stanja u kome se glava nalazi, i od simbola koji se nalazi u kućici iznad koje je glava postavljena, glava će u tu kućicu upisati određeni simbol, pomeriti se levo ili desno (ili ostati u mestu), i promeniti svoje stanje. Ovaj proces se ponavlja dok Tjuringova mašina ne stigne u završno stanje.

Stanja mašine su $\mathbf{S} \in \{S_1, S_2, S_3, S_4\}$, gde je S_1 početno, a S_4 krajnje stanje, a instrukcije, odnosno prelasci iz stanja u stanje u zavisnosti od pročitanog simbola mogu se definisati na sledeći način:

- S_1 **1** $\rightarrow$ **a** S_2 **D** - što znači da ako je glava u stanju S_1, i nalazi se nad poljem u kome je upisano 1, u to polje se upisuje 'a', glava se pomera desno, i prelazi u stanje S_2

- S_2 **1** $\rightarrow$ **1** S_2 **D** - ako je glava u stanju S_2, i nalazi se nad poljem u kome je upisano 1, u to polje se upisuje 1, glava se pomera udesno, i ostaje u stanju S_2
- S_2 **+** $\rightarrow$ **1** S_3 **L** - ako je glava u stanju S_2, i nalazi se nad poljem u kome je upisano +, u to polje se upisuje 1, glava se pomera ulevo, i prelazi u stanje S_3
- S_3 **1** $\rightarrow$ **1** S_3 **L** - ako je glava u stanju S_3, i nalazi se nad poljem u kome je upisano 1, u to polje se upisuje 1, glava se pomera ulevo, i ostaje u stanju S_3
- S_3 **a** $\rightarrow$ **a** S_4 **D** - ako je glava u stanju S_3, i nalazi se nad poljem u kome je upisano a, u to polje se upisuje a, glava se pomera udesno, i prelazi u završno stanje, S_4

Princip rada mašine sa ovakvim skupom stanja i instrukcija je da se prva jedinica sa leve strane levog niza zameni simbolom 'a', a znak '+' simbolom 1. Time je broj jedinica u levom nizu očuvan (jedna obrisana, a jedna dodata), a nizovi "izgledaju" spojeno.

Prateći instrukcije, sled događaja je ovakav: mašina polazi iz stanja S_1, a glava se nalazi nad poljem gde je prvi simbol '1'. Ona taj prvi simbol '1' briše i na njegovo mesto upisuje 'a' (prva instrukcija, S_1 1 $\rightarrow$ a S_2 D), i zatim se pomera desno sve dok ne naiđe na znak '+' (druga instrukcija, S_2 1 $\rightarrow$ 1 S_2 D). Umesto znaka plus, upisuje se '1' (treća instrukcija, S_2 + $\rightarrow$ 1 S_3 L), i time se dva niza spajaju u jedan. Zatim se glava pomera levo (četvrta instrukcija, S_3 1 $\rightarrow$ 1 S_3 L), sve dok ne naiđe na 'a', onda se vraća za jedno polje desno (kako bi bila na početku novog niza), i tu se rad završava (peta instrukcija, S_3 a $\rightarrow$ a S_4 D).

Potrebno je napomenuti da vraćanje na početak nije bilo neophodno. Takođe, važno je istaći da ovo nije jedino rešenje problema. △

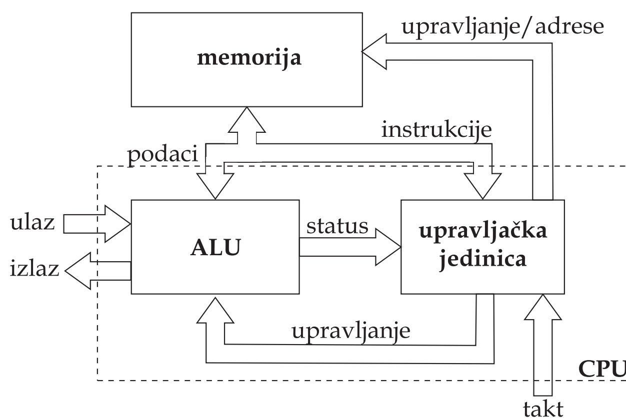
Koncept Tjuringove mašine postavio je temelje današnjih računara i procesora (centralna procesorska jedinica, eng. *Central Processing Unit* - CPU). Ukoliko se izuzmu protočni, superskalarni i višejezgarni procesori, koji u cilju ubrzanja izračunavanja u obradu uključuju neki vid paralelizma, može se reći da su kod svih procesora instrukcije zapamćene u operativnoj memoriji i izvršavaju se rednom jedna po jedna. Procesori se pored brzine obrade instrukcija razlikuju i po skupu instrukcija koje mogu da obavljaju. Međutim, kod svih procesora prisutne su sledeće elementarne operacije, nasleđene iz koncepta Tjuringove mašine:

1. upis i čitanje iz određene memorijske lokacije ili ulazno/izlaznih uređaja,
2. aritmetičke i logičke operacije nad pročitanim vrednostima,
3. bezuslovni skok na proizvoljnu instrukciju u memoriji od koje se dalje izvršenje nastavlja redom,
4. uslovni skok na proizvoljnu instrukciju u memoriji od koje se dalje izvršenje nastavlja redom.

1.1.2 Fon Nojmanova arhitektura

Na osnovu Tjuringovog modela, 1945. godine je Džon fon Nojman (John von Neumann) sa Prinston univerziteta predložio arhitekturu centralne procesorske jedinice računara poznatu kao "fon Nojmanova arhitektura", ili "Prinston arhitektura". Fon Nojmanova arhitektura prikazana je na slici 1.3. Ovu arhitekturu čine:

1. Centralna procesorska jedinica (eng. *Central Processing Unit* - CPU), koja se sastoji od
    - **aritmetičke jedinice** (eng. *Arithmetic Logic Unit* - ALU) za obavljanje aritmetičkih i logičkih operacija,
    - relativno malog broja procesorskih **registara** za pamćenje međurezultata, i
    - **upravljačke jedinice** za upravljanje radom procesorske jedinice, u skladu sa instrukcijom koja se trenutno izvršava.
2. Memorija - zajednička memorija za instrukcije programa i podatke. Instrukcije programa se smeštaju u jedan deo memorije, a podaci u drugi.
3. Eksterni uređaj velikog kapaciteta za skladištenje podataka.
4. Ulazno/izlazni uređaji - različiti uređaji koji se povezuju na računar.



Slika 1.3: Fon Nojmanova arhitektura računara

Upravljačka jedinica sadrži **programski brojač** i **registar za instrukcije** u koji se privremeno smešta jedna po jedna instrukcija iz memorije. U programski brojač upisuje se adresa memorije na kojoj se nalazi prva instrukcija programa, koju treba pribaviti iz memorije. Ovu adresu upravljačka jedinica postavlja na magistralu ka memoriji (vidi sliku 1.3), nakon čega memorija sadržaj adresirane

lokacije postavlja na magistralu za podatke/instrukcije. Pročitana instrukcija pamti se u registru instrukcija upravljačke jedinice.

Instrukcija se dekodira i na osnovu toga koja je instrukcija u pitanju šalju se upravljački signali ka aritmetičko-logičkoj jedinici. Ovi upravljački signali, pojednostavljeno rečeno, na principu multipleksera otvaraju i zatvaraju određene komunikacione kanale u ALU da bi se obavila željena instrukcija. Nakon obavljanja instrukcije i eventualnog upisa rezultata u lokalne registre ili memoriju, programski brojač se menja, čime se prelazi na narednu instrukciju.

Aritmetičko-logička jedinica i upravljačka jedinica čine centralnu procesorsku jedinicu. Centralna procesorska jedinica, pored komunikacije sa memorijom, ima i magistrale za komunikaciju sa drugim uređajima u vidu ulaza i izlaza (slika 1.3).

Na slici 1.3 oznaka "takt" ukazuje na spoljašni električni signal koji definiše radnu učestanost procesora, a koja predstavlja brzinu izvršenja elementarnih koraka CPU. U većini slučajeva je za izvršenje jedne instrukcije procesora potrebno više elementarnih koraka.

Generalno, sve instrukcije koje CPU može izvršiti mogu se svrstati u neku od sledećih kategorija instrukcija:

1. instrukcije ulaza i izlaza - vrše komunikaciju sa perifernim uređajima, tj. šalju i čitaju podatke sa ovih uređaja,
2. instrukcije za čitanje podataka iz memorije - upotrebljavaju se da bi se podaci učitali iz memorije radi dalje obrade,
3. instrukcije za upis podataka u memoriju - rezultati i međurezultati se ovim skupom instrukcija mogu preneti iz CPU, koji sadrži relativno mali broj registara za pamćenje rezultata, u memoriju velikog kapaciteta na trajnije čuvanje,
4. aritmetičko-logičke instrukcije - izračunavanja nad zadatim skupom podataka, čiji se rezultat privremeno smešta u registre CPU-a ili u memoriju,
5. instrukcije uslovnog i bezuslovnog skoka - ove instrukcije bezuslovno, ili u zavisnosti od zadatog uslova menjaju sadržaj programskog brojača, čime se kontroliše redosled izvršenja instrukcija programa, pa se tako određene grupe instrukcija mogu preskakati ili ponovo izvršavati u zavisnosti od zadatih kriterijuma.

Nakon "fon Nojmanove arhitekture" usledila su mnoga poboljšanja, ali je koncept ostao isti. Bitnije poboljšanje uvela je takozvana "Harvard arhitektura", kod koje postoje dve memorije: jedna za instrukcije a druga za podatke. Kod "Harvard arhitekture" moguće je istovremeno čitati narednu instrukciju dok se pribavljaju operandi za trenutnu instrukciju, čime se dobija na brzini. Jedan od trendova u projektovanju procesora od harvardske arhitekture do danas je paralalelizacija izvršenja onih instrukcija koje su, uslovno rečeno, nezavisne jedna od druge.

Na tržištu postoji veliki broj različitih procesora. Svaki procesor ima svoj skup instrukcija, koje je moguće klasifikovati u neku od navedenih elementarnih grupa. Osnovne karakteristike po kojima se razlikuju procesori, definisane arhitekturom, su:

- skup instrukcija koje procesor podržava,
- način na koji su instrukcije kodirane u memoriji,
- lokacije gde se mogu naći operandi, i
- načini adresiranja.

U zavisnosti od lokacije gde se mogu naći operandi razlikujemo registarsko-registarske, memorijsko-registarske i memorijsko-memorijske arhitekture. Registarsko-registarske arhitekture su arhitekture koje ne mogu direktno izvršiti operaciju nad operandima u memoriji računara. Kod ovih arhitektura pre izvršavanja izračunavanja prvo treba jednom instrukcijom jedan od operanada pročitati iz memorije i upisati u lokalni registar ALU, a zatim i drugi, pa tek tada izvršiti operaciju. Nakon toga se posebnom naredbom rezultat smešta u memoriju, pa je generalno potrebno **najmanje 4 instrukcije** da bi se izvršila operacija nad dva operanda smeštena u memoriji i rezultat upisao u memoriju.

Ne razmatrajući upis rezultata u memoriju, kod memorijsko-memorijskih arhitektura postoje implementirani putevi u arhitekturi tako da je **jednom instrukcijom** moguće npr. sabrati brojeve. Kod memorijsko-registarskih ovo je moguće obaviti sa **dve instrukcije**: prebacivanje jednog operanda iz memorije u registar procesora i sabiranje sa drugim koji je u memoriji.

Za svaki tip procesora definisan je konkretan skup i, uslovno rečeno, ponašanje instrukcija. Da bi program napisan za jednu generaciju procesora bilo moguće izvršiti na narednoj generaciji proizvođači se neprestano trude da održe kompatibilnost, u smislu zadržavanja skupa instrukcija, načina adresiranja i načina kodiranja instrukcija.

Svi procesori podržavaju osnovne instrukcije, dok neki procesori imaju instrukcije kojima se jednom instrukcijom mogu obaviti veoma složenije funkcije. Ukoliko procesor nema instrukciju koja može izvršiti složenu funkciju, ovakve funkcije se mogu implementirati programski izvršavanjem više osnovnih funkcija. Ovo će biti pokazano na primeru izračunavanja vrednosti funkcije $x = \sqrt[n]{a}$ u narednim poglavljima.

Na osnovu kriterijuma vezanog za broj instrukcija u skupu instrukcija nekog procesora razvila su se dva pravca u arhitekturama procesora: CISC i RISC procesori. CISC (eng. *Complex Instruction Set Computer*) je tip procesora koji ima veoma veliki broj kompleksnih instrukcija. RISC (eng. *Reduced Instruction Set Computer*), s druge strane, ima relativno mali broj instrukcija. Zbog jednostavnosti arhitekture može se reći da su RISC procesori generalno brži. Na primer, ukoliko je RISC procesor 100 puta brži od CISC procesora, a za izvršenje neke složene instrukcije koja je implementirana na CISC-u kao osnovna instrukcija, RISC procesoru treba 10 instrukcija, nije teško zaključiti da će izvršenje takve operacije na RISC procesoru biti 10 puta brže. Postavlja se pitanje zbog čega nije napušten CISC koncept? Odgovor je: kompatibilnost.

1.1.3 Mašinsko i asemblersko programiranje

Mašinski kod je jedini jezik koji mikroprocesor može da direktno izvrši (procesira) bez prethodne transformacije. Mašinsko programiranje je pisanje programa koji se direktno, bez prilagođavanja mogu izvršavati na datom procesoru. Naredbe mašinskog programa predstavljaju se nizom nula i jedinica. Tako na primer, ilustracije radi, program na mašinskom jeziku za mikrokontroler PIC16F84, koji se često koristi u automatici za kontrolu uređaja i procesa, izgleda ovako:

```
11 0000 0001 0000
11 1110 0001 0000
```

U konkretnom slučaju, navedene instrukcije sabiraju dve konstante. Prva instrukcija upisuje konstantu 10 u jedan od registara procesora, a druga sadržaju tog registra dodaje još 10, tako da je u registru nakon izvršenja programa vrednost 20. Obično se mašinski kod predstavlja u heksadekadnom zapisu, pa je tako prethodni mašinski program od dve instrukcije moguće napisati i ovako:

```
3010
3E10
```

Mane mašinskog programiranja su da programer mora dobro poznavati arhitekturu procesora za koju piše program, a eventualne greške se teško otkrivaju i ispravljaju. Takođe, programi napisani za jedan tip procesora ne mogu se izvršiti na drugom tipu.

Programeri gotovo nikada ne pišu programe direktno na mašinskom jeziku, jer zahteva obraćanje pažnje na brojne detalje koje bi jezik visokog nivoa automatski obradio, i takođe zahteva poznavanje i čestu proveru numeričkih kodova za svaku instrukciju koja se koristi. Iz ovog razloga programski jezici druge generacije pružaju dodatni nivo apstrakcije u odnosu na mašinske programe.

Nečitke nizove nula i jedinica kod mašinskih jezika vremenom su zamenile simboličke naredbe **asemblerskih jezika**. Kod asemblerskih jezika instrukcija je zapisana simbolima koji asociraju na dejstvo instrukcije, a registri se označavaju imenima. Prethodno navedeni primer na asemblerskom jeziku navedenog mikrokontrolera bio bi:

```
MOVLW 10
ADDLW 10
```

Mana asemblerskog jezika je i dalje to što programer mora poznavati arhitekturu procesora i načine adresiranja, ali je program čitljiviji i lakše se pronalaze i ispravljaju greške.

Cena uvođenja asemblera je ta da se ovako napisan program ne može direktno izvršiti. Pre izvršenja asemblerski program je potrebno **prevesti**, ili **kompajlirati** (eng. *compile*), tj. simboličke reči asemblerskog jezika prevesti u nule i jedinice mašinskog jezika i tek tada izvršiti program.

Problem poznavanja arhitekture rešavaju **viši programski jezici**. U više programske jezike spadaju: Fortran, Pascal, Ada, C, C++, Java, C#, itd. Viši programski jezici osmišljeni su tako da naredbe ne zavise od arhitekture procesora. Uglavnom se baziraju na govornom jeziku, u većini slučajeva engleskom, pa su kao takvi jednostavni za učenje. Čine ih precizno definisan način pisanja naredbi i upravljačkih struktura kojima raspolažu (**sintaksa** jezika). Na primer, program napisan na C-u za sabiranje dva cela broja je:

```
main()
{
    int a,b,c;
    a = 10;
    b = 5;
    c = a + b;
}
```

Naredbe su standardne i iste bez obzira na procesor, a da bi se program izvršio potrebno ga je pre toga prevesti prevodiocem za konkretan procesor na kome će se izvršavati. Tako, program napisan na C-u je isti za bilo koji procesor, a postoje prevodioci za Inteovu familiju procesora, za PIC mikrokontrolere, za sve vrste mobilnih telefona, za ARM procesore, itd.

Jedna naredba višeg programskog jezika prevešće se uglavnom u veći broj mašinskih instrukcija, a broj i redosled instrukcija zavise od arhitekture, o čemu vodi računa prevodilac, tj. kompajler. U ovom slučaju o arhitekturi brine prevodilac, a ne programer. Prevodioci za više programske jezike su složeniji od asemblerskih prevodilaca, ali im je u krajnjoj instanci namena ista - izvršni, mašinski kod.

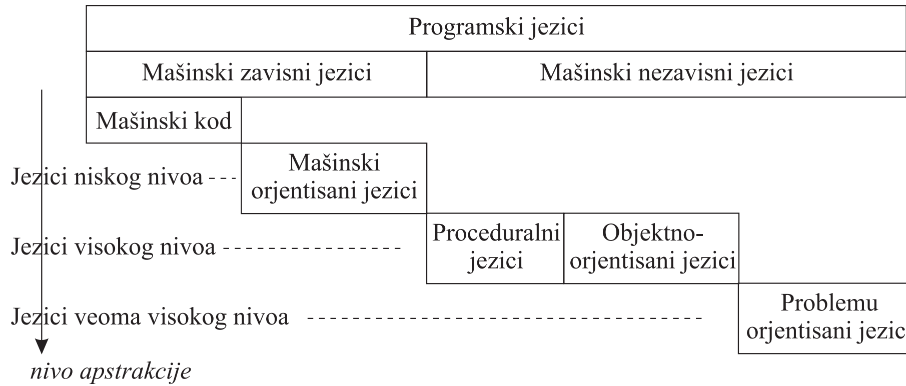
1.2 Klasifikacija programskih jezika

Programski jezici se po nivou apstrakcije mogu podeliti na programske jezike:

1. niskog nivoa,
2. visokog nivoa,
3. veoma visokog nivoa.

Grafička ilustracija podele programskih jezika po nivou apstrakcije i njihovoj zavisnosti od mašinskih jezika data je na slici 1.4.

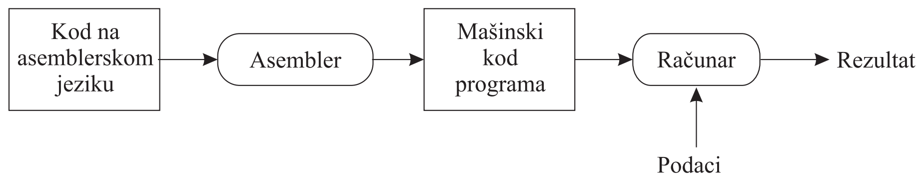
Programski **jezici niskog nivoa** su programski jezici koji pružju malo ili nimalo apstrakcije u odnosu na set instrukcija arhitekture računara. Generalno se ovo odnosi na mašinski ili asemblerski kod. Reč "niskog" se odnosi na skoro nepostojeću apstrakciju između jezika i mašinskog jezika. Zbog toga se često za jezike niskog nivoa kaže da su "blizu hardvera". Obično važi da su programi pisani na jezicima niskog nivoa brzi, jer programer vodi računa o najsitnijim mogućim detaljima, kao što je kretanje podataka kroz procesor. Ekvivalentan program napisan na jeziku



Slika 1.4: Klasifikacija programskih jezika po nivou apstrakcije

visokog nivoa je zahtevniji. Jezici niskog nivoa su jednostavni, ali se smatra da su teški za korišćenje, zbog raznih tehničkih detalja o arhitekturi procesora koje treba poznavati.

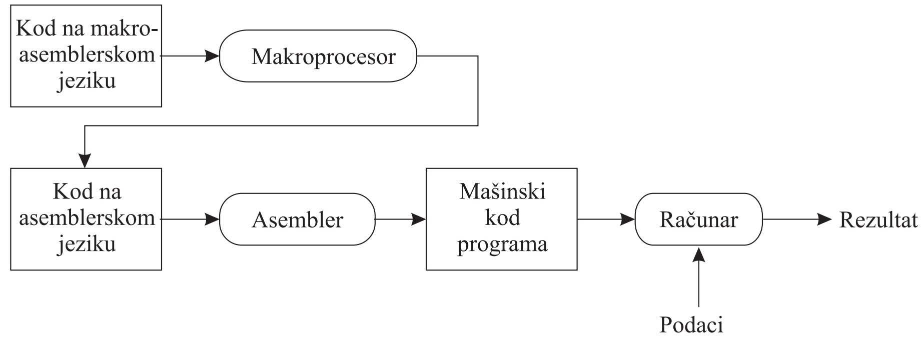
Za asemblerski jezik se kaže da je jezik niskog nivoa, iako nije jezik koji mikroprocesor može direktno izvršiti, zato što programer za pisanje programa na asemblerskom jeziku i dalje mora da poznaje arhitekturu mikroprocesora (npr. njegove registre, instrukcije, načine adresiranja). Asembler se u izvornom obliku ne može izvršiti, a proces generisanja izvršnog koda od izvornog koda na asemblerskom jeziku prikazan je na slici 1.5.



Slika 1.5: Proces generisanja izvršnog koda od izvornog koda na asemblerskom jeziku

Makro-asembler je specifičan asemblerski jezik kod koga, pored asemblerskih naredbi, postoje i tzv. makro-naredbe. Makro-naredbe su naredbe koje mogu predstavljati delove koda, pa je tako za delove koda koji se često ponavljaju moguće uvesti makro-naredbe i pojednostaviti pisanje. Ovo pojednostavljenje pisanja zahteva dodatnu fazu u prevođenju. Proces generisanja izvršnog koda od izvornog koda na makro-asemblerskom jeziku prikazan je na slici 1.6.

Asemblerski jezici se često označavaju skraćenicom *asm*, a makro-asembleri skraćenicom *masm*.



Slika 1.6: Proces generisanja izvršnog koda od izvornog koda na makro asemblerskom jeziku

Programski **jezici visokog nivoa** su programski jezici sa jakom apstrakcijom u odnosu na mašinski kod. U poređenju sa programskim jezicima niskog nivoa, programski jezici visokog nivoa mogu koristiti elemente iz prirodnih jezika. To ih čini razumljivijim i jednostavnijim za korišćenje. Kod programskih jezika visokog nivoa, umesto da se manipuliše registarima, memorijskim adresama i sl., radi se sa promenljivama, nizovima, objektima, složenom aritmetikom, procedurama i funkcijama, petljama, i drugim apstraktnim pojmovima, sa fokusom na upotrebljivost, zanemarujući donekle programsku efikasnost.

Količina apstrakcije definiše na koliko "visokom nivou" je programski jezik. U ovu grupu jezika spadaju **proceduralni jezici** i **objektno-orjentisani jezici**. Kod proceduranlih jezika opisuje se sled događaja, odnosno procedura, dok je kod objektno-orjentisanih jezika akcenat na opisivanju objekata kao aktera u događajima i njihove interakcije. Objektni jezici su nadgradnja proceduralnih i za sastavne delove opisa ponašanja objekata imaju procedure. Programski jezik C je primer proceduralnih jezika, a programski jezik C++ objektno-orjentisanih jezika. U proceduralne jezike spadaju i Pascal, Fortran, Cobol, Basic, i dr., a u objektno-orjentisane Java, C#, itd. Grafičke aplikacije su primer aplikacija pogodnih za opis objektnim jezicima. Na primer, objekti u *Windows* operativnom sistemu su: prozor, taster, pointer miša, a taster reaguje na klik miša tako što izvršava neku zadatu proceduru.

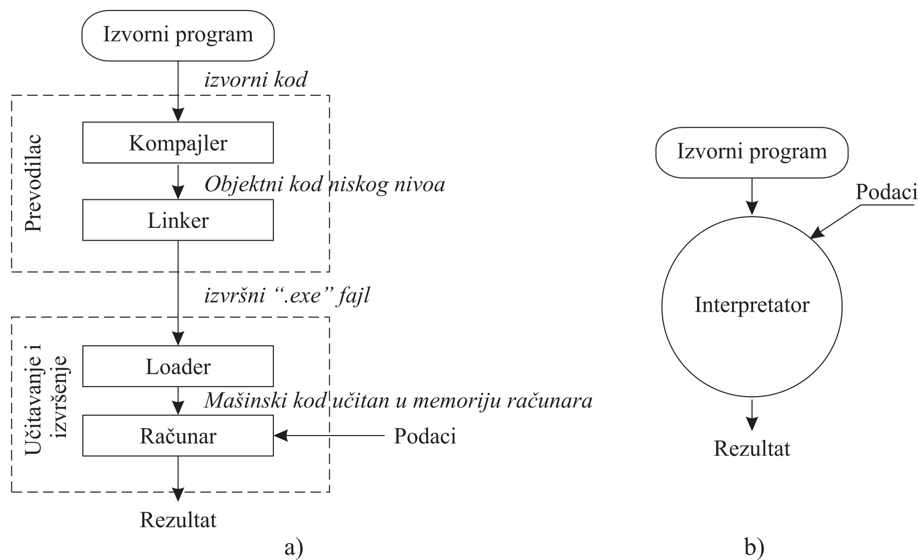
Po načinu izvršenja na računaru, razlikujemo dve vrste viših programskih jezika:

1. Kompajlere, i
2. Interpretere.

Interpreteri, ili drugačije poznati kao **skript** jezici (od engleske reči *script*), ne prevode program u celosti, već jednu po jednu naredbu i kako koju naredbu prevedu odmah je i izvršavaju. Interpreteri su računarski programi koji *interpretiraju* (čitaju i izvršavaju) jednu po jednu naredbu iz izvornog koda programa

napisanog na jeziku interpretera. Program napisan na interpreterskom jeziku se prevodi u toku izvršenja. Pri svakom pokretanju programa vrši se prevođenje jedne po jedne naredbe (interpretiranje). Program na interpreterskom jeziku nije moguće izvršiti ukoliko na računaru na kom se program izvršava nije instaliran interpreter za konkretni jezik. Skoro svi jezici za razvoj web aplikacija su ovog tipa (PHP, ASP, JavaScript, itd.)

Kompajlerski jezici program prevode u mašinski kod u celosti, pa ga tek nakon prevođenja izvršavaju. Rezultat prevođenja je izvršni fajl koji se može učitati od strane operativnog sistema u memoriju računara i izvršiti (slika 1.7a). Program se prevodi jednom i može se kao takav izvršiti proizvoljno mnogo puta, bez potrebe da se kompajler instalira na računaru na kome se izvršava program. Primeri kompajlerskih jezika su C, C++, Java, Fortran, Pascal, i dr. Ovakvi programi se po pravilu brže izvršavaju od interpretera. Proces prevođenja i izvršavanja kompajlerskih i interpreterskih jezika grafički je ilustrovan na slici 1.7.



Slika 1.7: Proces prevođenja i izvršavanja viših programskih jezika: a) kompajlerski jezici, b) interpreteri

Programski **jezici veoma visokog nivoa** su specifični jezici, često usko specijalizovani za određenu oblast. Karakteristika ovih jezika je veoma visok nivo apstrakcije, tako da je program pisan na jeziku ovog tipa veoma blizak govornom jeziku. Primeri jezika veoma visokog nivoa su Prolog i Lisp za programiranje, ili bolje rečeno, opisivanje sistema veštačke inteligencije.

1.3 Faze u rešavanju problema uz pomoć računara

Programi su vremenom postajali složeniji, pa se ukazala potreba da se koraci u projektovanju softvera sistematizuju, čime bi se omogućilo sagledavanje svake faze u cilju unapređenja i povećavanja efikasnosti procesa razvoja softvera. Bez obzira na složenost sistema koji se razvija, moguće je jasno izdvojiti faze u razvoju, a zavisno od veličine sistema svaka faza može biti implementirana na različite načine.

Proces razvoja softvera je predmet izučavanja posebnih grana računarstva, a ovde ćemo dati kratak pregled, kako bi na što bolji način sagledali poziciju algoritma u procesu projektovanja i razvoja softvera. Faze u rešavanju problema uz pomoć računara su sledeće:

1. Precizan opis problema
2. Izbor metode
3. Projektovanje modela/algoritma
4. Pisanje programa
5. Testiranje programa
6. Pisanje dokumentacije

Precizan opis problema

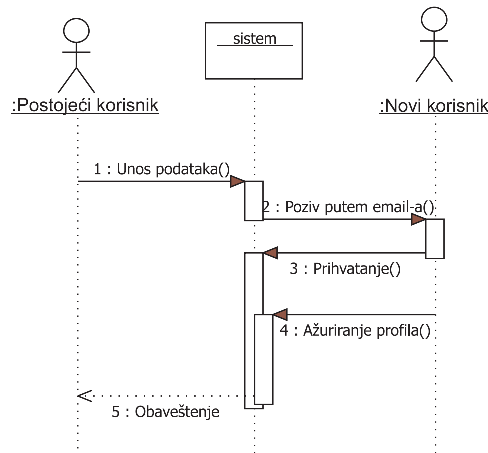
Svaki razvoj počinje preciznim opisom problema. Opis problema koji je potrebno rešiti uz pomoć računara treba da sadrži precizne informacije o načinu upotrebe programa, ulaznim podacima i očekivanim rezultatima. Opis problema može donekle sugerisati i način funkcionisanja rešenja, ukoliko je to neophodno, ali bez prevelikih ulaženja u detalje implementacije.

U najjednostavnijem obliku i na najvećem nivou apstrakcije, opis problema je zadatak koji je potrebno rešiti.

Primer 1.2. Napisati program koji niz od N elemenata uređuje u opadajući redosled. Broj elemenata niza i vrednosti elemenata zadaje korisnik. Podrazumevati da pri unosu nije bilo ponavljanja elemenata. Zatim učitati broj $K(K < N)$ i K brojeva redom. Posle svakog učitanog broja, ukloniti iz niza element koji je jednak učitanom broju, a ostale elemente pomeriti ulevo kako bi se popunila nastala praznina. Ukoliko se broj ne nalazi u nizu, niz ostaje nepromenjen. Prikazivati niz nakon sortiranja i nakon svakog izbacivanja. △

Kod složenijih sistema, ukoliko je u pitanju razvoj komercijalnog sistema, dokument sa opisom problema služi kao deo ugovora između strane koja razvija sistem i korisnika sistema. Pošto je u ovakvim slučajevima neophodno nedvosmisleno formalno definisati funkcionisanje sistema razvijeni su posebni jezici za opis. Ovi jezici predstavljaju formalnu grafičku reprezentaciju načina funkcionisanja sistema i osmišljeni su tako da budu čitljivi kako ekspertima, tako i korisnicima koji ne moraju nužno poznavati jezik za reprezentaciju. Najpoznatiji iz ove grupe jezika je UML (eng. *Unified Modeling Language*).

Primer 1.3. Jedan od tipova dijagrama UML-a je i "dijagram sekvence" koji je ilustrovan u ovom primeru. Dijagram sekvence opisuje zavisnost između događaja na taj način što po vertikalnoj osi daje sled događaja, a po horizontalnoj aktere u događajima. Na primer, scenario registracije korisnika na neki hipotetički sistem, po sistemu prihvatanja poziva od strane postojećeg korisnika, može biti opisan na način prikazan na slici 1.8. Po scenariju grafički opisanom na slici 1.8, nakon unosa



Slika 1.8: Strategija registracije korisnika

podataka sistem prosleđuje poziv novom korisniku, čeka na potvrdu i popunjavanje profila, o čemu izveštava onoga ko je uputio poziv. △

Izbor metode

Izbor metode takođe zavisi od veličine sistema koji se razvija. Kod procesa razvoja velikih sistema postoji veliki broj metoda čija primenljivost zavisi od situacije. Tako na primer, ukoliko iz nekog razloga nije moguće sa korisnikom precizno definisati sistem primenjuje se tehnika po kojoj se sistem razvija u malim inkrementima. Ovaj metod je u literaturi poznat kao inkrementalni razvoj. Svaki od inkrementa je jedna funkcionalna verzija sistema, koja se daje korisniku na reviziju. Nakon dobijanja mišljenja od korisnika sistem se prerađuje i prilagođava i ponovo daje korisniku, i tako redom dok se ne dobije finalna verzija sistema.

Kod relativno malih sistema, na kojima ćemo se kroz ovaj udžbenik zadržati, kod kojih je očekivani ishod izračunavanje vrednosti neke matematičke funkcije, razlikujemo dve grupe matematičkih pristupa u rešavanju samog problema:

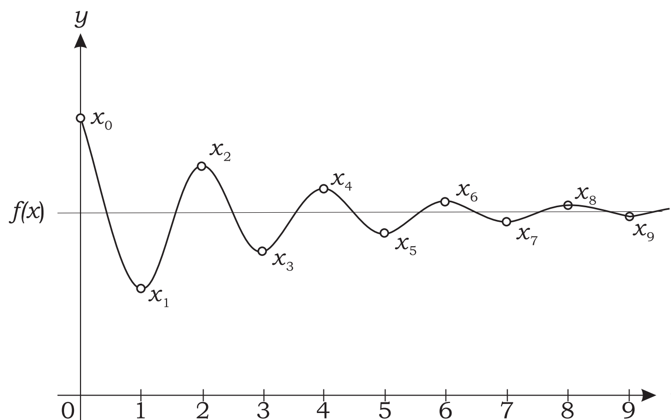
- tačne, i
- približne metode.

Razmotrićemo ovu fazu detaljnije iz ugla tačnih i približnih metoda. Kako naziv kaže, rezultat izvršenja programa koji implementira tačnu metodu je tačan rezultat, dok se približnim metodama dobija rezultat koji nije u potpunosti tačan, ali je dovoljno blizu tačnom rešenju da ga korisnik može smatrati zadovoljavajućim.

Veliku grupu približnih metoda čine **iterativni metodi**. Ovi metodi su predmet istraživanja posebnih grana matematike. Iterativni metodi imaju veoma značajnu ulogu u računarstvu, kao i tehničkim naukama uopšte, pa ćemo im ovom prilikom posvetiti posebnu pažnju.

U tehnici nije uvek neophodno poznavati tačan rezultat matematičkih funkcija. Tako na primer, ako se radi o nekom složenom integralu koji daje površinu nekog objekta, nije uvek od značaja poznavati rešenje u zatvorenom obliku, već imati numeričku vrednost koja izražava površinu, npr. $123,05\ m^2$. Numerička vrednost rezultata funkcije za zadate parametre može se odrediti iterativnim postupkom za datu funkciju. Ovde se nećemo baviti načinima za generisanje iterativnih postupaka, već programskom implementacijom ovakvih postupaka na računaru.

Iterativni metod je matematički obrazac, koji opisuje niz vrednosti koji u beskonačnosti teži vrednosti zadate funkcije. Za svaku matematičku funkciju moguće je kreirati iterativni metod koji opisuje način generisanja članova niza, tako da elementi niza teže rešenju funkcije. Pojednostavljeno rečeno, iterativni metodi opisuju početnu vrednost i kako od te vrednosti dobiti "tačniju" vrednost, pa od te "još tačniju", itd. Ovaj postupak je ilustrovan na slici 1.9. Kod metoda prikazanog



Slika 1.9: Konvergencija niza zadatog iterativnim postupkom

na slici 1.9, na osnovu zadatog parametra funkcije x određuje se početna vrednost x_0. Ova vrednost se koristi da bi se odredila "naredna" vrednost x_1, koja je bliža tačnoj vrednosti od x_0. Vrednost x_1 se koristi da bi se odredila vrednost x_2, i tako dalje, dok se vrednost niza dovoljno ne približi tačnoj vrednosti. Svaki od ovih koraka predstavlja jednu iteraciju, pa otuda i naziv.

Za zadatu funkciju $y = f(x)$, iterativni postupak definiše skup od n početih vrednosti $\{x_0, x_1, \ldots, x_{n-1}\}$, gde je $n \geq 1$, kao i funkciju I koja definiše način kako se od već izračunatih vrednosti može dobiti naredna vrednost:

$$x_i = I(x_{i-1}, x_{i-2}, \ldots, x_{i-n}), \quad i \geq n.$$

Način formiranja iterativnog metoda za zadatu funkciju je predmet posebnih grana matematike i ovde neće biti razmatran. Zadržaćemo se na implementaciji približnih matematičkih metoda. U narednim poglavljima biće pokazano kako se iterativni metodi mogu implementirati algoritmom i izvršiti na računaru.

Primer 1.4 (Iterativni metod za izračunavanje korena broja)**.** Iterativni postupak za izračunavanje vrednosti funkcije $x = \sqrt[n]{a}$ je

$$\begin{aligned} x_0 &= \frac{(a + n - 1)}{n} \\ x_{i+1} &= \frac{\left((n-1) \cdot x_i + \frac{a}{x_i^{n-1}}\right)}{n}, \quad i = 0, 1, 2, \ldots. \end{aligned}$$

Ovaj iterativni postupak definiše jednu početnu vrednost (x_0), koja zavisi od zadatih parametara (a, n), kao i način na koji se može odrediti naredna vrednost x_{i+1} ukoliko se zna prethodna vrednost x_i. Ilustrovaćemo ovaj iterativni postupak na primeru određivanja vrednosti $\sqrt[2]{3}$ i $\sqrt[2]{2}$.

Za primer $x = \sqrt[2]{3}$, imamo da je $n = 2$ i $a = 3$. Po iterativnom obrascu, početnu vrednost x_0 određujemo kao

$$x_0 = \frac{(3 + 2 - 1)}{2} = 2.$$

Na osnovu ove vrednosti narednu vrednost dobijamo kao

$$x_1 = \frac{\left((2-1) \cdot 2 + \frac{3}{2^{2-1}}\right)}{2} = 1,75.$$

Tačniju vrednost x_2 dobijamo zamenom vrednosti x_1 u iterativni obrazac, pa je $x_2 = 1,73214$, čime već dobijamo tačnost na treću decimalu. Ukoliko želimo još veću tačnost možemo nastaviti ovaj postupak dalje.

Vrednost $x = \sqrt[2]{2}$, možemo odrediti uzevši $n = 2$ i $a = 2$. Po iterativnom obrascu, početna vrednost x_0 je

$$x_0 = \frac{(2 + 2 - 1)}{2} = 1,5.$$

Narednu vrednost u nizu $\{x\}$ dobijamo kao

$$x_1 = \frac{\left((2-1) \cdot 1.5 + \frac{2}{1.5^{2-1}}\right)}{2} = 1,4166.$$

Tačniju vrednost moženo dobiti zamenom vrednosti x_1 u iterativni obrazac, itd.

Iterativni postupak opisan u ovom primeru spada u grupu takozvanih "brzih" iterativnih postupaka, jer se za relativno mali broj iteracija dobija zadovoljavajuća tačnost. Pored brzo konvergentnih postupaka postoje i sporo konvergentni iterativni procesi kod kojih je zadovoljavajuću tačnost moguće dobiti tek nakon više hiljada iteracija, pa je razlog implementacije ovakvih algoritama na računaru očigledan. △

Iterativnim postupcima svaku matematičku funkciju moguće je svesti na elementarne matematičke operacije. Ovim je omogućeno da procesori koji obavljaju samo elementarne matematičke funkcije mogu da vrše izračunavanja složenih funkcija. Iterativni postupci omogućavaju izračunavanje vrednosti svake matematičke funkcije samo pomoću operacije sabiranja. Oduzimanje je sabiranje broja i komplementa broja, množenje se u krajnjoj instanci može svesti na sabiranje, kao i deljenje.

Po ovom principu, ručni kalkulatori (tzv. digitroni) koji imaju veoma jednostavnu ALU računaju vrednost složenih matematičkih funkcija, uz napomenu da je vreme potrebno za izračunavanje direktno proporcionalno složenosti funkcije, tj. brzini konvergencije iterativnog procesa i željenoj tačnosti. Kod ručnih kalkulatora za svaki taster je vezana po jedna matematička funkcija implementirana svojim iterativnim postupkom, a sa tačnošću se ide do broja decimalnih pozicija koje ekran može prikazati.

Projektovanje algoritma

Nakon izbora metode za rešavanje problema, potrebno je izvršiti dekompoziciju problema, odnosno deljenje problema na manje celine, koje je moguće jednostavnije rešiti. Kao i kod prethodnih faza, i ova faza zavisi od veličine problema.

Za projektovanje složenih softverskih sistema najčešće se koristi ranije pomenuti UML. UML-om se opisuju celine i veze, poruke koje se razmenjuju, i sl. Za precizan opis bilo kog dela sistema koji zadati problem rešava u vidu niza koraka koriste se algoritmi.

Definicija 1.1. Algoritam je precizno definisani postupak koji vrši transformaciju ulaza u izlaz i u konačnom broju koraka vodi rešenju zadatog problema.

Algoritmom se može opisati bilo koji postupak sa jasno definisanim koracima i sledom događaja. Na osnovu definicije 1.1, algoritmom se može nazvati i postupak dat u sledećem primeru.

Primer 1.5. Algoritam za kuvanje čaja glasi:

1. Uzeti lonče i sipati vodu.
2. Uključiti ringlu.
3. Sačekati dok voda ne proključa.
4. Kada voda proključa, pomeriti lonče i isključiti ringlu.

5. Staviti kesicu čaja u lonče i sačekati 5 minuta.
6. Po želji, dodati kašiku šećera, mleko ili limun.
7. Sipati čaj u šolju.

Reč *algoritam* (eng. *algorithm*) je reč latinskog porekla, koja je svoje današnje značenje dobila u 12. veku greškom u prevodu. Naime, persijski astronom i matematičar Abu Abdullah Muhammad ibn Musa Al-Khwarizmi 825. godine napisao je knjigu *"Računanje pomoću hindu-brojeva"* (eng. *"On Calculation with Hindu Numerals"*). Knjiga je u 12. veku sa arapskog prevedena na latinski kao *"Algoritmi de numero Indorum"*, odnosno, u prevodu *"Algoritmi o brojevima Hindusa"*, gde je reč "Algoritmi" u nazivu na latinskom predstavljala genitiv imena autora. Ova reč je pogrešno tumačena kao imenična množina latinske reči, što je dovelo do današnjeg naziva "algoritmi" (lat. *algorismus*), t.j. metod izračunavanja. Slovo **h** iza slova **t** u nazivu na engleskom je najverovatnije dodato pogrešnim povezivanjem ove reči sa grčkim "$\alpha\rho\iota\theta\mu o\varsigma$" (arit**h**mos), što znači broj.

Prelazak iz stanja u stanje ne mora biti strogo definisan i jasno određen. Postoji grupa algoritama, poznata kao "neodređeni algoritmi" (eng. *probabilistic algorithms*), kod kojih se definiše verovatnoća prelaska iz stanja u stanje i koji uključuju slučajnost u izvršenje. U ovoj knjizi bavićemo se isključivo algoritmima sa precizno definisanim prelazima između koraka.

Različiti načini za predstavljanje algoritama biće prikazani u narednom poglavlju. Bez ulaženja u detalje vezane za način predstavljanja algoritama u narednom primeru ilustrovaćemo algoritam za nalaženje najvećeg zajedničkog delioca (NZD) zadatih brojeva M i N. Praćenjem koraka ovog algoritma moguće je odrediti NZD bilo koja dva zadata broja. Ovaj algoritam je u literaturi poznat kao Euklidov algoritam (grčki matematičar, oko 300. godine pre n.e.).

Primer 1.6 (Euklidov algoritam)**.** Neka su brojevi M i N zadati iz skupa celih brojeva, tako da je $M > N$, što ne umanjuje opštost algoritma.

1. Podeliti M sa N i ostatak zapamtiti u R;
2. Ako je R jednako 0 preći na korak 4, inače preći na korak 3;
3. Zameniti M sa N, a za tim N sa R, i preći na korak 1;
4. NZD je N. Kraj.

Ilustrovaćemo ovaj primer konkretnim vredostima.

Neka je $M = 20$ i $N = 15$. Na osnovu 1. koraka potrebno je ostatak deljenja M/N zapamtiti u R, t.j. $R = (M \mod N) = 5$. Kako je $R = 5 \neq 0$ po 2. koraku treba preći na korak 3, gde M dobija vrednost N, t.j. $M = N = 5$, a N dobija vrednost R, t.j. $N = R = 5$, nakon čega se vraća na korak 1. Novi ostatak deljenja M i N je jednak 0, pa se po koraku 2 prelazi na korak 4, koji kaže da je NZD$= N = 5$, čime je algoritam okončan. $\triangle$

Osnovne osobine algoritama su:

1. **Diskretnost** – Svaki algoritam se sastoji od jasno izdvojenih koraka. Algoritam čini konačno mnogo koraka. Diskretnost se dobija dekompozicijom problema, odnosno deljenjem problema na manje korake.
2. **Determinisanost** – Svaki korak algoritma treba da ima precizno određene ulaze, izlaze i zadatke koje obavlja.
3. **Efikasnost(konačnost)** – Svaki algoritam treba da dâ rezultat nakon konačnog broja koraka.
4. **Rezultativnost** – Algoritam treba da dâ rezultat za svaki skup ulaznih podataka.
5. **Masovnost** – Dobar algoritam rešava određeni problem za celu klasu ulaznih parametara, a ne samo za partikularni slučaj.
6. **Optimalnost** – Optimalan algoritam vodi rešenju nekog problema za što manji broj koraka.

Pisanje programa

Kao što se iz prethodnog primera može videti, algoritmi u većini slučajeva nisu opterećeni detaljima programskih jezika. Algoritam je postupak koji vodi rešenju konkretnog problema, bez obzira da li se implementira na računaru, ili se koristi kao postupak za računanje na papiru, bez računara, kao u ilustraciji sa konkretnim vrednostima u primeru 1.6.

Definicija 1.2 (Program)**.** Program je implementacija algoritma na računaru instrukcijama konkretnog programskog jezika.

Skup instrukcija koje čine program često se u žargonu nazivaju "programski kod", ili "izvorni kod" (od engleskih reči *source code*), ili kraće "kod".

Dizajn algoritma je faza projektovanja, a pisanje programa je faza implementacije algoritma na računaru. Najvažnija odluka koju je u ovoj fazi potrebno doneti je izbor jezika na kome će program biti napisan. Veliki je broj dostupnih programskih jezika i teško je izdvojiti neki jezik kao "najbolji". U krajnjoj instanci, da postoji "najbolji" jezik, samo bi taj jezik i postojao kao takav. Svaki programski jezik ima svoje prednosti i mane, koje se izražavaju u vidu karakteristika programskog jezika. U nekom slučaju karakteristika može biti prednost, a u nekom mana programskog jezika. U zavisnosti od namene programa neke karakteristike su poželjne, a neke ne. Najčešće karakteristike koje se koriste prilikom poređenja programskih jezika su:

1. **Čitljivost** – Dobar jezik rezultuje čitljivim programom koji je blizak govornom jeziku. Na tržištu postoje ekstremni slučajevi programskih jezika kod kojih program pre liči na dokumentaciju sa opisima šta bi program trebalo da radi, nego na sekvencu instrukcija.

2. **Prenosivost** – prenosivost ili portabilnost je karakteristika jezika da je program bez modifikacija, ili sa manjim izmenama moguće preneti na drugu platformu.

3. **Opštost** – je karakteristika po kojoj je u jednom programskom jeziku moguće rešiti puno različitih problema, bez potrebe da se koriste različiti jezici. Na primer, dovoljno opšti programski jezik omogućava razvoj video-igara sa dobrom grafikom, pisanje korisničkih programa uključujući i dizajn interfejsa, razvoj web aplikacija, i sl.

4. **Konciznost zapisa** – je mogućnost da se algoritam implementira sa što manje instrukcija.

5. **Jednostavnost otklanjanja grešaka** – jednostavnost otklanjanja grešaka je karakteristika koja je u direktnoj vezi sa čitljivošću programa.

6. **Cena** – vreme potrebno prosečnom programeru da implementira algoritam. Za program na npr. mašinskom jeziku je vreme razvoja duže, pa je i cena automatski veća.

7. **Prepoznatljiva notacija** – jezik treba da ima prepoznatljivu notaciju, tako da većina programera koji znaju neki drugi programski jezik mogu prepoznati sintaksu. Danas je najprepoznatljivija notacija jezika C. Mnogi programski jezici su "dijalekti" programskog jezika C, t.j. imaju veoma sličnu sintaksu, kao što su C++, Java, C#, J#, PHP, JavaScript, itd.

8. **Brzo prevođenje (kompajliranje)** – Vreme potrebno kompajleru da na osnovu instrukcija višeg programskog jezika generiše mašinski kod.

9. **Efikasnost** – je karakteristika koja se odnosi na mogućnost kompajlera da kreira efikasan kod koji će se brzo izvršavati. Generalno pravilo koje važi je da je efikasnost obrnuto proporcijalna prenosivosti, opštosti, jednostavnosti otklanjanja grešaka i ceni, a direkno proporcionalna brzini kompajliranja. Na primer, asemblerski jezici su vema efikasni, brzo se prevode na mašinski, ali nisu prenosivi, skupi su, i sl.

10. **Modularnost** – Poželjno je da programi mogu da se razvijaju na jeziku kao skup posebno izrađenih modula, uz obezbeđivanje odgovarajućih mehanizama za povezivanje modula. Modularnost jezika omogućava timski rad.

11. **Široko dostupan** – prevodilac za jezik treba da bude široko dostupan za sve glavne arhitekture i za sve operativne sisteme.

U tabeli 1.1, ilustracije radi, date su generalne ocene nekih programskih jezika, gde znak '+' označava prisustvo karakteristike u jeziku, znak '++' izuzetno prisustvo, '-' odsustvo, '- -' izuzetno odsustvo, a '/' označava da je za određenu karakteristiku teško reći u kojoj je meri zastupljena. Navedene karakteristike nisu jedine po kojima je moguće upoređivati programske jezike. Za konkretan slučaj upotrebe programskog jezika moguće je uvesti i nove kategorije.

	Maš.	Asem.	Pascal	C	C++	Java
Čitljivost	- -	-	+	+	+	+
Prenosivost	-	-	+	+	+	++
Opštost	+	+	/	+	+	+
Konciznost zapisa	-	-	/	+	+	+
Otklanjanje grešaka	- -	-	+	+	+	+
Cena	- -	-	+	+	+	++
Prepoznatljiva notacija	-	-	/	++	++	++
Brzo prevođenje	+	+	+	+	/	/
Efikasnost	++	++	/	+	/	- -
Modularnost	-	-	+	+	++	++
Široko dostupan	+	+	-	++	++	++

Tabela 1.1: Ocene karakteristika programskih jezika

U fazi implementacije projektovanog algoritma na računaru svaki algoritamski korak se zamenjuje nizom instrukcija konkretnog jezika. Svaka instrukcija mora biti napisana po određenim pravilima.

Definicija 1.3 (Sintaksa)**.** Sintaksa programskog jezika je skup pravila za pisanje instrukcija (naredbi) određenog jezika. Kaže se da je program *sintaksno ispravan* ukoliko je napisan strogo po pravilima jezika i da je *sintaksno neispravan*, odnosno sadrži sintaksne greške, ukoliko jedno ili više pravila sintakse nisu poštovana.

Da bi prevodilac mogao prevesti izvorni kod programa u izvršni program, izvorni kod programa mora biti sintaksno ispravan. Međutim, program može da bude sintaksno ispravan, a da prilikom izvršenja kao rezultat ne daje ono što je programer hteo zbog logičke greške (ili grešaka) u algoritmu.

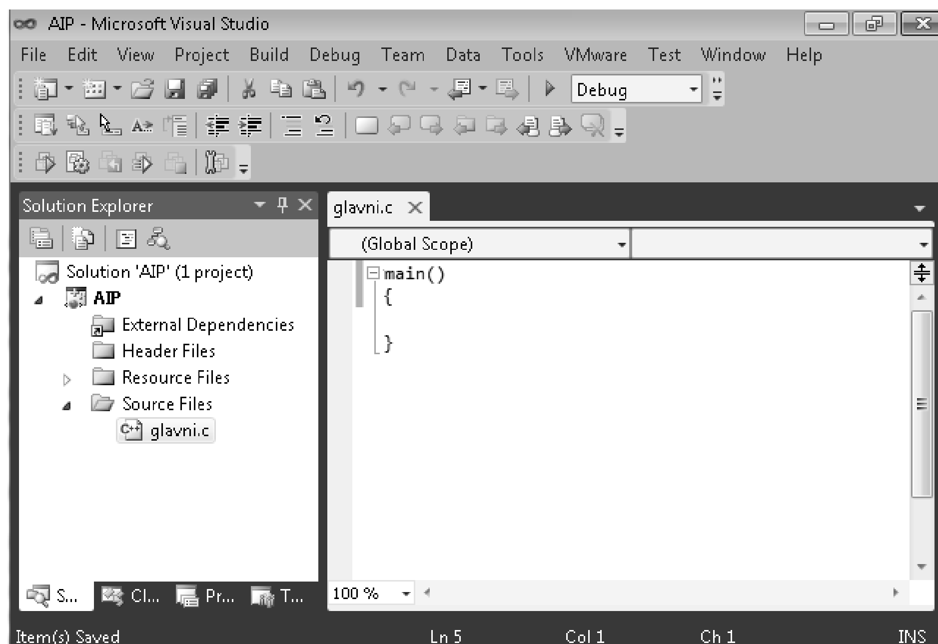
Definicija 1.4 (Semantika)**.** Semantika programa je očekivano dejstvo, t.j. logički smisao instrukcija programa posmatranih u celini.

Kaže se da je program semantički ispravan ukoliko daje očekivani rezultat.

Za konkretnu arhitekturu procesora i operativni sistem neophodno je postojanje kompajlera, a poželjno je da postoji i uglavnom je dostupno radno okruženje za implementaciju programa. Radno okruženje je jedinstvena aplikacija koja uključuje:

1. Editor - za pisanje koda,
2. Kompajler - za prevođenje, koji se direktno poziva iz okruženja i prevodi napisani kod,
3. Debuger - dodaci za analiziranje toka izvršenja programa, u cilju otklanjanja semantičkih grešaka.

Primeri razvojnih okruženja su *Microsoft Visual Studio* za jezike C, C++, C# i *Eclipse* za C, C++, Java programski jezik, PHP, itd. Izgled razvojnog okruženja *Microsoft Visual Studio* prikazan je na slici 1.10.



Slika 1.10: Izgled razvojnog okruženja *Microsoft Visual Studio*

Testiranje programa

Testiranje programa je podjednako važna faza u razvoju programa kao i samo pisanje programa. Može se reći da programeri veći deo svog radnog dana, ne računajući faze projektovanja, posvećuju testiranju i otklanjanju grešaka nego pisanju programa.

U skladu sa prethodnim definicijama, postoje dva tipa grešaka:

1. sintaksne greške, i
2. semantičke greške.

Kako bi kompajler mogao da niz instrukcija višeg programskog jezika automatski prevede u niz mašinskih instrukcija, instrukcije višeg programskog jezika moraju biti sintaksno ispravno napisane.

Kod interpreterskih programskih jezika program je obično moguće pokrenuti i izvršavati program čak i u slučaju da program sadrži sintaksne greške. Kada u toku izvršenja interpreterskih programskih jezika interpreter naiđe na sintaksnu grešku izvršenje se prekida i prikazuje se odgovarajuća poruka o grešci.

Primer 1.7. Svaki matematički ili logički izraz u programskom jeziku C neophodno je završiti znakom ';', kako bi kompajler razlikovao različite instrukcije. Sledeći kod je sintaksno ispravan:

```
x = 3 + 5; y = x + 2;
```

gde znak '=' predstavlja dodelu vrednosti promenljivoj na levoj strani. Sledeći kod sadrži sintaksnu grešku u vidu nedostajućeg simbola ';', pa kompajler ne može da prepozna kraj prvog izraza:

```
x = 3 + 5 y = x + 2;
```

△

Kompajler u jednoj od faza prevođenja programa vrši sintaksu analizu. On otkriva sintaksne greške u toku prevođenja programa i o tome obaveštava programera. Obaveštenje sadrži tačnu lokaciju greške (liniju koda i tačno mesto u liniji). Uz podatak o lokaciji greške kompajleri daju i informaciju o tome šta se po sintaksi jezika očekuje na tom mestu, a nije navedeno. Zbog ove funkcionalnosti kompajlera sintaksne greške se lako otklanjaju. Tek kada se otklone sve greške, kompajler može uspešno da generiše izvršni program.

Semantičke, odnosno logičke greške kompajler ne može otkriti. Semantički neispravan program može biti sintaksno ispravan, ali da prilikom izvršenja ne daje očekivane rezultate. Za otklanjanje ovakvih grešaka razvojna okruženja nude mogućnost debagiranja programa (eng. *debug* - otklanjanje grešaka).

Definicija 1.5 (Debagiranje programa)**.** Debagiranje programa je proces praćenje izvršenja programa naredba-po-naredbu (ili blok-po-blok), pri čemu se može pratiti sadržaj željenih memorijskih lokacija.

Program se u fazi debagiranja izvršava nad test skupom ulaznih podataka za koje se unapred zna koji će rezultat dati, kao i to koji će međurezultati u kojim memorijskim lokacijama biti prisutni u kom trenutku. Na ovaj način se može uočiti odstupanje od željenog toka izvršenja programa i intervenisati tako što će se ispraviti algoritam.

U nekim slučajevima je program dovoljno složen da sadrži i delove koji se retko izvršavaju. Naravno, postoji mogućnost pojave semantičkih grešaka i u takvim delovima programa.

Definicija 1.6 (Testiranje programa)**.** Testiranje programa je opsežan proces kreiranja skupa različitih ulaznih podataka sa unapred poznatim rezultatom izvršenja, takvih da se nakon izvršavanja programa za svaki iz skupa ulazih podataka sa sigurnošću može reći da program ne sadrži semantičke greške ni u jednom svom delu.

Veoma često je testiranje programa posebno radno mesto u kompanijama koje se profesionalno bave razvojem softvera i sistema. Kod aplikacija sa vizuelnim interfejsom, pojednostavljeno rečeno, u toku testiranja treba testirati sve scenarije kroz koje korisnik može proći, i proveriti da li sve funkcije softvera rade kako je predviđeno.

Formalni način određivanja optimalnog skupa test-primera pomoću kojih je moguće proći kroz sve delove i proveriti program/sistem u celosti je složen proces i neće biti razmatran u ovom kursu.

Pisanje dokumentacije

Dokumentovanje programa je važna faza u razvoju softvera. Program sa kompletnom dokumentacijom je moguće nazvati gotovim proizvodom.

Postoje dva glavna razloga zbog čega je potrebno sastaviti dokumentaciju. Prvi razlog je eventualna dorada programa od strane istog programera koji je razvio program. Bez obzira što je neko autor programa, u slučaju potrebe da se program doradi posle određenog vremena teško je brzo locirati delove koda koji se odnose na željenu funkciju. Takođe, može biti vremenski zahtevno i otkrivanje semantike željenog dela koda, odnosno na koji način je određeni problem rešen u datom trenutku. Drugi razlog je nastavak rada na programu i dorada funkcija programa od strane programera koji nije originalno učestvovao u pisanju programa. Ovo je čest slučaj u programerskim firmama zbog preraspodele posla ili promene pozicija/kompanija od strane programera. U ovom slučaju za programera koji se po prvi put susreće sa kodom pisanim od strane drugog programera prethodno navedeni problemi su još izraženiji, a postojanje dokumentacije znači brže uključivanje programera u problematiku.

Dokumentacija u velikoj meri ubrzava "snalaženje" po "tuđem" kodu, ili ponovni povratak na program posle određenog vremena radi dorade. Dokumentacija najčešće sadrži:

1. detaljan i precizan opis problema,
2. detaljan opis korišćenog metoda,
3. algoritam (algoritme) i/ili UML dijagrame,
4. komentare prožete kroz sam program,
5. sistemsko uputstvo,
6. korisničko uputstvo.

Kontrolna pitanja

1. Ko se smatra prvim programerom i na osnovu rada na kom uređaju?
2. Koji koncept nedostaje prvobitnim uređajima za računanje da bi se mogli smatrati računarima u današnjem smislu te reči?
3. Od čega se sastoji Tjuringova mašina?
4. Modifikovati prelaske stanja Tjuringove mašine iz primera 1.1, tako da se glava za čitanje simbola na kraju programa ne vraća na početak niza simbola.
5. Opisati osnovne elemente Von Nojmanove arhitekture. Na koji način se izvršavaju instrukcije na ovoj arhitekturi?
6. Koje kategorije mašinskih instrukcija postoje?
7. Koje su prednosti, a koje mane programiranja na asemblerskim jezicima?
8. Objasniti proces generisanja izvršnog koda iz programa napisanog na asemblerskom jeziku. Koji su dodatni koraci potrebni kod makro-asemblerskih jezika i zašto?
9. U čemu se ogleda razlika u izvršavanja programa napisanih na interpreterskim jezicima i kompajlerskim jezicima?
10. Koje su faze u rešavanju problema uz pomoć računara?
11. Odrediti vrednosti $\sqrt[2]{5}$, $\sqrt[2]{7}$ i $\sqrt[3]{7}$ korišćenjem iterativnog postupka za izračunavanje vrednosti funkcije $x = \sqrt[n]{a}$:

$$x_0 = \frac{(a+n-1)}{n}$$

$$x_{i+1} = \frac{\left((n-1)\cdot x_i + \frac{a}{x_i^{n-1}}\right)}{n}, \quad i = 0, 1, 2, \ldots.$$

   Kroz korake postupka objasniti primenu ovih postupaka kod ručnih kalkulatora.
12. Šta je sintaksa, a šta semantika programa?

2

Algoritmi

Algoritam je precizno definisani postupak, takav da praćenje koraka od početka pa do kraja algoritma vodi rešenju nekog konkretnog problema. Koraci algoritma su elementarne celine koje se precizno i eksplicitno navode, bez podrazumevanih delova.

Algoritmi se mogu predstaviti na različite načine. Načini za predstavljanje algoritama uključuju:

1. tekstualni opis na prirodnom jeziku,
2. dijagrame toka,
3. pseudokod,
4. strukturograme, i
5. programske jezike.

Algoritmi predstavljeni prirodnim jezikom koriste izraze koji imaju tendenciju da budu razumljiviji i nedvosmisleni. Ovakvo predstavljanje se retko koristi za složene i tehnički zahtevne algoritme. Dijagrami toka, pseudokod i strukturogrami su strukturirani načini predstavljanja algoritama kojima se izbegavaju mnoge nejasnoće i dvosmislenosti tipične za prirodne jezike. Programski jezici, s druge strane, su prvenstveno namenjeni za predstavljanje algoritama u obliku koji se može direktno izvršiti na računaru.

Pored navedenih postoje i drugi načini za predstavljanje algoritama. Jedan od načina koji nije među gore navedenim je predstavljanje algoritma pomoću Tjuringove mašine. Reprezentacija algoritma pomoću Tjuringove mašine uključuje skup simbola, stanja i definicije prelaza iz stanja u stanje, na način ilustrovan na primeru 1.1.

Po detaljnosti algoritma, moguće je izvršiti podelu na tri nivoa:

1. **Opisi na visokom nivou** – su opisi algoritama koji ne zalaze u detalje i u potpunosti ignorišu implementaciju. Na primeru Tjuringove mašine na ovom

nivou nije potrebno opisivati način na koji se glava kreće po traci, stanja mašine, prelaze, i sl, već samo princip rada.

2. **Implementacioni opis** – je opis čiji se koraci direktno mogu prevesti u instrukcije programskog jezika. U slučaju Tjuningove mašine ovakav opis sadrži precizan opis koraka mašine i pomeranja glave prilikom pamćenja podataka na traci.

3. **Formalni opis** – je opis predstavljen matematičkim jezikom.

Primer 2.1. Jedan od najjednostavnijih algoritama je algoritam za određivanje najvećeg broja u neuređenoj listi brojeva. Rešenje nužno nameće razmatranje svakog broja iz liste, po jednog u jednom trenutku. Iz ovoga proizlazi jednostavan algoritam, koji može biti opisan na različitim nivoima apstrakcije na neki od sledećih načina:

Opis na visokom nivou
1. Pretpostavimo da je prvi broj najveći.
2. Proveriti svaki od preostalih elemenata iz liste. Ako je vrednost elementa veća od poslednje zabeležene, pribeležiti je.
3. Poslednji broj koji je zabeležen nakon prolaza kroz celu listu je najveći broj.

Implementacioni opis
Ulaz: neprazna lista brojeva L.
1. Uzeti prvi broj u listi za najveći broj ($Max = L_1$).
2. Uporediti sledeći broj iz liste L sa onim za koji se u tom trenutku smatra da je maksimalan ($L_i > Max$).
3. Ako je trenutno posmatrani broj veći od onog za koji se smatra najvećim u tom trenutku, uzeti trenutni broj za najveći $Max = L_i$.
4. Da li je provereni broj poslednji u listi? Ako jeste, preći na korak 5, ako nije vratiti se na korak 2.
5. Izlaz: najveći broj.

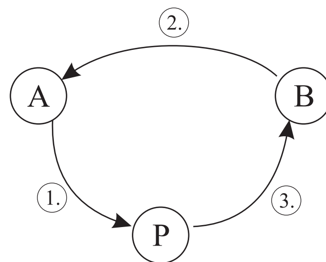
Formalni opis
$\mathbf{L} = L_i, i = 1, 2, \ldots, n,$
$\{\exists L_k \in \mathbf{L} \mid L_k > L_i, i = 1, 2, \ldots, k-1 \bigwedge L_k > L_i, i = k+1, k+2, \ldots, n\}$
$\Rightarrow max = L_k$
$\triangle$

2.1 Tekstualni opis algoritma

Jedan od načina za predstavljanje algoritama je tekstualni opis algoritma na govornom jeziku. Kod tekstualnog opisa algoritma postupak za rešavanje zadatog problema opisuje se preciznim rečenicama govornog jezika. Tekstualni opis na visokom nivou često predstavlja sam opis zadatka. Implementacioni opis sadrži rečenice koje opisuju algoritam korak po korak.

Primer 2.2 (Zamena mesta vrednosti dve lokacije)**. Zadatak:** Neka su A i B dve lokacije u koje je upisan po jedan broj. Zameniti mesta brojevima, tako da broj iz lokacije A bude upisan u lokaciju B, i obrnuto.

Rešenje: Ovaj problem je generalni problem zamene mesta sadržaja dveju lokacija, i sreće se i u realnom životu. Neka su lokacije A i B apstraktne lokacije u koje staje tačno po jedan broj, prirodno se nameće potreba za korišćenjem još jedne pomoćne lokacije. Nije moguće sadržaj lokacije A prebaciti na lokaciju B, dok je ova, uslovno rečeno, puna, kako ne bismo izgubili sadržaj lokacije B. Zato je potrebno prvo sadržaj jedne lokacije premestiti u pomoćnu lokaciju. Jedno moguće rešenje prikazano je na slici 2.1. Na slici je svako pomeranje vrednosti označeno strelicom uz naznaku rednog broj koraka u algoritmu. Tekstualni opis algoritma



Slika 2.1: Zamena mesta sadržaja dveju lokacija

moguće je predstaviti na sledeći način:

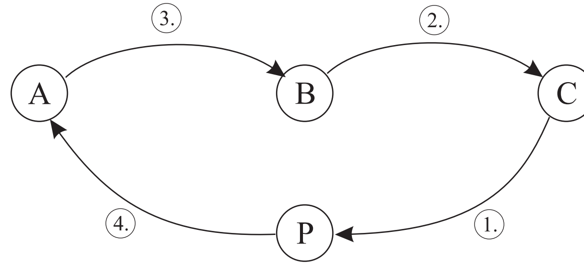
1. Sadržaj lokacije A pomeriti na lokaciju P.
2. Sadržaj lokacije B pomeriti na lokaciju A.
3. Sadržaj lokacije P pomeriti na lokaciju B.

△

Potrebno je naglasiti da je skoro svaki problem moguće rešiti na više različitih načina. Problem zamene lokacija iz prethodnog primera moguće je rešiti bar na još jedan način i to počev od pomeranja sadržaja lokacije B na lokaciju P u prvom koraku, umesto inicijalnog pomeranja sadržaja lokacija A.

Primer 2.3 (Ciklično pomeranje sadržaja tri lokacije za jedno mesto u desno)**. Zadatak:** Neka su A, B i C tri lokacije u koje je upisan po jedan broj. Ciklično pomeriti brojeve za jedno mesto u desno, tako da broj iz lokacije A bude pomeren na lokaciju B, broj iz lokacije B na lokaciju C i broj iz lokacije C na lokaciju A.

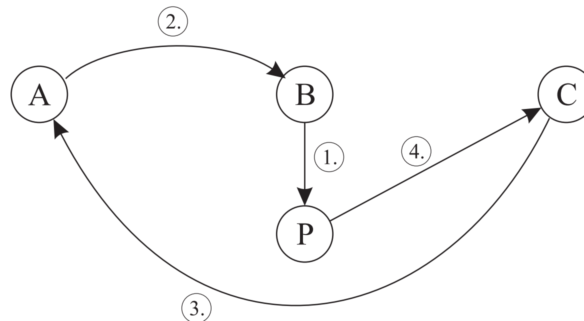
Rešenje: Ukoliko kao u prethodnom primeru uzmemo da su lokacije apstraktne lokacije u koje staje tačno po jedan broj, ponovo se nameće potreba za korišćenjem još jedne pomoćne lokacije. Jedno moguće rešenje prikazano je na slici 2.2. Na slici je svako pomeranje vrednosti označeno strelicom uz naznaku rednog broj koraka u



Slika 2.2: Ciklično pomeranje sadržaja tri lokacije za jedno mesto u desno

algoritmu. Tekstualni opis algoritma moguće je predstaviti na sledeći način: Sadržaj lokacije C pomeriti na lokaciju P; B pomeriti u C; A u B; P u A. △

Prethodno tvrđenje da postoji više mogućih rešenja problema ilustrovaćemo modifikacijom rešenja primera 2.3. Naime, izbor prvog koraka da sadržaj lokacije C prvo premesti u uslovno rečeno "praznu" lokaciju P je odluka programera. Algoritam ne mora početi ovako. Problem je moguće rešiti tako što se sadržaj bilo koje od tri lokacije na početku pomeri u pomoćnu lokaciju P. Rešenje kod koga se u prvom koraku na pomoćnu lokaciju P prebacuju sadržaj lokacije B ilustrovano je na slici 2.3.



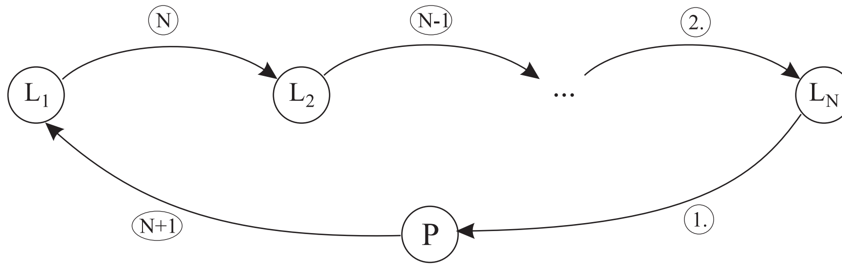
Slika 2.3: Ciklično pomeranje sadržaja tri lokacije za jedno mesto u desno, II način

Tekstualni opis algoritma je sledeći: Sadržaj lokacije B pomeriti na lokaciju P; A pomeriti u B; sadržaj lokacije C u A; na kraju sadržaj lokacije P pomeriti na lokaciju C.

Sledeći primer ilustruje opšti oblik pomeranja sadržaja niza lokacija za jedno mesto u desno.

Primer 2.4 (Ciklično pomeranje niza elemenata za jedno mesto u desno)**. Zadatak:** Neka su $L_i, i = 1, 2, \ldots, N$ elementi niza **L** koji predstavljaju N lokacija u koje je upisan po jedan broj. N je proizvoljan broj koji zadaje korisnik. Ciklično pomeriti brojeve za jedno mesto tako da vrednost lokacije L_1 bude pomerena na lokaciju L_2, vrednost lokacije L_2 u L_3, itd. Vrednost poslednje lokacije L_N pomeriti na prvu lokaciju L_1.

Rešenje: Rešenje je prikazano na slici 2.4. U prethodnim primerima broj lokacija



Slika 2.4: Ciklično pomeranje niza elemenata za jedno mesto u desno

bio je konstantan, dok u ovom primeru broj lokacija zavisi od parametra N. Tako, ako je N=5, biće 5 lokacija, ako je N=50 biće 50 lokacija, itd. Da je broj lokacija unapred poznat, rešenje bi bilo urađeno po istom principu kao i u prethodnim primerima. Odnosno, za slučaj da korisnik zada parametar N=5, algoritam bi bio:

1. sadržaj lokacije L_5 pomeriti na lokaciju P;
2. L_4 u L_5;
3. L_3 u L_4;
4. L_2 u L_3;
5. L_1 u L_2;
6. sadržaj lokacije P pomeriti u L_1.

Da korisnik zada parametar N=8, algoritam bi bio:

1. sadržaj lokacije L_8 pomeriti na lokaciju P;
2. L_7 u L_8;
3. L_6 u L_7;
4. L_5 u L_6;
5. L_4 u L_5;
6. L_3 u L_4;
7. L_2 u L_3;

8. L_1 u L_2;
9. sadržaj lokacije P pomeriti u L_1.

Iz prethodna dva navedena algoritma vidimo da je broj koraka u algoritmima različit. Ovo bi prilikom implementacije iziskivalo dva različita programa, jedan za vrednost N=5, a drugi za vrednost N=8. To znači da prethodna dva rešenja ne zadovoljavaju jednu od osnovnih osobina algoritama datih u prethodnom poglavlju: **masovnost**. Postavlja se pitanje da li je moguće napisati takav algoritam da **broj koraka algoritma ne zavisi od parametra N**? Ovakav algoritam bi imao osobinu masovnosti, bio bi uslovno rečeno "upotrebljiviji", odnosno jedan program bi rešavao klasu problema za bilo koje zadato N.

Problem je moguće rešiti ponavljanjem jednog koraka N puta, gde u koraku koji se ponavlja figurišu susedne lokacije L_i i L_{i+1}. U tu svrhu potrebno je uvesti dodatni brojač i za indeksiranje lokacije koju treba pomeriti. Algoritamsku strukturu koja višestruko ponavlja izvršenje određenog dela algoritma nazivamo petljom. Algoritam je sledeći:

1. Početak
2. Sadržaj lokacije L_N pomeriti na lokaciju P.
3. $i = N$
4. Umanjiti i za 1.
5. Sadržaj lokacije L_i pomeriti na lokaciju L_{i+1}.
6. Da li je $i = 1$? Ako jeste preći na korak 7, a ako nije vrati se na korak 4.
7. Sadržaj lokacije P pomeriti na lokaciju L_1.
8. Kraj

Prethodni algoritam je formalno moguće opisati na sledeći način:

1. $L_N \rightarrow P$
2. $L_i \rightarrow L_{i+1}$, i=N-1,N-2,...,1
3. $P \rightarrow L_1$

△

2.2 Dijagrami toka

Algoritmi se grafički mogu predstaviti dijagramima toka.

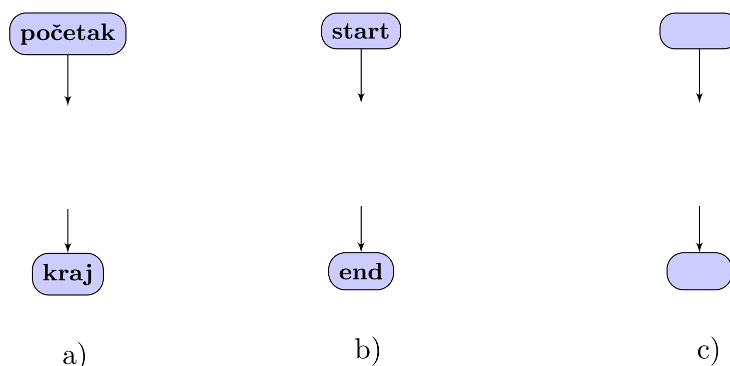
Definicija 2.1 (Dijagram toka algoritma)**.** Dijagram toka algoritma je grafička reprezentacija algoritma, sastavljena od niza grafičkih simbola (blokova), koji opisuju obradu podataka u datom trenutku, ili kontrolu toka izvršenja, povezanih putevima koji definišu smer izvršenja programa.

Postoje četiri tipa blokova koji se koriste za predstavljanje koraka algoritama:

1. Blokovi za označavanje početka i kraja
2. Blokovi za unos i prikaz rezultata
3. Blokovi obrade
4. Blokovi za kontrolu toka izvršenja programa.

2.2.1 Početak i kraj programa

Simboli za označavanje početka i kraja programa prikazani su na slici 2.5.



Slika 2.5: Blokovi za označavanje početka i kraja programa

Svaki algoritam ima tačno jedan blok za početak i tačno jedan blok za završetak. Ovi blokovi se obično predstavljaju ovalnim oblicima, kako je to označeno na slici 2.5, međutim, u literaturi je moguće naći i drugačije oblike za označavanje ovih blokova.

Tekst u blokovima za početak i kraj je relativna stvar govornog jezika, pa je tako moguće koristiti reči ”početak” i ”kraj”, ili ”start” i ”end”, kako je dato na slikama 2.5a i 2.5b, respektivno. Takođe je moguće koristiti i bilo koju drugu oznaku, ili ostaviti čvorove praznim, kao na slici 2.5c. Bez obzira na oblik i oznaku koja se nalazi u čvoru, ovi čvorovi se precizno mogu definisati na sledeći način.

Definicija 2.2 (Blok početka algoritma)**.** Blok početka algoritma je blok koji nema dolaznih grana i ima samo jednu odlaznu granu.

Definicija 2.3 (Blok kraja algoritma)**.** Blok kraja algoritma je blok koji nema odlaznih grana i ima samo jednu dolaznu granu.

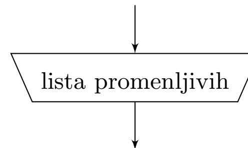
2.2.2 Unos i prikaz rezultata

Da bi program napisan za računar imao upotrebnu vrednost, program mora imati osobinu rezultativnosti i masovnosti. Kao što je u prethodnom poglavlju rečeno, rezultativnost znači da program daje rezultat koji korisnik očekuje. Masovnost znači da program rešava problem za klasu zadatih podataka, a ne samo za određeni partikularni slučaj. Programi napisani za partikularne slučajeve daju istu informaciju nakon svakog izvršenja, pa samim tim i nemaju upotrebnu vrednost.

Primer 2.5 (Masovnost)**.** Za primer algoritma koji ne ispunjava osobinu masovnosti može se uzeti algoritam, odnosno program koji sabira, na primer, brojeve 2 i 3. Nakon prvog izvršenja program će prikazati zbir brojeva 2+3=5, kao i nakon svakog sledećeg izvršenja, pa se postavlja pitanje svrhe ovakvog programa.

Za primer algoritma koji ispunjava osobinu masovnosti može se uzeti algoritam, odnosno program koji sabira vrednosti promenjljivih a i b, čije vrednosti na početku programa zadaje korisnik. Ovakav program je upotrebljiviji, jer ispunjava osobinu masovnosti, odnosno rešava problem sabiranja za bilo koja dva zadata broja. △

Da bi se obezbedila masovnost, potrebno je obezbediti mogućnost ulaza u algoritam, odnosno mogućnost zadavanja vrednosti. Za ulaz podataka u algoritam koristi se blok sa slike 2.6.



Slika 2.6: Ulazni blok

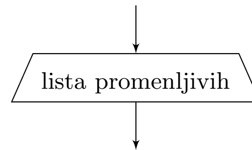
Potrebno je napomenuti da blok za ulaz označava ulaz podatka bilo kog tipa u algoritam, sa bilo kog ulaznog medijuma. Ulaz podataka se predstavlja istim blokom bez obzira da li se radi o unošenju vrednosti sa tastature, učitavanju iz fajla sa diska računara, ili očitavanju podataka sa nekog senzora, i sl.

Definicija 2.4 (Promenljiva)**.** Promenljiva u kontekstu programiranja je simboličko ime za memorijsku lokaciju u kojoj je moguće pamtiti vrednosti.

Dijagram toka ne razlikuje nužno različite medijume. Tek prilikom implementacije algoritma na konkretnom programskom jeziku, blok za unos vrednosti u promenljivu se zamenjuje željenim skupom naredbi za učitavanje sa medijuma koji programski jezik podržava (tastatura, fajl, port računara, itd). Ukoliko je neophodno za bolje razumevanje algoritma, dodatnim tekstualnim opisom u bloku unosa može se pojasniti sa kod medijuma se podaci unose.

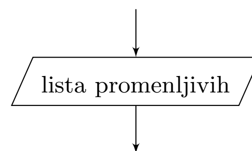
Izlazni blok prikazan je na slici 2.7. Kao i kod ulaznog bloka, izlazni blok ne sadrži nužno oznaku medijuma na koji se šalju podaci. To je problem samog

programskog jezika koji se koristi za implementaciju. U okviru bloka se redom navode podaci, odvojeni zarezom, koje je potrebno izvesti iz algoritma. Izlaz može biti na monitoru računara, štampaču, upis u fajl na disku, ili sl.



Slika 2.7: Izlazni blok

Umesto blokova za ulaz i izlaz datih na slikama 2.6 i 2.7, kao univerzalni ulazno/izlazni blok u literaturi se sreće i blok prikazan na slici 2.8. U ovom slučaju informaciju o tome da li se radi o ulazu ili izlazu potrebno je eksplicitno naznačiti tekstom u okviru bloka, ili se može izostaviti, ako se iz konteksta okolnih blokova može nedvosmisleno zaključiti o kom se tipu bloka radi.

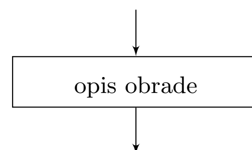


Slika 2.8: Univerzalni ulazno-izlazni blok

Blokovi za ulaz i izlaz imaju jednu dolaznu i jednu odlaznu granu.

2.2.3 Obrada podataka

Blok za obradu podataka prikazan je na slici 2.9.

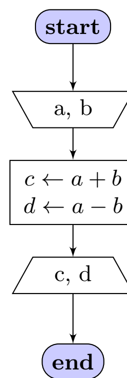


Slika 2.9: Blok za obradu podataka

Blok obrade takođe ima jednu ulaznu i jednu izlaznu granu. Blok obrade može da sadrži tekstualni opis obrade na različitim nivoima detaljnosti, jedan ili više matematičkih izraza napisanih tako da je svaki izraz u novom redu, ili pozive funkcija, o čemu će biti reči kasnije.

Primer 2.6. Zadatak: Nacrtati dijagram toka algoritma koji određuje i prikazuje zbir i razliku brojeva koje zadaje korisnik.

Rešenje: Tekstom se algoritam može opisati na sledeći način: korisnik zadaje (unosi) vrednosti promenljivih; nazovimo te promenljive a i b; vrednosti promenljivih a i b sabrati i upisati ih u promenljivu c ($a+b \rightarrow c$); vrednosti promenljivih a i b oduzeti i upisati ih u promenljivu d ($a-b \rightarrow d$); prikazati vrednosti promenljivih c i d. Odgovarajući dijagram toka prikazan je na slici 2.10.



Slika 2.10: Dijagram toka algoritma za sabiranje i oduzimanje brojeva

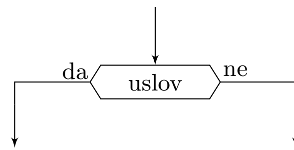
Napomena: Zbog opštosti dijagrama toka algoritma, u ovom primeru je za dodelu vrednosti promenljivoj korišćen simbol $\rightarrow$. Kod programskih jezika najčešće se sreću operatori '=', ili ':='. U C-u se za dodelu vrednosti promenljivoj koristi operator '=', gde je promenljiva kojoj se dodeljuje vrednost levi operand, a desni operand može biti konstanta, promenljiva, ili izraz.

△

2.2.4 Kontrola toka izvršenja programa

Za kontrolu toka izvršenja programa koriste se blokovi grananja (alternacije). Blok za grananja prikazan je na slici 2.11.

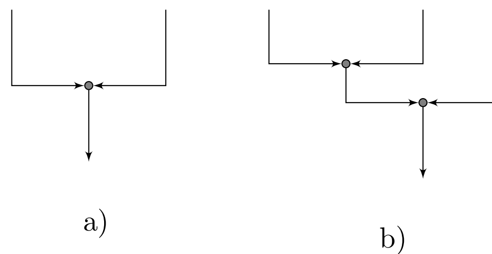
Blokovi grananja predstavljaju trenutke odluke u algoritmu. Blok grananja ima jednu ulaznu i dve izlazne grane. Sastoji se od logičkog uslova, u zavisnosti od čije vrednosti izvršenje može nastaviti jednim od dva ponuđena puta. Jedna od grana



Slika 2.11: Blok grananja

nosi oznaku "da" i ovom granom će se nastaviti izvršenje, ukoliko je uslov ispunjen. Druga grana nosi oznaku "ne" i njom će se izvršenje nastaviti, u slučaju da uslov nije ispunjen.

Blok grananja za jednu ulaznu granu daje dve izlazne grane (slika 2.11). Kako svaki algoritam ima jedan početni i jedan završni čvor, to se sve grane na kraju moraju na neki način spojiti i završiti u tom jednom završnom čvoru. Za spajanje različitih puteva koristi se čvor prikazan na slici 2.12.



a) b)

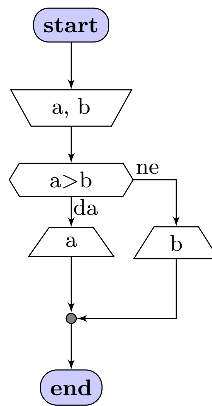
Slika 2.12: Čvor za spajanje potega

Primer 2.7. Zadatak: Nacrtati dijagram toka algoritma koji određuje i prikazuje koji je broj, od dva zadata broja koje zadaje korisnik, veći. Smatrati da su brojevi koje zadaje korisnik različiti.

Rešenje: Dijagram toka algoritma prikazan je na slici 2.13. △

2.3 Osnovne i složene algoritamske strukture

Skup osnovnih blokova dijagrama toka algoritma, povezanih na određeni način u celinu, čini algoritamsku strukturu. Da bi algoritam bilo moguće implementirati na programskom jeziku, blokovi se moraju povezivati na strogo definisani način.



Slika 2.13: Primer određivanja većeg od dva broja

Načini na koje se mogu povezati blokovi definisani su dostupnim strukturama u programskim jezicima.

Definicija 2.5 (Algoritamske strukture)**.** Pravila povezivanja blokova dijagrama toka algoritma granama nazivaju se algoritamske strukture.

Algoritamske strukture uglavnom su definisane time šta je u određenom programskom jeziku moguće implementirati direktnim preslikavanjem algoritma u kod. Algoritamske strukture se u određenoj meri mogu razlikovati po tome što neki jezici mogu imati dodatne strukture, koje nisu podržane u drugim jezicima. Algoritamske strukture se mogu podeliti na

1. osnovne, i
2. složene algoritamske strukture.

Definicija 2.6 (Osnovne algoritamske strukture)**.** Osnovne algoritamske strukture su elementarne funkcionalne celine koje odgovaraju elementarnim funkcionalnim celinama programskog jezika.

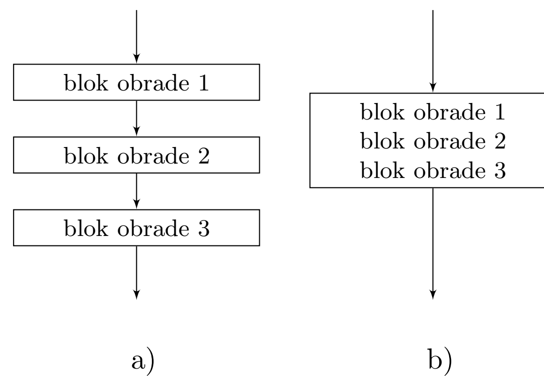
Osnovne algoritamske strukture se mogu implementirati elementarnim strukturama bilo kog programskog jezika. Algoritam sastavljen pravilnim kombinovanjem osnovnih algoritamskih struktura moguće je implementirati na bilo kom programskom jeziku. U osnovne algoritamske strukture spadaju:

1. sekvenca,
2. alternacija, i
3. petlje.

Složene algoritamske strukture dobijaju se pravilnim kombinovanjem osnovnih struktura. Pravilnim kombinovanjem osnovnih algoritamskih struktura moguće je predstaviti bilo koji algoritam.

2.3.1 Sekvenca

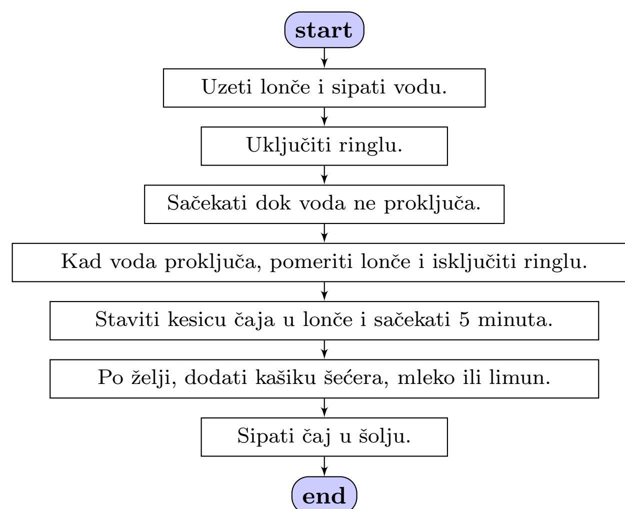
Sekvenca je osnovna algoritamska struktura koja se dobija kaskadnim nadovezivanjem blokova obrade. Dobija se tako što se izlazni poteg prvog bloka vezuje na ulazni poteg drugog bloka, izlazni poteg drugog na ulazni trećeg, itd. Sekvenca blokova obrade prikazana je na slici 2.14.



Slika 2.14: Algoritamska struktura sekvenca

Više sukcesivnih blokova obrade u sekvenci moguće je zameniti jednim blokom na način prikazan na slici 2.14b.

Primer 2.8. Algoritam za kuvanje čaja iz primera 1.5 moguće je predstaviti dijagramom toka, korišćenjem strukture sekvence na sledeći način:



△

2.3.2 Alternacija

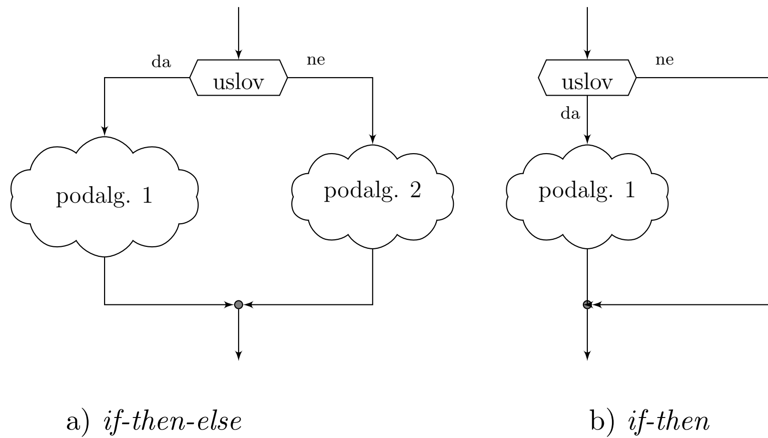
Alternacija, ili grananje je osnovna algoritamska struktura kod koje se u zavisnosti od zadatog uslova izvršava jedan od dva podalgoritma. Grananje je prikazano na slici 2.15. Svi programski jezici imaju naredbe kojima se struktura prikazana na slici može implementirati.

U zavisnosti od zadatog uslova izvršenje algoritma, nakon bloka alternacije može se nastaviti u dva pravca. Jedan pravac je za slučaj da je uslov ispunjen, a drugi pravac je za slučaj da ulslov nije ispunjen, što je prikazano na slici 2.15.

Definicija 2.7 (Grane alternacije)**.** ”Da” granom algoritamske strukture alternacije naziva se podalgoritam na putu, odnosno grani koja se izvršava ako je uslov grananja ispunjen. ”Ne” granom algoritamske strukture alternacije naziva se podalgoritam, odnosno grana na putu koji se izvršava ako uslov grananja nije ispunjen.

If-then-else algoritamska struktura, čiji naziv potiče od engleskih reči *if-then-else*, što u prevodu znači *ako-onda-u suprotnom*, se sastoji od: uslova grananja, podalgoritma u ”da” grani i podalgoritma u ”ne” grani, kao što je prikazano na slici 2.15a.

Potrebno je napomenuti da se kod ove strukture putevi nakon završetka podalgoritama iz ”da” i iz ”ne” grane obavezno spajaju u jednoj tački. To znači da se izvršenje nastavlja počev od iste naredbe nakon strukture alternacije bez obzira na to kojim se putem krenulo u zavisnosti od uslova grananja. Ukoliko se grane ne spajaju u jenoj tački, kaže se da algoritam nije **strukturni** i takav algoritam



a) *if-then-else* b) *if-then*

Slika 2.15: Algoritamska struktura alternacija: a) tip *if-then-else*, b) tip *if-then*

nije moguće direktno implementirati na nekom programskom jeziku korišćenjem naredbi grananja.

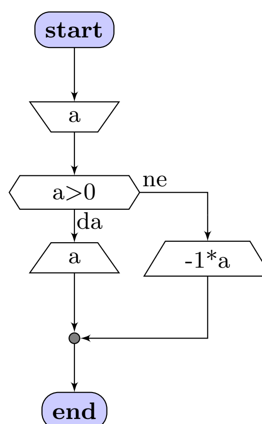
If-then algoritamska struktura je struktura koja nema drugih blokova u "ne" grani (slika 2.15b). Ovu strukturu je rečima moguće opisati kao strukturu koja "preskače" deo algoritma i ne izvršava ga ako zadati uslov nije ispunjen. Generalno, treba napomenuti da je "preskakanje" dela algoritma u slučaju neispunjenja uslova relativno, jer je ovo lako promeniti u "preskakanje" u slučaju ispunjenja uslova negacijom uslova.

Primer 2.9. Zadatak: Nacrtati strukturni dijagram toka algoritma koji određuje i prikazuje apsolutnu vrednost broja koji zadaje korisnik.

Rešenje: Apsolutna vrednost broja je vrednost broja napisana bez znaka. Neka korisnik zadaje vrednost broja koja se pamti u promenljivoj sa simboličkim imenom a. Ukoliko je broj pozitivan, tada je apsolutna vrednost broja sam broj a, pa ga kao takvog i treba prikazati. Ukoliko je vrednost broja negativna, treba se "osloboditi" znaka. Da bi se oslobodili predznaka '-', broj se može pomnožiti sa -1. Strukturni dijagram toka ovog algoritma prikazan je na slici 2.16. △

2.3.3 Petlje

Petlje su osnovne algoritamske strukture koje omogućavaju višestruko izvršavanje delova algoritma.



Slika 2.16: Strukturni dijagram toka algoritma za određivanje apsolutne vrednosti zadatog broja

Definicija 2.8 (Telo petlje)**.** Telom petlje naziva se podalgoritam koji je obuhvaćen petljom i koji se u okviru petlje višestruko izvršava.

Definicija 2.9 (Iteracija petlje)**.** Jedno izvršenje tela petlje naziva se iteracijom petlje.

Petlje u velikoj meri omogućavaju postizanje osobine masovnosti kod algoritma, kako što je ilustrovano u primeru 2.4. Broj ponavljanja tela petlje ne mora biti poznat u toku projektovanja algoritma i može da zavisi od parametara koje zadaje korisnik ili parametara koji se mogu menjati u toku izvršenja.

U zavisnosti od toga da li je broj ponavljanja tela petlje poznat pre samog početka petlje, razlikujemo dva tipa petlji:

1. bezuslovne petlje, i
2. uslovne petlje.

Definicija 2.10 (Bezuslovne petlje)**.** Bezuslovne petlje su petlje koje ponavljaju telo petlje tačno određeni broj puta, koji je poznat neposredno pre početka izvršenja petlje.

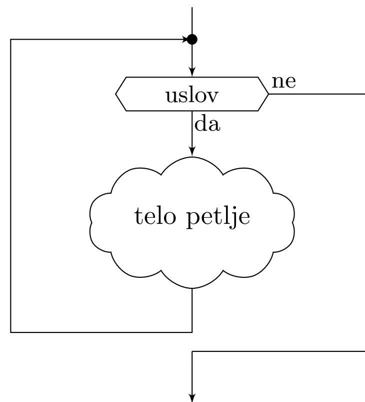
Broj iteracija bezuslovne petlje je zadat ili konstantom, ili parametrom čija vrednost mora biti poznata pre prve iteracije petlje.

Definicija 2.11 (Uslovne petlje)**.** Uslovne petlje su petlje kod kojih broj ponavljanja tela petlje nije poznat pre početka petlje. Telo petlje se ponavlja nakon završetka jedne iteracije, ili se petlja prekida, u zavisnosti od zadatog uslova.

U kategoriji bezuslovnih petlji razlikujemo petlju tipa *for*, dok u kategoriji uslovnih petlji razlikujemo petlje tipa *while-do* i *do-while*.

Petlja tipa *while-do*

Petlja tipa *while-do*, u prevodu sa engleskog *"sve dok je - radi"*, je osnovna algoritamska struktura kojom se implementira uslovna petlja, kod koje se telo petlje višestruko izvršava, kako joj i ime kaže, sve dok je uslov za izvršavanje tela petlje ispunjen. Ova petlja se kraće naziva i *while* petlja. Dijagram toka ove strukture prikazan je na slici 2.17.



Slika 2.17: Petlja tipa *while*

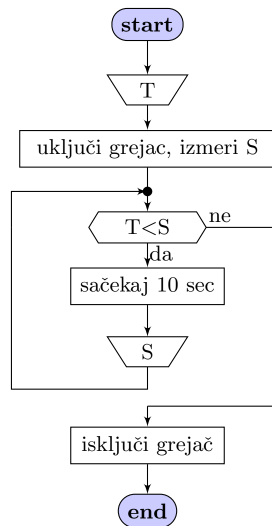
Kada se u toku izvršenja algoritma naiđe na *while* petlju, proverava se uslov za izvršenje tela petlje. Svi parametri koji figurišu u uslovu moraju biti poznati u tom trenutku, inače je algoritam neodređen i neprecizan. Ukoliko je uslov ispunjen izvršava se telo petlje. Nakon izvršenja tela petlje izvršenje se vraća se na trenutak neposredno pre provere uslova (slika 2.17).

Napomena: Povratna grana iz tela petlje nazad na uslov je uvek prazna (ne sadrži nijedan blok) i crta se sa leve strane tela petlje, kako je prikazano na slici 2.17. Praćenjem ove povratne grane izvršenje se vraća neposredno trenutak pre ispitivanja uslova (slika 2.17). Između tog trenutka (crna tačka iznad uslova na slici 2.17) i samog uslova takođe ne sme biti drugih blokova.

U telu petlje se mogu menjati parametri uslova. Kada se iteracija petlje završi uslov petlje se proverava ponovo. Ako uslov nije ispunjen petlja se završava, čime se prelazi na izvršenje prvog narednog bloka iza petlje.

Primer 2.10 (Algoritam regulatora za grejanje vode)**. Zadatak**: Nacrtati strukturni dijagram toka algoritma po kome radi regulator za grejanje vode. Regulator ima potenciometar za zadavanje željene temperature (T) i senzor za merenje trenutne temperature vode (S). Grejač vode treba da bude uključen sve dok se ne postigne željena temperatura vode. Kada je željena temperatura vode postignuta grejač treba isključiti.

Rešenje: Dijagram toka algoritma regulatora opisanog zadatkom prikazan je na slici 2.18.



Slika 2.18: Dijagram toka algoritma regulatora za grejanje vode

Na slici 2.18 dato je jedno moguće rešenje kod koga je kao pretpostavka uzeto da je voda hladna, t.j. da je početna temperatura obavezno manja od zadate. Zbog toga, kod ovog rešenja odmah nakon što korisnik zada željenu temeraturu T, grejač se uključuje. Kako svi parametri uslova moraju biti poznati pre petlje, uzima se i merena temperatura sa senzora S. Ukoliko je željena temperatura manja od merene, čeka se određeni vremenski period, pa se stanje senzora S ponovo očitava. Nakon očitavanja vrednosti sa senzora proverava se ponovo uslov, i tako redom sve dok uslov da je željenja temperatura vode manja od zadate ne prestane da važi. Kada se ovo desi algoritam izlazi iz petlje i isključuje grejač.

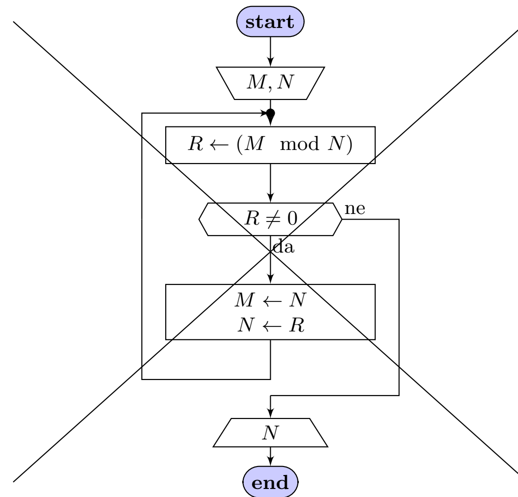
Potrebno je napomenuti da na osnovu algoritma nije moguće ustanoviti koliko vremena će biti potrebno za zagrevanje vode. Broj prolazaka kroz petlju nije moguće unapred odrediti, jer zavisi od mnogo faktora, kao što su inicijalna temperatura vode, spoljna temperatura, itd. Jedino se može reći, što je i karakteristika ovog tipa petlji, da će se izvršiti onoliko puta koliko je potrebno da bi se zagrejala voda.

Da bi ovaj algoritam bilo moguće implementirati, uređaj mora imati "naredbu", ili u opštem slučaju mehanizam za uključivanje i isključivanje grejača, kao i "naredbe" za očitavanje zadate temperature i temperature senzora. △

Primer 2.11 (Dijagram toka Euklidovog algoritma)**. Zadatak**: Nacrtati strukturni dijagram toka Euklidovog algoritma, opisanog u primeru 1.6.

Rešenje: Po algoritmu iz primera 1.6 prvi korak posle zadavanja brojeva je računanje ostatka, nakon čega se proverava uslov i vrši pomeranje N u M i R u N. Nakon ovog pomeranja algoritam iz primera 1.6 vraća se nazad na korak gde se određuje ostatak. Direktna grafička reprezentacija Euklidovog algoritma tekstualno opisanog u primeru 1.6 prikazana je na slici 2.19.

Dijagram toka algoritma sa slike 2.19 ne odgovara osnovnoj algoritamskoj strukturi *while* petlje sa slike 2.17 zato što se povratna grana ne vraća nazad u tačku neposredno pre uslova. Ovakav algoritam nije moguće implementirati na programskom jeziku osnovnim strukturama.

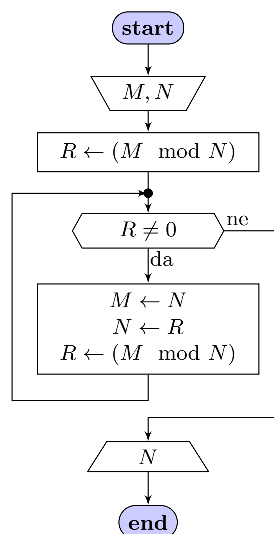


Slika 2.19: Nestrukturni dijagram toka Euklidovog algoritma

Da bi se rešio ovaj problem i dijagram toka sveo na osnovnu algoritamsku strukturu neophodno je modifikovati algoritam. Svakako se nakon pomeranja N u M i R u N mora odrediti novi ostatak (slika 2.19), da bi se parametar uslova R, koji se verovatno u međuvremenu promenio, ponovo odredio. Najjednostavnije rešenje za transformaciju ovog algoritma u strukturni je "izbacivanje" spornog koraka iz dela između povratne tačke i uslova, ali i njegovo umetanje na kraj tela petlje, zbog očuvanja redosleda događaja (slika 2.20). Time je sled događaja opisanih Euklidovim algoritmom održan, a struktura je zadovoljena. Strukturni dijagram toka Euklidovog algoritma prikazan je na slici 2.20.

$\triangle$

Napomena: Kod petlje tipa *while*, uslov za izvršenja tela petlje se nalazi ispred tela petlje, pa je minimalni broj izvršenja tela petlje 0. Petlja se neće nijednom izvršiti ako uslov u startu nije ispunjen.



Slika 2.20: Strukturni dijagram toka Euklidovog algoritma

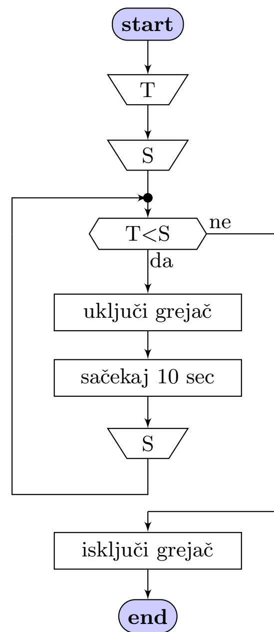
Primer 2.12 (Algoritam regulatora za grejanje vode, II način)**. Zadatak**: Nacrtati strukturni dijagram toka algoritma po kome radi regulator za grejanje vode, uz pretpostavku da temperatura vode inicijalno može biti i veća od zadate. Regulator ima potenciometar za zadavanje željene temperature (T) i senzor za merenje trenutne temperature vode (S).

Rešenje: Dijagram toka algoritma regulatora opisanog zadatkom prikazan je na slici 2.21.

Algoritam sa slike 2.21 na početku zahteva od korisnika da zada temperaturu T i vrši učitavanje trenutne vrednosti temperature sa senzora S. Za razliku od algoritma iz primera 2.10, algoritam u ovom trenutku ne uključuje grejač. Ukoliko je izmerena temperatura S veća od zadate, voda je već zagrejana, pa se petlja preskače. U ovom slučaju broj iteracija petlje je 0. Ukoliko je uslov za izvršenje tela petlje ispunjen, grejač će se uključiti, sačekaće se određeni vremenski period, pa će se ponovo očitati vrednost sa senzora S, i tako sve dok se ne postigne željena temperatura, kada se isključuje grejač.

Pretpostavka koja je učinjena u ovom rešenju je da je električni sklop takav da ukoliko se ponovo uključuje grejač koji je već uključen, grejač ostaje uključen, i obrnuto, ukoliko se isključuje već isključeni grejač, grejač će ostati isključen.
△

Da bi se telo petlje izvršilo bar jednom, uslov pre početka izvršenja petlje mora biti ispunjen.



Slika 2.21: Dijagram toka algoritma regulatora za grejanje vode, II način

Primer 2.13 (Iterativni postupak)**. Zadatak**: Nacrtati strukturni dijagram toka algoritma i na programskom jeziku C napisati strukturni program koji određuje i prikazuje vrednost $x = \sqrt[n]{a}$, za $a > 0$, primenom iterativnog postupka iz primera 1.4 (strana 18):

$$
\begin{aligned}
x_0 &= \frac{(a+n-1)}{n} \\
x_{i+1} &= \frac{\left((n-1)\cdot x_i + \frac{a}{x_i^{n-1}}\right)}{n},
\end{aligned}
\tag{2.1}
$$

gde je $i = 0, 1, 2, \ldots$ Izračunavanje prekinuti kada je $|x_{i+1} - x_i| \leq \epsilon$, gde je ϵ zadata tačnost.

Rešenje: Dati iterativni postupak, kako je u primeru 1.4 pokazano, počinje izračunavanje određivanjem vrednosti x_0 iz izraza (2.1). Nakon određivanja ove vrednosti, vrednost x_0 se koristi da bi se odredila vrednost x_1, koja je nešto bliža rešenju od x_0. Tada se na osnovu x_1 određuje x_2, itd. Generalno, kako je u jednačini (2.1) dato, na osnovu x_i određuje se x_{i+1}.

Problem je u tome što broj izračunavanja nije unapred poznat. Svaki naredni x_{i+1} je malo bliži traženom rešenju n-tog korena iz zadatka od x_i. Kako broj

međurezultata unapred nije poznat, glavno pitanje koje se nameće je koliko promenljivih treba za pamćenje međurezultata?

Ovo je ujedno i jedno od ključnih pitanja na koje treba odgovoriti u svakom algoritmu, mada je kod ovog problema, za razliku od primera obrađenih do sada, problem izraženiji i netrivijalan. Do odgovora na pitanje ćemo doći analizirajući sam postupak. Dakle, na osnovu x_0 određujemo x_1. Kada odredimo x_1, vrednost x_0 više nije potrebno pamtiti. Dalje, na osnovu x_1 određujemo x_2. Kada odredimo x_2, vrednost x_1 više nije potrebno pamtiti, itd.

Na osnovu prethodnog izlaganja može se zaključiti da su za implementaciju ovog iterativnog postupka neophodne samo 2 promenljive: x-prethodno i x-naredno. Daćemo im skraćena imena xp i xn.

Broj iteracija nije poznat, ali kako se približavamo tačnom rešenju razlika $xn - xp$ postaje sve manja i manja. Ovu vrednost treba uzeti kao apsolutnu vrednost jer je pitanje šta je od xn i xp veće. Moguće je da iterativni postupak osciluje oko tačnog rešenja. Kako je u tekstu zadatka rečeno, tačnost izračunavanja, što je ekvivalentno pomenutoj razlici, je $|x_{i+1} - x_i|$.

Evidentno je da se radi o *while* petlji koja se izvršava sve dok je razlika susednih međurezultata veća od zadate tačnosti ϵ (ne manja, jer kada je manja to je znak da je izračunavanje obavljeno). Na primer, da bi dobio rezultat sa tačnošću na tri decimale, korisnik će zadati vrednost $\epsilon = 10^{-4}$.

Da bi bilo moguće po prvi put ući u *while* petlju, treba obezbediti prethodno da su sve promenljive iz uslova petlje poznate. Dijagram toka ovog algoritma dat je na slici 2.22.

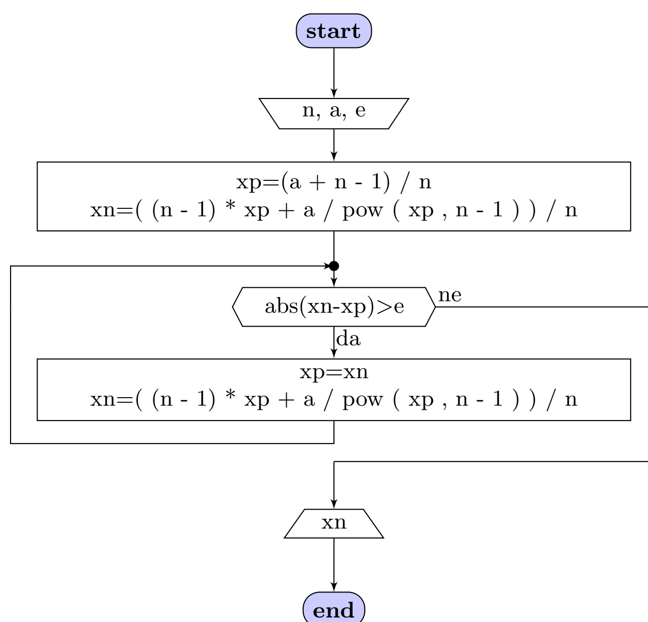
Kako nam nakon određivanja xn vrednost xp više nije potrebna, ovu promenljivu koristimo da u nju upišemo novoizračunatu vrednost xn, da bi na osnovu nje odredili naredno xn. △

Da bi se petlja okončala, parametri uslova petlje se u toku iteracija moraju promeniti, kako bi uslov prestao da važi. Promena parametara uslova u telu petlje je ključna za okončanje petlje.

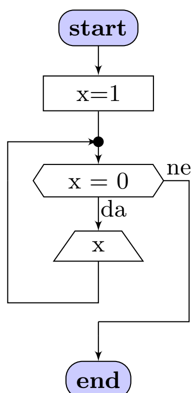
Napomena: *While* petlja kod koje je uslov inicijalno ispunjen, a parametri uslova se ne menjaju u toku izvršenja ponavlja telo petlje beskonačno puta. Ovakav program ispoljava efekat "blokade" funkcija, jer se nikada ne izlazi iz ovakve petlje.

Definicija 2.12 (Mrtva petlja)**.** Mrtvom petljom naziva se uslovna petlja kod koje uslov ostaje ispunjen nakon svake iteracije tela petlje.

Primer 2.14 (Mrtva petlja)**.** Na slici 2.23 prikazan je dijagram toka algoritma koji sadrži mrtvu petlju. Nakon postavljanja vrednosti x na 1 proverava se uslov da li je $x = 0$. Telo petlje prikazuje vrednost x. S obzirom da se u telu petlje parametar uslova x ne menja, ovaj algoritam prikazuje vrednost 1 beskonačno mnogo puta. Implementacija ovog algoritma na programskom jeziku bi proizvela program koji prikazuje vrednost 1 sve dok se nasilno ne prekine. △



Slika 2.22: Dijagram toka algoritma regulatora za grejanje vode, II način

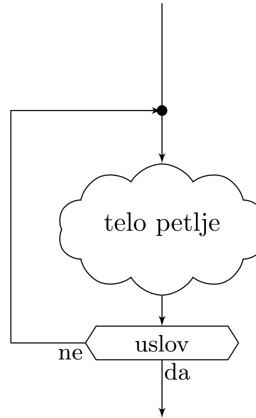


Slika 2.23: Ilustracija mrtve petlje

Petlja tipa *do-while*

Petlja tipa *do-while*, u prevodu sa engleskog *"radi - sve dok je"*, je osnovna algoritamska struktura kojom se implementira uslovna petlja koja iterira u zavisnosti od

zadatog uslova, kao i *while-do* petlja. Osnovna razlika je u tome što je kod *while-do* petlje uslov na početku, a kod *do-while* petlje uslov je na kraju petlje. Dijagram toka ove strukture prikazan je na slici 2.24.



Slika 2.24: Petlja tipa *repeat-until*

Razlika je i u tome što se kod ovog tipa petlje telo petlje ponavlja sve dok se uslov ne ispuni, odnosno sve dok za uslov važi da nije ispunjen. U literaturi je moguće naći i obrnute oznake grana, što uglavnom zavisi od ciljnog programskog jezika. Bez obzira na to kako su orjentisane grane "da" i "ne" koje izviru iz uslova, potrebno je jasno naznačiti "da" i "ne" granu, kako ne bi došlo do zabune. Ova petlja se drugačije naziva i *repeat-until* petlja (*prev. ponavljaj-dok*).

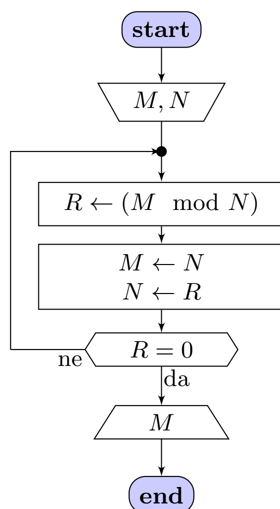
Postavljanje uslova na kraj petlje ima za posledicu to da se telo petlje mora izvršiti minimalno jednom (vidi sliku 2.24), za razliku od *while* petlje, gde je minimalni broj izvršenja petlje 0. Kod *repeat-until* petlje prvo se izvršava telo petlje, pa se nakon toga proverava uslov za kraj. Ako je uslov ispunjen posle prve iteracije, petlja će se završiti, ako ne, izvršenje će se vratiti na početak petlje, itd.

Napomena: Povratna grana iz uslova petlje nazad na tačku neposredno pre početka tela petlje je uvek prazna (ne sadrži nijedan blok) i obično se crta sa leve strane tela petlje, kako je prikazano na slici 2.24.

U telu petlje mogu se menjati parametri uslova, kao i kod *while* petlje, što je neophodno za uspešan završetak petlje i prelazak na blokove koji se nalaze iza petlje.

Primer 2.15 (Dijagram toka Euklidovog algoritma korišćenjem *do-while* petlje)**.** Na osnovu Euklidovog algoritma tekstualno opisanog u primeru 1.6, prirodno se nameće korišćenje *repeat-until* strukture za grafičku reprezentaciju ovog algoritma pomoću dijagrama toka, jer se izbegava dodavanje još jednog koraka za određivanje

ostatka, kao što je to bio slučaj u primeru 2.11. Strukturni dijagram toka Euklidovog algoritma prikazan je na slici 2.25.

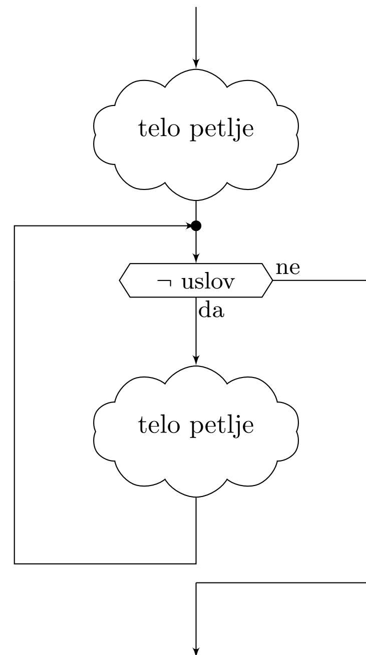


Slika 2.25: Dijagram toka Euklidovog algoritma implementiran petljom tipa *do-while*

Da bi do kraja bili dosledni, potrebno je reći da je algoritam iz primera 1.6 neznatno izmenjen da bi u potpunosti odgovarao ovom tipu petlji. Prvo, ispitivanje uslova iz drugog koraka tekstualnog opisa Euklidovog algoritma iz primera 1.6 (strana 19), kod dijagrama toka sa slike 2.25 nalazi se tek posle pomeranja vrednosti N u M i R u N. Ovo je moguće zbog toga što ispitivanje uslova pre ili posle pomeranja sadržaja ne utiče na ishod algoritma.

Druga izmena odnosi se na prikaz promenljive M, umesto N iz primera 1.6 i 2.11. Da je prikazana vrednost promenljive N, prikazana vrednost bi uvek bila 0. Razlog za ovo je korak neposredno pre uslova, koji u N upisuje R. Ako je algoritam završio petlju i došao do prikaza rezultata, to znači da je $R = 0$, a zbog pomenutog koraka i $N = 0$. Vrednost koju treba prikazati nalazi se u promenljivoj M, jer je tu pomerena u pretposlednjem koraku pre uslova ($M \leftarrow N$). $\triangle$

Za implementaciju algoritma koji prirodno nameće korišćenje *repeat-until* petlje može se koristiti i *while* petlja. Kako je ključna razlika između ovih petlji to da se kod *repeat-until* petlje telo petlje izvršava bar jednom pre nego što izvršenje dođe do ispitivanja uslova, *repeat-until* se može implementirati pomoću *while* umetanjem jedne kopije tela *repeat-until* petlje pre početka *while* petlje, na način prikazan na slici 2.26.



Slika 2.26: Implementacija petlja tipa *repeat-until* pomoću petlje tipa *while*

Potrebno je napomenuti da je uslov na dijagramu sa slike 2.26 negiran, da bi zadovoljio sled događaja po kome se kod *while* petlje telo izvršava dok je uslov ispunjen, a ne dok nije, kao kod *repeat-until* petlje.

Ova osobina da je algoritamsku strukturu koja prirodno nameće upotrebu *do-while* petlje moguće implementirati i *while-do* petljom ima za posledicu da neki programski jezici nemaju osnovnu strukturu koja implementira ovu petlju. Bez obzira na ovaj nedostatak u tim programskim jezicima, upotrebom *while-do* petlje moguće je implementirati bilo koji algoritam.

For **petlja**

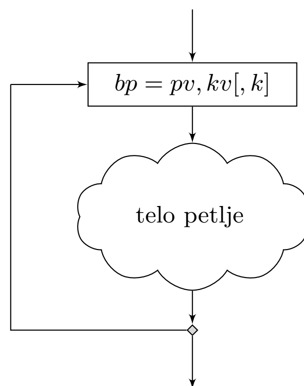
Petlja tipa *for* je osnovna algoritamska struktura kod koje se telo petlje izvršava više puta, a broj iteracija jeste poznat neposredno pre početka izvršavanja petlje. Broj iteracija petlje može biti zadat konstantom (npr. 10 ponavljanja), ili može biti zadat parametarski (npr. N ponavljanja), s tim da vrednost parametra u slučaju parametarskog zadavanja mora biti poznata pre početka petlje.

Pored zadatog broja iteracija, *for* petlja ima dodatni brojač čija je vrednost jednaka rednom broju trenutne iteracije. Vrednost ovog brojača se pamti u pro-

menljivoj koja se naziva *brojač petlje*, i čije ime zadaje programer. Broj iteracija petlje određena je

1. početnom vrednošću brojača,
2. krajnjom vrednošću brojača, i
3. korakom.

Način predstavljanja algoritamske strukture *for* dijagramom toka prikazan je na slici 2.27. Oznake na slici su: bp - brojač petlje, pv - početna vrednost, k - krajnja vrednost, i k - korak. Korak k nalazi se u zagradama '[' i ']', što znači da je ovaj parametar opcioni, t.j. da se može i izostaviti. Ukoliko je korak izostavljen podrazumevana vrednost je $k = 1$. U tom slučaju blok *for* petlje je $bp = pv, kv$.



Slika 2.27: Petlja tipa *for*

Princip rada ove petlje je sledeći: kada izvršenje algoritma dođe do *for* petlje, brojač petlje dobija početnu vrednost $bp = pv$ i telo petlje se izvršava jednom. Nakon završetka tela petlje brojač petlje se uvećava za korak k i to postaje nova vrednost brojača petlje $bp = bp + k$.[1] Izvršenje se vraća na zaglavlje petlje gde se proverava da se ovim uvećavanjem brojača premašila krajnja vrednos kv, t.j. da li je $bp \leq kv$? Ako jeste $bp \leq kv$, telo petlje se izvršava još jednom, itd.

Kao što je prethodno pomenuto, iako se broj iteracija *for* petlje eksplicitno ne zadaje, broj iteracija je poznat i može se odrediti na osnovu parametara pv, kv i k

[1] Operator '=' koristi se za dodelu vrednosti promenljivoj sa leve strane. Izraz $bp = bp + k$ ne treba posmatrati kao matematički izraz, već kao naredbu koja se izvršava iz dva dela: u prvom delu se određuje vrednost $bp + k$, a u drugom delu se ta vrednost operatorom dodele '=' upisuje ponovo u istu promenljivu bp.

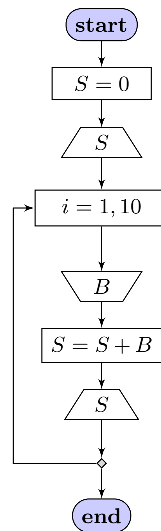
zadatih u okviru dijagrama toka sa slike 2.27. Broj iteracija *for* petlje je

$$BI = \left\lfloor \frac{kv - pv}{k} \right\rfloor + 1, \tag{2.2}$$

gde oznaka $\lfloor x \rfloor$ označava prvi ceo broj manji ili jednak x.

Primer 2.16 (Primer upotrebe *for* petlje)**. Zadatak**: Nacrtati strukturni dijagram toka algoritma za pomoć operateru semafora na košarkaškom takmičenju u slobodnim bacanjima. Takmičar ima pravo na 10 slobonih bacanja. Na početku bacanja na semaforu je prikazan rezultat 0. Nakon svakog bacanja operater unosi postignuti broj poena - 0 ili 1. Nakon unosa na semaforu se prikazuje nova vrednost, uvećana za broj poena postignut u proteklој iteraciji.

Rešenje: Tekst zadatka se može uzeti kao algoritam u tekstualnoj reprezentaciji, jer su koraci za unos, obradu i prikaz jasno izdvojeni, uz precizan opis njihove strukture. Algoritam opisan u tekstu zadatka u osnovi odgovara osnovnoj algoritamskoj strukturi *for* petlje. Algoritam je predstavljen dijagramom toka na slici 2.28.



Slika 2.28: Primer petlje tipa *for*

Vrednost semafora označena je promenljivom S, kojoj je na početku dodeljena vrednost $S = 0$, a vrednost postignutih poena u jednoj iteraciji promenljivom B. Promenljivoj za pamćenje brojača petlje dat je naziv i. Početna i krajnja vrednost su konstante 1 i 10. Korak *for* petlje k je izostavljen, pa se smatra da ima podrazumevanu vrednost $k = 1$.

Telo ove petlje će se izvršiti tačno 10 puta, jer je početna vrednost 1, za koju se telo petlje izvršava jednom, nakon čega se brojač uvećava za korak, t.j. postaje 2.

Sa ovom novom vrednošću telo petlje se izvršava još jednom, i tako dalje, sve dok brojač i ne dostigne vrednost 10. Poslednji put će se telo petlje izvršiti za vrednst brojača $i = 10$. Uvećavanje brojača dato je izrazom u kom figuriše operator dodele $S = S + B$.

Napomena: Telo petlje bi se izvršavalo 10 puta i da je početna vrednost $pv = 11$, a krajnja vrednost $kv = 20$, sa korakom $k = 1$, odnosno petlja bi se izvršila 10 puta za bilo koje vrednosti pv, kv i k koje zamenom u izraz (2.2) daju rezultat $BI = 10$. Takav algoritam bi u potpunosti bio ekvivalentan algoritmu sa slike 2.28.

△

Prilikom svakog novog izvršenja tela petlje kod petlji tipa *for*, brojač petlje ima novu vrednost, koja je u odnosu na prethodnu vrednost uvećana za korak petlje. Ova vrednost se može koristiti u telu petlje kako bi se u okviru tela razlikovalo koja se iteracija trenutno izvršava.

Napomena: Promenljiva u kojoj se pamti vrednost brojača petlje se, ukoliko je to potrebno, može koristiti u izrazima u okviru petlje. Ova promenljiva u svakoj iteraciji ima novu vrednost, koja se u odnosu na vrednost u prethodnoj iteraciji razlikuje za korak petlje k.

Primer 2.17 (Dijagram toka algoritma za sumiranje prvih N brojeva)**. Zadatak**: Nacrtati strukturni dijagram toka algoritma koji određuje i prikazuje sumu prvih N prirodnih brojeva. Parametar N zadaje korisnik.

Rešenje: Suma prvih N prirodnih brojeva je

$$S = 1 + 2 + 3 + \cdots + N. \tag{2.3}$$

Kako ne možemo sabrati sve brojeve u jednom koraku, jer parametar N zadaje korisnik u toku izvršenja programa, a operator "$+ \cdots +$" ne postoji u programskom jeziku, prethodni izraz je potrebno podeliti na korake, koji će ujedno i biti koraci algoritma. Jedno moguće rešenje je sledeće:

1. Postavimo vrednost sume S na početku na $S = 1$, što je prva vrednost sume ukoliko sumiranje izraza (2.3) počnemo sa leve strane.

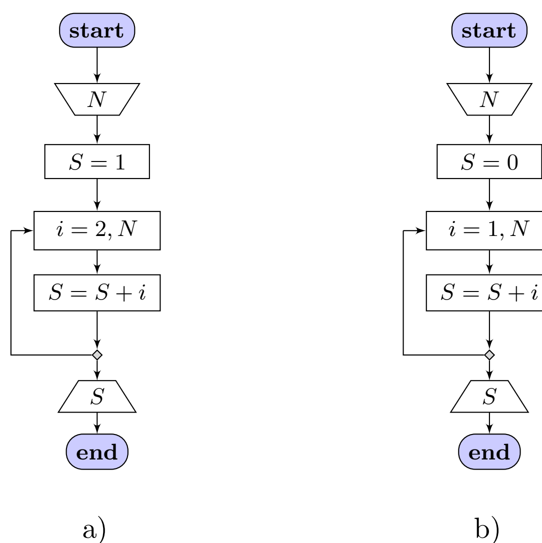
2. Sumi S dodajemo sledeći broj, t.j. određujemo vrednost izraza $S + 2$. U ovom trenutku se nameće ključno pitanje: Gde pamtiti ovaj rezultat? Jedno rešenje je kreirati novu promenljivu i dati joj neko ime. Međutim, u tom slučaju bi i za naredne korake trebalo da kreiramo po jednu promenljivu za svaki korak, pa kako broj koraka zavisi od parametra N koji zadaje korisnik dolazimo u situaciju da ne znamo koliko nam tačno promenljivih treba.

   Kako nam nakon određivanja nove vrednosti sume prethodna vrednost nije potrebna, možemo novu vrednost zapamtiti na mestu stare, t.j. $S + 2 \to S$. U S je nakon ovog koraka $S = 1 + 2 = 3$.

3. Na isti način trenutnoj sumi S dodajemo treći broj $S + 3 \to S$. U S je nakon ovog koraka $S = (1 + 2) + 3 = 6$.

4. ...
5. Poslednji korak je $S + N \to S$, nakon čega je u S vrednost izraza (2.3).

Iz navedenog algoritma se vidi da se koraci od 2. koraka pa nadalje ponavljaju. Operacija koja se izvodi u svakom koraku je $S = S + i$, gde je i na početku $i = 2$, nakon toga je $i = 3$, itd. U poslednjem koraku je $i = N$. Ovo ponavljanje se može opisati *for* petljom kod koje je brojač petlje i, početna vrednost $i = 2$, a krajnja vrednost $i = N$. Dijagram toka ovog algoritma prikazan je na slici 2.29a. Rešenje dato na slici 2.29b je potpuno ekvivalentno, uz razliku da je prvi korak $S = 1$ "uvučen" u petlju, pa ova varijanta algoritma radi tačno, čak i ako korisnik za parametar N unese $N = 0$. Za slučaj $N = 0$ algoritam sa slike 2.29a neće nijednom izvršiti telo *for* petlje, a u promenljivoj S će ostati vrednost sa početka $S = 1$, koja za ovaj slučaj nije tačna.



Slika 2.29: Dijagram toka algoritma za sumiranje prvih N brojeva

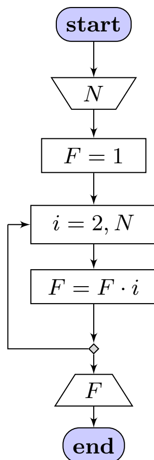
△

Primer 2.18 (Dijagram toka algoritma za određivanje faktorijala zadatog broja)**.**
Zadatak: Nacrtati strukturni dijagram toka algoritma koji određuje i prikazuje faktorijal zadatog broja N. Broj N zadaje korisnik. Smatrati da korisnik zadaje broj N tako da važi $N \geq 1$.

Rešenje: Faktorijal broja N se može odrediti kao

$$F = 1 \cdot 2 \cdot 3 \cdot \ \cdots \ \cdot N. \tag{2.4}$$

Algoritam je moguće tekstualno opisati na sličan način kako je urađeno u primeru 2.17. Dijagram toka algoritma za određivanje faktorijala zadatog broja N prikazan je na slici 2.30. △



Slika 2.30: Dijagram toka algoritma za određivanje faktorijala zadatog broja

U prethodnim primerima algoritmi su projektovani tako da je korak petlje $k = 1$. U narednom primeru ćemo razmotriti slučaj upotrebe *for* petlje kada je korak petlje različit od podrazumevane vrednosti.

Primer 2.19 (Dijagram toka algoritma za sumiranje prvih N parnih brojeva)**.**
Zadatak: Nacrtati strukturni dijagram toka algoritma koji određuje i prikazuje sumu prvih N parnih brojeva. Parametar N zadaje korisnik.

Rešenje: Suma prvih N parnih prirodnih brojeva je

$$S = 2 + 4 + 6 + \cdots + 2N. \tag{2.5}$$

Ukoliko algoritam za ovu sumu projektujemo kao u prethodnim primerima *for* petljom, tako da u svakoj iteraciji petlje određujemo jednu vrednost $S + i$ i ponovo je upisujemo u S, tada sumu možemo predstaviti izrazom

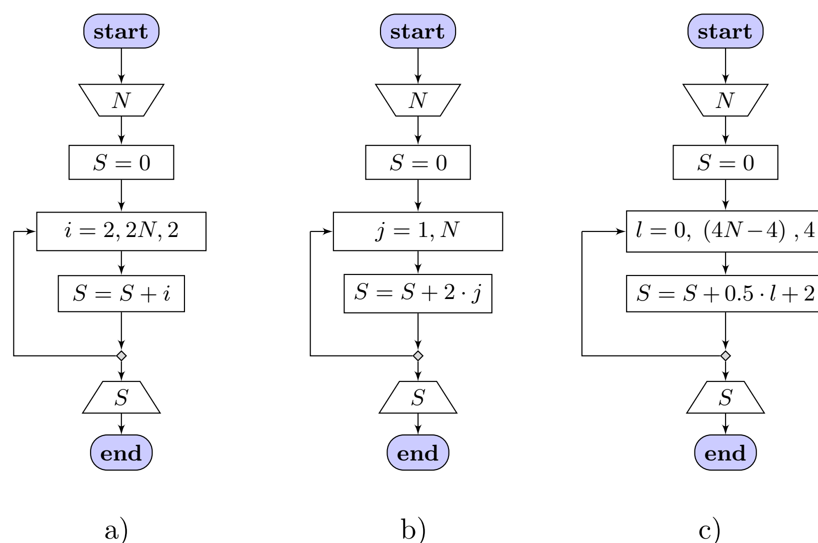
$$S = ((((0 + 2) + 4) + 6) + \cdots + 2N).$$

gde su koraci algoritma:

1. $S = 0$
2. $S = S + 2$
3. $S = S + 4$

4. $S = S + 6$
5. ...
6. $S = S + 2N$

iz čega se može uočiti ponavljanje počev od drugog koraka, takvo da je jedina razlika između operacija $S = S + x$ u vrednosti x, koja je jednaka redom $2, 4, 6$, itd. Na mestu ove pozicije se može uvesti brojač sa početnom vrednošću brojača petlje $pv = 2$, tako da se brojač petlje u svakom koraku povećava za $k = 2$, a krajnja vrednost brojača petlje je $kv = 2N$. Strukturni dijagram toka ovog algoritma prikazan je na slici 2.31a.



Slika 2.31: Dijagram toka algoritma za sumiranje prvih N parnih brojeva

Ovaj algoritam moguće je projektovati i uzimajući u obzir matematičku činjenicu da je prvih N parnih brojeva moguće dobiti tako što se prvih N celih brojeva redom množe sa 2, čime se takođe dobija niz brojeva iz izraza (2.5). Uzmimo j za oznaku brojača ove petlje, da bismo izbegli zabunu. U ovom slučaju operacija u jednoj iteraciji petlje algoritma bi bila $S = S + 2 \cdot j$. Petlja u ovom slučaju počinje za $j = 1$, a završava se za $j = N$, sa podrazumevanim korakom $k = 1$. Dijagram toka ovog algoritma prikazan je na slici slici 2.31b. Algoritmi sa slike 2.31a i slike 2.31b daju isti rezultat, odnosno ekvivalentni su.

Napomena: Zamenom brojača *for* petlje x brojačem y tako da je

$$x = c_1 \cdot y + c_2,$$

gde su c_1 i c_2 konstante, vrši se linearna transformacija *for* petlje u **ekvivalentnu** *for* petlju, ukoliko se nove vrednosti granica petlje odrede na osnovu starih pomoću izraza

$$y = \frac{x - c_2}{c_1},$$

odnosno, drugim rečima ukoliko se:

1. granice petlje i korak zamene novim vrednostima

$$pv_y = \frac{pv_x - c_2}{c_1}, \quad kv_y = \frac{kv_x - c_2}{c_1}, \quad k_y = \frac{k_x}{c_1},$$

2. kao i da se sva pojavljivanja brojača x u izrazima u telu petlje zamene sa $x = c_1 \cdot y + c_2$.

Algoritam sa slike 2.31b dobijen je zamenom brojača i sa slike 2.31a brojačem j, za koji važi $i = 2j + 0$, t.j. $c_1 = 2$ i $c_2 = 0$, a algoritmi sa slike 2.31c dobijen je zamenom brojača i sa slike 2.31a brojačem l, za koji važi $i = 0.5l + 2$, t.j. $c_1 = 1/2$ i $c_2 = 2$.

Za slučaj algoritma sa slike 2.31c, početna vrednost brojača je $l = 0$. Tada je operacija koja se izvršava $S = S + 0.5 \cdot 0 + 2$, t.j. $S = S + 2$. U sledećem koraku vrednost brojača je $l = 4$, jer je korak ove petlje $k = 4$. Operacija koja se izvršava je $S = S + 0.5 \cdot 4 + 2$, t.j. $S = S + 4$. Kada se zamene vrednosti za treći korak u kome je $l = 8$ dobija se $S = S + 6$, itd.

Algoritmi sa slika 2.31 a, b i c su ekvivalentni.

△

Definicija 2.13 (Linearna transformacija *for* petlje)**.** Zamenu brojača *for* petlje x brojačem y tako da je

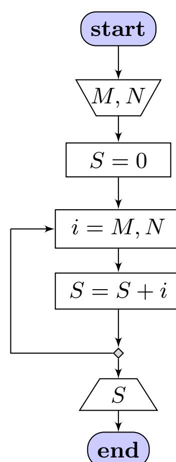
$$x = c_1 \cdot y + c_2,$$

gde su c_1 i c_2 konstante, nazivamo linearnom transformacijom *for* petlje.

U svim prethodnim primerima početna vrednost brojača *for* petlje pv zadavana je konstantom, dok smo za krajnju vrednost koristili u nekim slučajevima konstante, a u nekim slučajevima promenljive. Generalno, svi parametri petlje (pv, kv i k) mogu se zadati proizvoljno konstantama, promenljivama, ili izrazima, zavisno od algoritma koji je potrebno predstaviti.

Primer 2.20 (Zbir celih brojeva počev od broja M do broja N)**. Zadatak**: Nacrtati strukturni dijagram toka algoritma koji određuje i prikazuje sumu prirodnih brojeva počev od broja M zaključno sa brojem N. Parametre M i N zadaje korisnik. Smatrati da korisnik zadaje parametre tako da uvek važi $M \leq N$.

Rešenje: Dijagram toka algoritma opisanog zadatkom prikazan je na slici 2.32. Granične vrednosti brojača petlje su parametri koje zadaje korisnik. △



Slika 2.32: Dijagram toka algoritma za sumiranje zadatog opsega celih brojeva

Parametri *for* petlje mogu imati kako celobrojne, tako i realne vrednosti, ili vrednosti bilo kog drugog numeričkog tipa. U nekim programskim jezicima, pored numeričkih vrednosti, moguće je zadati *for* petlju nad bilo kojim uređenim skupom, tako da brojač uzima redom elemente iz zadatog skupa (npr. programski jezik PHP). Ovakve petlje se nazivaju *foreach* petlje i neće biti obrađivane u ovom udžbeniku.

Takođe je potrebno napomenuti da za granične vrednosti brojača *for* petlje ne mora važiti $pv < kv$. Za granične vrednosti brojača može važiti i $pv > kv$. U slučaju da je $pv > kv$, korak petlje k mora biti negativan broj.

Definicija 2.14 (Petlja koja broji unazad)**.** Za *for* petlju za koju važi da je $pv > kv$ i korak petlje k je negativan broj se kaže da *broji unazad*.

Definicija 2.15 (Petlja koja broji unapred)**.** Za *for* petlju za koju važi da je $pv < kv$ i korak petlje k je pozitivan broj se kaže da *broji uunapred*.

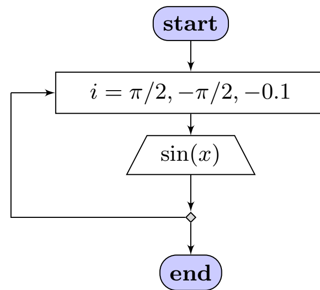
Petlja kod koje su granične vrednosti brojača postavljene za brojanje unapred, t.j. $pv < kv$, a korak petlje k je negativan neće se nijednom izvršiti, već će se izvršenje preneti na prvi naredni blok iza petlje. Telo *for* petlje se takođe neće nijednom izvršiti ukoliko su granične vrednosti brojača postavljene za brojanje unazad, t.j. $pv > kv$, a korak petlje k je pozitivan broj. Granični slučaj imamo za $pv = kv$. U tom slučaju će se telo petlje izvršiti samo jednom, bez obzira na to da li je korak petlje k pozitivan ili negativan broj. Brojač petlje će u toj jednoj iteraciji imati vrednost $bp = pv$.

Primer 2.21 (Dijagram toka algoritma za prikaz vrednosti matematičke funkcije).
Zadatak: Nacrtati strukturni dijagram toka algoritma koji prikazuje vrednosti matematičke funkcije

$$y = \sin(x)$$

za vrednosti parametra x od $\pi/2$ unazad do $-\pi/2$, sa korakom 0.1.

Rešenje: Kako je u prethodnom tekstu rečeno, brojač *for* petlje može biti realan broj, a petlja može imati i negativan korak. Za rešenje ovog primera izabraćemo za početnu vrednost brojača zadatu vrednost $pv = \pi/2$, a za krajnju vrednost brojača vrednost $kv = -\pi/2$. Korak petlje je zadat i iznosi $k = -0.1$. Dijagram toka ovog algoritma prikazan je na slici 2.33.



Slika 2.33: Dijagram toka algoritma za prikaz vrednosti matematičke funkcije $\sin(x)$ opisan *for* petljom sa realnim granicama koja broji unazad

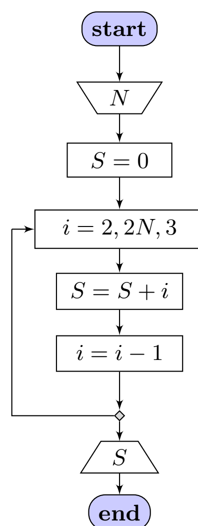
Brojač *for* petlje prikazane na slici 2.33, u zavisnosti od preciznosti sa kojom se predstavljaju realni brojevi u konkretnoj implementaciji, imaće redom vrednosti $i = 1.57, 1.47, 1.37, 1.27, \ldots, -1.47, -1.57$. $\triangle$

Postoje programski jezici koji dozvoljavaju promenu parametara *for* petlje u toku izvršenja tela petlje, kao i jezici koji ovakvu promenu parametara u toku izvršenja petlje ne dozvoljavaju. Programski jezik C spada u grupu jezika koji dozvoljavaju postojanje naredbi koje menjaju sadržaj brojača petlje u petlji.

Kod strukturnog programiranja promena parametara *for* petlje u toku izvršenja petlje nije dozvoljena. Ovakva promena je nestrukturna operacija, što znači da se tok ovakvog algoritma teško prati, a semantika teško razume. Ovo je ilustrovano u sledećem primeru.

Primer 2.22 (Nestrukturni dijagram toka algoritma za sumiranje parnih brojeva).
Zadatak: Nacrtati dijagram toka algoritma koji određuje i prikazuje sumu prvih N parnih brojeva. Parametar N zadaje korisnik.

Rešenje: Jedno moguće rešenje za sumiranje prvih N parnih brojeva kod koga se parametar petlje menja u petlji prikazano je na slici 2.34. Koraci algoritma predstavljenog dijagramom toka na slici 2.34 su:



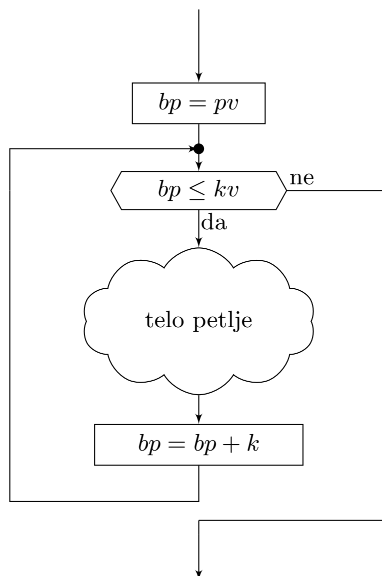
Slika 2.34: Nestrukturni dijagram toka algoritma za sumiranje prvih N parnih brojeva

1. Korisnik zadaje N.
2. S postaje 0.
3. Počinje *for* petlja, $i = 2$.
4. $S + 2$ se upisuje ponovo S, pa je nova vrednost S vrednost $S = 2$.
5. i se umanjuje za 1 i postaje $i = 1$.
6. Po povratku na početak petlje brojaču petlje se dodaje vrednost koraka $k = 3$, čime i postaje $i = 4$.
7. $S + 4$ se upisuje ponovo S, pa je nova vrednos u S vrednost $S = 2 + 4 = 6$.
8. i se umanjuje za 1 i postaje $i = 3$.
9. Po povratku na početak petlje brojaču petlje se dodaje vrednost koraka $k = 3$, čime i postaje $i = 6$.
10. itd.

Kao što se iz ovog primera vidi, promena brojača u okviru petlje značajno komplikuje praćenje koraka algoritma. Teško je na osnovu dijagrama toka algoritma sa petljom čiji je korak $k = 3$ na prvi pogled zaključiti da je ovo algoritam koji sabira parne brojeve, odnosno svaki drugi broj. △

Samim tim što se semantika dijagrama toka koji menjaju brojač petlje u petlji teško prati, otklanjanje grešaka je podjednako teško. Kao što je već rečeno, strogo strukturni programski jezici sintaksno zabranjuju ovakve konstrukcije, mada ih je potrebno izbegavati i kod jezika koji ovakve konstrukcije dozvoljavaju.

Na početku ovog poglavlja je opisan princip rada *for* petlje, gde je rečeno da se telo petlje izvršava počev od vrednosti $bp = pv$, **sve dok** vrednost brojača bp ne dostigne krajnju vrednost $bp = kv$, uz uvećanje brojača u svakoj iteraciji za korak petlje k, uz naglasak na *"sve dok"*. Ova konstrukcija se može predstaviti i *while* petljom na opisani način: na početku petlje je brojač petlje $bp = pv$; sve dok važi da je brojač petlje $bp \leq kv$, izvršiti telo petlje, uvećati brojač za korak petlje k i vratiti se na početak petlje. Predstavljanje *for* petlje petljom tipa *while* ilustrovano je na slici 2.35.



Slika 2.35: Implementacija petlja tipa *for* pomoću petlje tipa *while*

Napomena: U prethodnom poglavlju je pokazano da se petlja tipa *repeat-until* može predstaviti *while* petljom. Pokazali smo da se i petlja tipa *for* može predstaviti *while* petljom. Iz ovoga sledi zaključak da je programski jezik kompletan, t.j. da se u njemu može implementirati bilo koji algoritam ukoliko pored strukture sekvence i alternacija postoji jedino petlja tipa *while*. Drugim rečima, minimalan broj osnovnih struktura za opis bilo kog algoritma je: sekvenca, alternacija i petlja tipa *while*.

Petlje tipa *repeat-until* i *for* nisu neophodne, mada se njihovom upotrebom izbegava bespotrebno komplikovanje algoritama zamenskom upotrebom petlje tipa *while*. U slučaju korišćenja *repeat-until* izbegava se jedna kopija tela petlje pre petlje (slika 2.26), a u slučaju *for* petlje, sama petlja brine o inicijalizaciji i uvećavanju brojača.

Transformacija *for* petlje u *while* petlju se koristi prilikom projektovanja algoritma kada se želi postići optimalnije rešenje, kod koga se izvršenje može prekinuti i iz nekog drugog razloga, a ne samo kada brojač petlje *bp* dosegne krajnju vrednost kv. U ovakvim slučajevima se petlja tipa *for* zamenjuje petljom tipa *while* sa slike 2.35, a uslov petlje $bp \leq kv$ se dopunjuje i postaje

$$\Big((bp \leq kv) \bigwedge dodatni_uslov\Big).$$

2.3.4 Složene algoritamske strukture

Složene algoritamske strukture dobijaju se kombinovanjem osnovnih algoritamskih struktura. Operacije kombinovanja koje su dozvoljene kod strukturnog programiranja su nadovezivanje i ugnežđavanje.

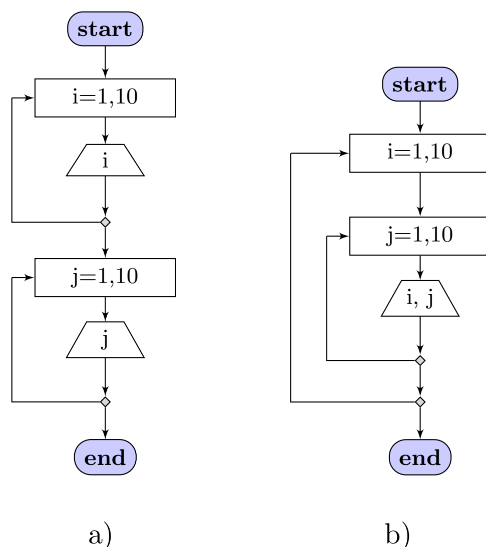
Definicija 2.16 (Nadovezivanje struktura)**.** Nadovezivanje algoritamskih struktura je operacija povezivanja kraja jedne strukture na početak druge strukture.

Definicija 2.17 (Ugnežđavanje struktura)**.** Ugnežđavanje algoritamskih struktura je operacija umetanja jedne strukture u drugu, tako da se struktura koja je kasnije počela (unutrašnja struktura) prva završava.

Primer nadovezivanja i ugnežđavanja struktura prikazan je na slikama 2.36a i 2.36b, respektivno, na primeru povezivanja dve *for* petlje.

Napomena: Na primeru sa slike 2.36b, naziv brojača unutrašnje petlje j razlikuje se od imena brojača spoljašnje petlje i, zbog toga što promena brojača petlje u petlji kod strukturnih algoritama nije dozvoljena, a upravo to bi uradila unutrašnja petlja brojaču spoljašnje petlje da je korišćena ista oznaka.

Algoritam predstavljen dijagramom toka na slici 2.36a sastoji se od dve *for* petlje nadovezane jedna na drugu. Brojač prve petlje je i, sa početnom i krajnjom vrednošću 1 i 10, respektivno. Brojač druge petlje je j, takođe sa početnom i krajnjom vrednošću 1 i 10. Obe petlje u telu petlje prikazuju trenutne vrednosti svojih brojača. Kod nadovezanih struktura sledeća struktura neće započeti dok se prethodna ne završi u potpunosti, pa tako, prva započinje petlja po i i prikazuje vrednosti brojača i redom od 1 do 10. Kada se prikaže i poslednja, deseta vrednost prva petlja se završava i prelazi se na narednu strukturu, t.j. *for* petlju po j. I ova petlja prikazuje trenutnu vrednost brojača, pa će se telo petlje izvršiti 10 puta, sa vrednostima $j = 1, 2, \dots, 10$. Ove vrednosti će takođe biti prikazane. Generalno posmatrano, algoritam predstavljen dijagramom toka na slici 2.36a prikazaće redom vrednosti $1, 2, \dots, 9, 10, 1, 2, \dots, 10$, što je ukupno 20 prikaza za ceo algoritam.



Slika 2.36: Primer nadovezivanja i ugnežđavanja struktura

Dijagram toka na slici 2.36b sastoji se od dve ugnežđene strukture. Izvršenje prvo nailazi na početak prve petlje, čime se vrednost brojača postavlja na početnu vrednost ($i = 1$). U telu prve petlje se nalazi druga *for* petlja, pa se odmah nakon ulaska u prvu petlju inicijalizuje brojač druge petlje $j = 1$. U telu unutrašnje petlje će u prvoj iteraciji biti prikazane vrednosti brojača i i j: $(1, 1)$.

Kod ugnežđenih petlji spoljašnja petlja ne prelazi na narednu iteraciju dok se u potpunosti ne okončaju sve iteracije unutrašnje petlje. Kako je unutrašnja petlja izvršila telo petlje tek po prvi put, izvršenje se vraća na početak unutrašnje petlje, i vrednost brojača j se uvećava za korak, pa je $j = 2$. Vrednost i je nepromenjena, jer još uvek traje prva iteracija spoljašnje petlje ($i = 1$). Vrednosti $(1, 2)$ će takođe biti prikazane, nakon čega se prelazi na treću iteraciju unutrašnje petlje i prikazuje $(1, 3)$, pa $(1, 4)$, i tako redom do $(1, 10)$, čime je unutrašnja petlja okončana. Tek tada se završava u potpunosti jedna iteracija tela spoljašnje petlje, pa se izvršenje vraća na početak petlje po i i uvećava vrednost brojača na $i = 2$. Za $i = 2$ unutrašnja petlja ponovo izvršava ceo ciklus od inicijalizacije $j = 1$ do vrednosti $j = 10$, pa će prikazane vrednosti u ovoj iteraciji petlje po i biti: $(2, 1)$,$(2, 2)$,...,$(2, 10)$.

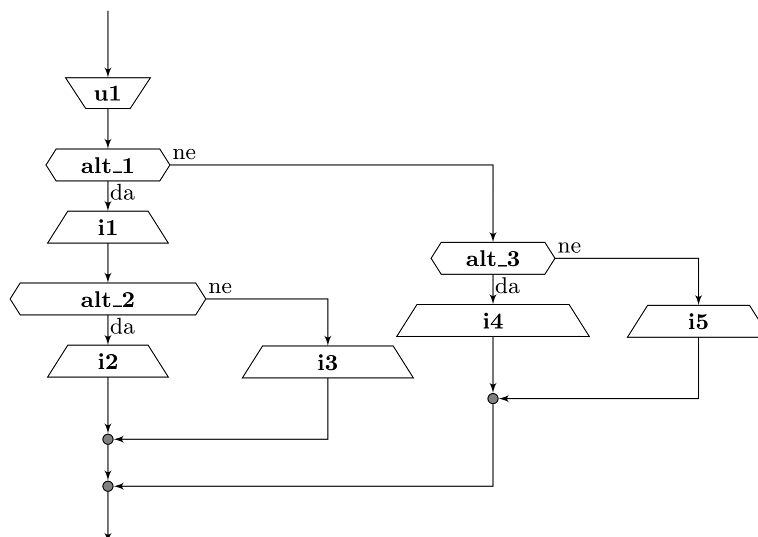
U trećoj iteraciji spoljašnje petlje biće prikazane vrednosti od $(3, 1)$, do $(3, 10)$, itd. Poslednje vrednosti koje će ovaj dijagram toka prikazati su $(10, 10)$. Za razliku od dijagrama toka algoritma sa slike 2.36a koji je prikazivao vrednosti 20 puta, dijagram toka sa slike 2.36b vrednosti prikazuje 100 puta.

Napomena: Spoljašnja petlja neće preći na narednu iteraciju sve dok se u potpunosti ne okončaju sve unutrašnje strukture.

Definicija 2.18 (Strukturni dijagram toka algoritma)**.** Strukturni dijagram toka algoritma je algoritamska struktura dobijena isključivo nadovezivanjem i ugnežđavanjem osnovnih algoritamskih struktura.

Strukturni dijagram toka ima osobinu da svaka struktura, osnovna ili složena ima jednu ulaznu i jednu izlaznu tačku.

Primer ispravno ugnježdenih alternacija prikazan je na slici 2.37. Dijagram toka

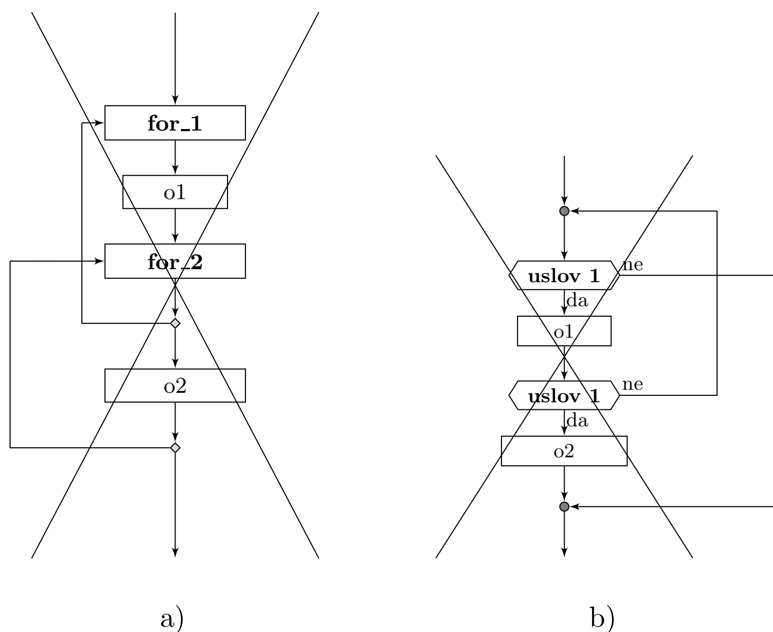


Slika 2.37: Primer ugnježdenih alternacija

sa slike 2.37 sadrži ukupno tri alternacije, *alt_1*, *alt_2* i *alt_3*, kojima je pridruženo 5 blokova za prikaz i jedan blok za unos parametara. Alternacija *alt_2* počinje i završava se u "da" grani alternacije *alt_1* i sa blokom *i1* čini sekvencu koja je ugnеždena u alternaciju *alt_1*. Alternacija *alt_3* počinje i završava se (ugnеždena je) u "ne" grani alternacije *alt_1*.

Primer nestruktunih dijagama toka prikazan je na slici 2.38. Na slici 2.38a prikazane su dve petlje koje se preklapaju. Početak druge petlje je pre kraja prve petlje. Da je obrnuto petlje bi činile sekvencu. Takođe, kraj prve petlje je pre kraja druge petlje. Da ovo nije slučaj petlje bi bile ugnеždene. Dijagram toka sa slike 2.38a nije moguće implementirati u ovakvom obliku.

Na slici 2.38b prikazan je nestrukturni primer alternacija čije se "da" i "ne" grane ne završavaju u jednoj tački, kako je to pokazano na slici 2.15. Ovaj dijagram toka takođe ne ispunjava uslove strukturalnosti. Međutim, dijagram toka sa slike 2.38b moguće je implementirati nestrukturnim naredbama tzv. **uslovnog "skoka"**, ukoliko ciljni programski jezik podržava nestrukturno programiranje i ima ovakve naredbe. Naredbe uslovnog skoka imaju uslov i oznaku linije programa

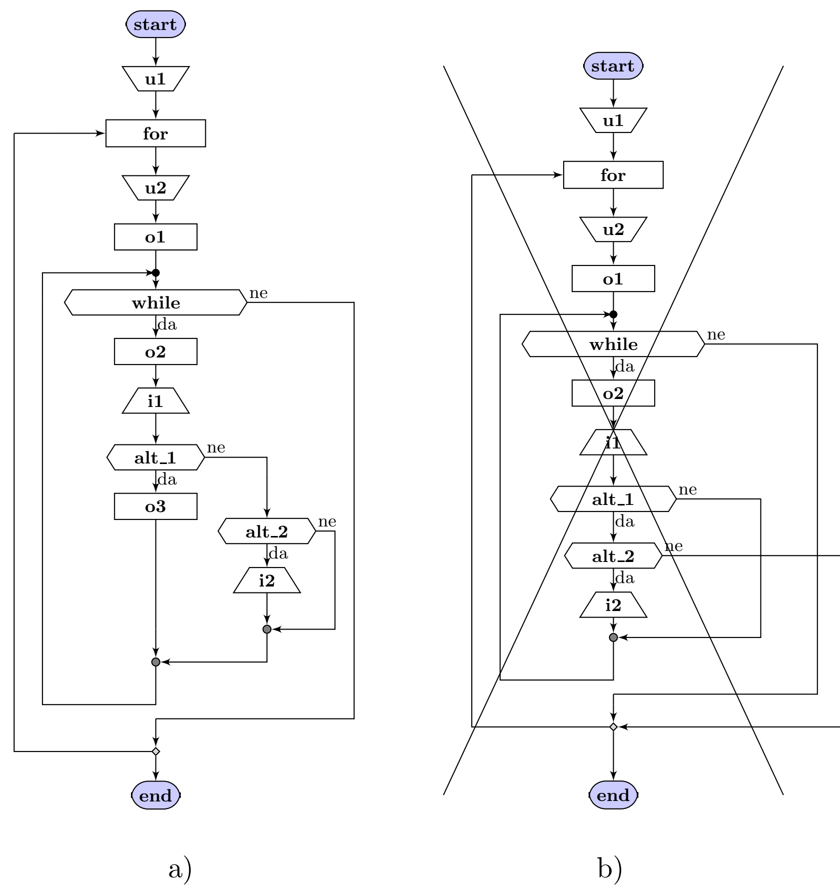


Slika 2.38: Primer nestrukturnih dijagrama toka: a) preklapajuće petlje, b) nestrukturni uslovni skokovi

na koju treba preći ako je uslov ispunjen. Ove naredbe unose nestrukturnost u tok izvršenja i veoma su česte kod mašinskih i asemblerskih jezika. Mašinski i asemblerski jezici nemaju strukture alternacije ni petlje, pa se sve algoritamske strukture implementiraju naredbama uslovnog skoka. Dijagrami toka izvršenja programa na asemblerskom jeziku imaju elemente dijagrama prikazanog na slici 2.38b, često su nestrukturni, i zbog toga ih je vrlo teško pratiti i odrediti semantiku tako napisanog programa. Otklanjanje semantičkih grešaka u kompleksnom nestrukturnom algoritmu je takođe vremenski veoma zahtevan proces.

Na slici 2.39 uporedo su prikazani primeri pravilnog i nepravilnog kombinovanja osnovnih algoritamskih struktura. Na slici 2.39a prikazana je kombinacija dve petlje, *for* i *while*, i dva grananja, označena sa *alt_1* i *alt_2*. Pored ove 4 strukture dijagram toka sadrži i dva bloka za unos (u1 i u2), dva bloka za prikaz (i1 i i2), i tri bloka za obradu (o1, o2 i o3). Posmatrajući strukture po dubini, počev od spoljašnjih struktura ka unutrašnjim, na slici 2.39a razlikujemo:

1. sekvencu ulaza **u1** i
2. **for** petlju (petlja se posmatra kao jedinstvena celina u sekvenci u kojoj se nalazi);



Slika 2.39: Primer složenog strukturnog i nestrukturnog dijagrama toka

2.1. u *for* petlji imamo sekvencu **u2**,

2.2. **o2** i

2.3. **while** petlju (petlja se posmatra kao celina u sekvenci);

2.3.1. u *while* petlji imamo sekvencu **o2**,

2.3.2. **i1** i

2.3.3. **alt_1** grananja (posmatra se kao celina u sekvenci);

2.3.3.1. u da grani *alt_1* grananja **o3**

2.3.3.2. u ne grani grananje **alt_2**

Kao što se iz prethodnog nabrajanja vidi, kod strukturnih algoritama se tačno može reći koja struktura se nadovezuje na koju strukturu, i koja struktura je deo

koje strukture. Ovo nije slučaj kod nestrukturnih dijagrama toka. Nestrukturni dijagram toka algoritma prikazan je na slici 2.39b. Nestrukturnost ovog dijagrama toka se ogleda u grananju koje se nalazi najdublje u strukturi i nosi oznaku *alt_2*. "Ne" grana ovog grananja izlazi iz okvira strukture u koju je grananje ugnežđeno, pa se grananje koje je počelo u strukturi *alt_1* završava van ove strukture. Na osnovu ovoga ne može se reći da se grananje *alt_2* koje počinje u okviru grananja *alt_1* u okviru ovog grananja i završava, kako bi se moglo reći da je ugnežđeno.

Po ovome ni uslov da svaka struktura ima jednu ulaznu i jednu izlaznu granu takođe nije ispunjen. Naime, *alt_1* grananje ima dve izlazne grane: jednu regularnu i drugu koja "iskače" ako uslov *alt_2* nije ispunjen. Kao što je ranije pomenuto, ovakav algoritam nije moguće implementirati na programskom jeziku bez korišćenja nestrukturnih naredbi uslovnog skoka.

Primer 2.23 (Strukturni dijagram toka za određivanje faktorijala zadatog broja). **Zadatak**: Nacrtati strukturni dijagram toka algoritma i napisati strukturni program na programskom jeziku C koji određuje i prikazuje faktorijal unetog celog broja. Faktorijal broja n definiše se kao

$$n! = \begin{cases} nije\ def., & n < 0 \\ 1, & n = 0 \\ 1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n, & x \geq 1 \end{cases} . \tag{2.6}$$

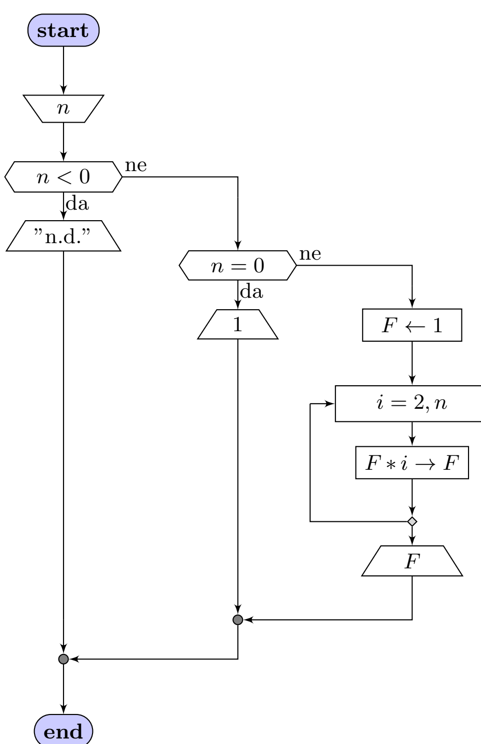
Smatrati da korisnik može zadati bilo koji ceo broj, pozitivan ili negativan.

Rešenje: Strukturni dijagram toka algoritma za određivanje faktorijala zadatog broja prikazan je na slici 2.40. Kako korisnik može zadati i negativan broj n, za koji faktorijal nije definisan, prvo je potrebno proveriti da se nije desio ovaj slučaj. Ukoliko je uslov prvog grananja $n < 0$ dijagrama toka sa slike 2.40 ispunjen, poruka "nije definisan" će biti prikazana i algoritam će se završiti. Ukoliko ne važi da je $n < 0$, tada se po jednačini (2.6) može desiti da je $n = 0$, ili da je $n > 0$. Način na koji se određuje faktorijal u oba slučaja se razlikuje, i opisan je jednačinom (2.6). Za slučaj da je zadato $n = 0$, vrednost faktorijala je 1. Ovaj slučaj je pokriven "da" granom alternacije $n = 0$. Ukoliko je broj pozitivan, algoritam u "ne" grani alternacije $n = 0$ svodi se na dijagram toka iz primera 2.18. △

Primer 2.24 (Dijagram toka algoritma za vraćanje kusura). **Zadatak**: Nacrtati strukturni dijagram toka algoritma koji određuje i prikazuje sve načine na koje je moguće vratiti kusur od N dinara, novčanicama u apoenima od 1, 2 i 5 dinara.

Rešenje: Jedno moguće rešenje problema je provera svih mogućih kombinacija i prikaz kombinacije koja zadovoljava uslov da je zbir jednak kusuru koji je potrebno vratiti. Sve moguće kombinacije su:

1. **0** od 5din, **0** od 2 din, **0** od 1 din;
2. **0** od 5din, **0** od 2 din, **1** od 1 din;
3. **0** od 5din, **0** od 2 din, **2** od 1 din;



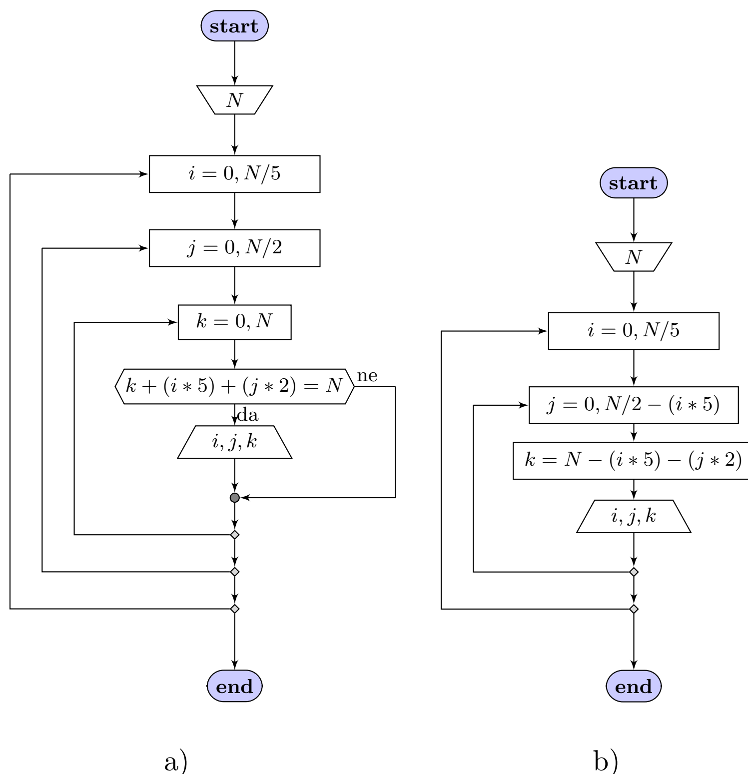
Slika 2.40: Primer složenog strukturnog i nestrukturnog dijagrama toka

4. ...
5. **0** od 5din, **0** od 2 din, **10** od 1 din;
6. **0** od 5din, **1** od 2 din, **0** od 1 din;
7. **0** od 5din, **1** od 2 din, **1** od 1 din;
8. ...
9. **2** od 5din, **0** od 2 din, **0** od 1 din;

Kompaktnije zapisano, moguće kombinacije su (0,0,0), (0,0,1), (0,0,2), ..., (0,0,10); (0,1,0), (0,1,1), ..., (0,1,10); (0,2,0), (0,2,1), ..., (0,2,10), ..., (1,0,0), itd. Navedene kombinacije mogu se implementirati ugnežđenim *for* petljama na način prikazan na slici 2.41a. Dijagram toka algoritma sa slike 2.41a sadrži tri ugnežđene petlje po i, po j i petlju po k. Brojač i je postavljen za brojač novčanica od 5 dinara, brojač j od 2 dinara i brojač k od jednog dinara.

Potrebno je odrediti granice ovih petlji. Najmanji broj novčanica od 5 dinara u kombinacijama za vraćanje kusura je 0, a najveći broj je ceo iznos N podeljen sa

5, t.j. $N/5$, pa su i granice petlje po i vrednosti 0 i $N/5$. Na isti način određeno je da su granice petlje po j 0 i $N/2$, a petlje po k 0 i N (slika 2.41).



Slika 2.41: Dijagram toka algoritma za vraćanje kusura prilikom plaćanja

Nakon unosa vrednosti N izvršenje počinje petlja po i, koja na početku inicijalizuje brojač i na početnu vrednost 0. Brojač i zadržava ovu vrednost dok se ne okonča ceo ciklus unutrašnje petlje j od 0 do $N/2$. Vrednost brojača j se takođe neće promeniti sve dok se ne okonča ceo ciklus petlje po k od $k = 0$ do N.

Petlja po k sadrži uslov kojim se proverava da li određena kombinacija zadovoljava uslov da je zbir jednak iznosu koji je potrebno vratiti i samo takve kombinacije koje zadovoljavaju uslov se prikazuju. Sve kombinacije za koje se izvršava uslov u

slučaju da korisnik unese vrednost $N = 5$ su:

$$\begin{aligned}(i,j,k) \quad = \quad & (0,0,0),(0,0,1),(0,0,2),(0,0,3),(0,0,4),(\mathbf{0},\mathbf{0},\mathbf{5}),\\ & (0,1,0),(0,1,1),(0,1,2),(\mathbf{0},\mathbf{1},\mathbf{3}),(0,1,4),(0,1,5),\\ & (0,2,0),(\mathbf{0},\mathbf{2},\mathbf{1}),(0,2,2),(0,2,3),(0,2,4),(0,2,5),\\ & (\mathbf{1},\mathbf{0},\mathbf{0}),(1,0,1),(1,0,2),(1,0,3),(1,0,4),(1,0,5),\\ & (1,1,0),(1,1,1),(1,1,2),(1,1,3),(1,1,4),(1,1,5),\\ & (1,2,0),(1,2,1),(1,2,2),(1,2,3),(1,2,4),(1,2,5).\end{aligned} \tag{2.7}$$

Kombinacije koje zadovoljavaju uslov su istaknute zadebljanim ispisivanjem cifara u prethodnom nabrajanju. Od svih kombinacija, jedino će ove kombinacije biti prikazane za zadato $N = 5$.

Algoritam prikazan dijagramom toka na slici 2.41a vrši proveru svih mogućih kombinacija i prikazuje samo odgovarajuće. Rešenje je moguće optimizovati tako što će se uzeti sve moguće kombinacije, ali samo novčanica od 5 i od 2 dinara, a ostatak do N se svakako može izračunati i isplatiti u novčanicama od jednog dinara, ako je potrebno. Dijagram toka ovog algoritma prikazan je na slici 2.41b. Dijagram toka čine dve ugneždene petlje u okviru kojih je blok za obradu u kome se vrednost k izračunava. Sve moguće kombinacije i i j, sa preračunatim k za zadato $N = 5$ su:

$$\begin{aligned}(i,j,k) \quad = \quad & (0,0,\mathbf{5}),(0,1,\mathbf{3}),(0,2,\mathbf{1}),\\ & (1,0,\mathbf{0}).\end{aligned} \tag{2.8}$$

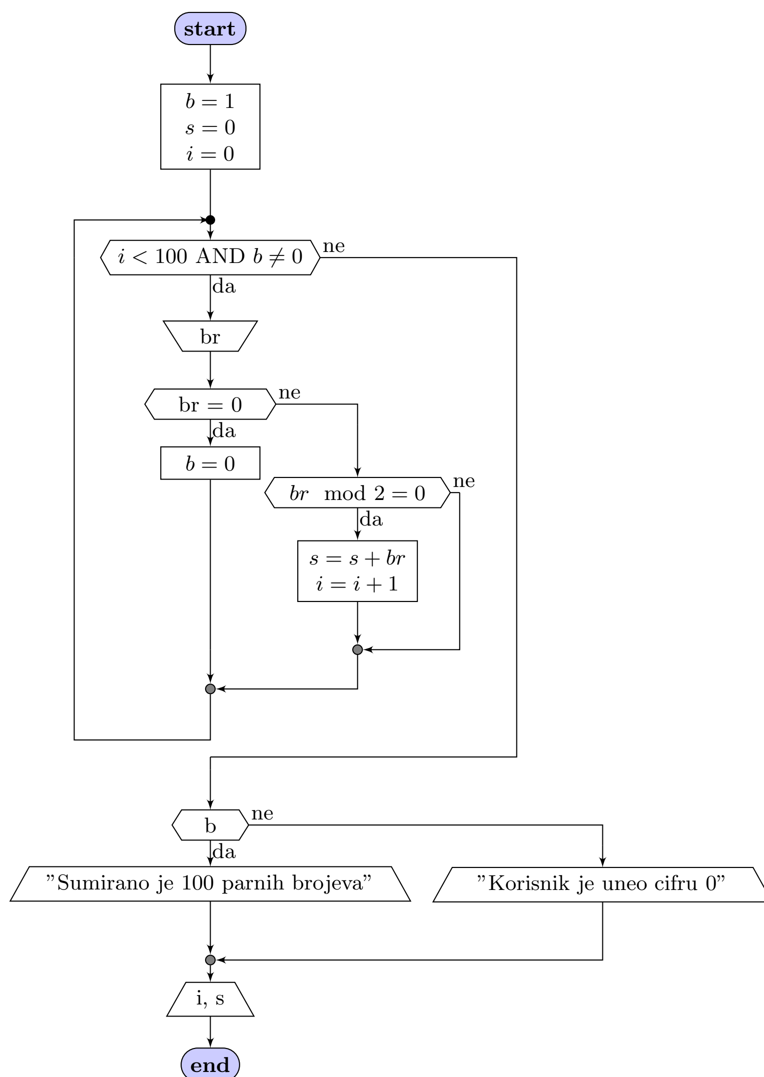
Zadebljanim ciframa je istaknuta izračunata vrednost k. Ovaj algoritam prikazuje sve kombinacije koje je odredio, jer svaka kombinacija zadovoljava uslov.

Program napisan po algoritmu sa slike 2.41b bi se višestruko puta brže izvršavao od programa napisanog po algoritmu sa slike 2.41a. Za primer $N = 5$ algoritam sa slike 2.41a ima 36 prolazaka kroz petlje, a algoritam 2.41b ima samo 4. U definiciji 1.1 rečeno je da je algoritam precizan postupak koji vodi rešenju problema. Oba algoritma na slici 2.41 su tačna, precizni su i vode rešenju problema. Međutim, sam postupak, se može razlikovati od čega zavisi i brzina izvršenja, kao što se iz ovog primera vidi. △

Primer 2.25 (Dijagram toka algoritma za sumiranje unetih brojeva)**. Zadatak**: Nacrtati strukturni dijagram toka algoritma i na programsku jeziku C napisati program koji određuje sumu prvih 100 unetih parnih brojeva. Brojeve unosi korisnik redom, sve dok ne unese nulu, ili 100 parnih brojeva. Kada se unese nula prekinuti sumiranje brojeva bez obzira da li je izvršeno sumiranje tačno 100 parnih brojeva. Prikazati razlog izlaska iz programa, sumu brojeva i koliko je ukupno brojeva sumirano.

Rešenje: Po tekstu zadatka, korisnik zadaje jedan po jedan broj. Uneti broj se dodaje sumi samo ako je paran. Ovo ponavljanje se vrši ili dok se ne unese broj 0,

ili dok se ne postigne uslov da je sumirano ukupno 100 parnih brojeva. Strukturni dijagram toka algoritma prikazan je na slici 2.42.



Slika 2.42: Dijagram toka algoritma za sumiranje unetih brojeva

Kako tačan broj unosa koje će korisnik izvršiti nije poznat, jer korisnik može da unese proizvoljan broj neparnih brojeva koji ne utiču na zadati uslov za sumiranjem 100 parnih brojeva, unos se vrši u *while* petlji (slika 2.42). Broj koji se zadaje se pamti u promenljivoj *br*. U svakoj iteraciji petlje zadaje se po jedan broj koji se

pamti u istoj promenljivoj br. Ovo je moguće zato što vrednost uneta u prethodnoj iteraciji nije potrebna, jer je već obrađena u okviru prethodne iteracije, pa se u njoj može zapamtiti novounešena vrednost.

Zadatkom su definisana dva uslova zbog kojih se može završiti unos: ako se unese 100 parnih brojeva, ili ako se unese $br = 0$ za eksplicitni prevremeni kraj programa. Za prvi uslov je potrebno obezbediti brojač. Brojač je promenljiva kojoj se pre petlje vrednost resetuje na 0, i čija se vrednost povećava za 1 kad god se unese paran broj. U ovu svrhu iskorišćena je promenljiva i na dijagramu toka sa slike 2.42. Pre petlje je vrednost i postavljena na 0, a svaki put kada se grananjem ($br \mod 2 = 0$) zaključi da je uneti broj paran, i se uvećava za 1 naredbom $i = i{+}1$.

Drugi uslov je isključiv: čim korisnik unese 0 treba izaći iz petlje. Promenljiva koja je uvedena da kontroliše ovaj uslov je promenljiva b. Algoritam je projektovan tako da promenljiva b može imati dve vrednosti: 0 i 1. Ukoliko je vrednost 1, to znači da je sve u redu, da korisnik nije uneo nulu i da treba nastaviti sa sumiranjem brojeva. Ukoliko korisnik unese 0, promenljiva b postaje 0 naredbom $b = 0$ (slika 2.42), pa drugi uslov petlje više nije ispunjen, zbog čega se izlazi iz petlje. Kako je uslov $b \neq 0$ sastavni deo uslova *while* petlje, da bi se po prvi put ušlo u petlju ovaj uslov mora biti zadovoljen, pa je zbog toga vrednost promenljive b pre početka petlje inicijalizovana na 1.

Prevremeni izlazak iz petlje kada korisnik unese $br = 0$ **strukturno je implementiran** dodatnim uslovom i uvođenjem promenljive b u algoritam predstavljen dijagramom toka na slici 2.42. Nestrukturno rešenje moguće je projektovati tako da se "iskače" iz *while* petlje naredbom uslovnog skoka čim se zaključi da je uneti broj $br = 0$, koje, kako je ranije rečeno, nije moguće implementirati na strukturnim jezicima, pa ovde neće biti razmatrano. △

Strukturni algoritmi u odnosu na nestrukturne algoritme imaju jasno izraženu strukturu koja se može opisati sekvencom i ugnežđavanjem. Posledica ovoga je da strukturni algoritmi uglavnom imaju mnogo manji broj mogućih tokova izvršenja kroz algoritam, pa se s tim u vezi može reći da su prednosti strukturnih algoritama u odnosu na nestrukturne

1. jednostavnije praćenje izvršenja i razumevanje semantike, i
2. brže testiranje od strane programskih okruženja za automatsko testiranje programa.

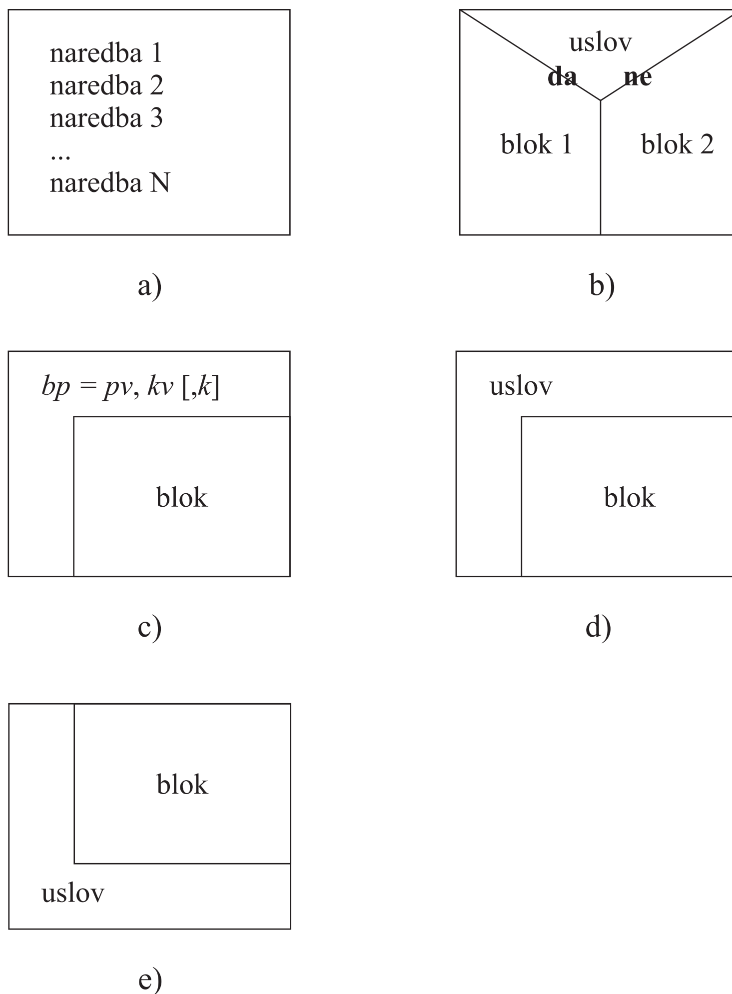
Takođe, prednost strukturnih algoritama je i ta da ih je moguće implementirati na bilo kom programskom jeziku u obliku u kom su dati.

2.4 Strukturogrami

Strukturogrami su način za predstavljanje algoritama koji predstavlja kombinaciju tekstualnog i grafičkog načina predstavljanja algoritama.

Definicija 2.19. Strukturogrami predstavljaju grafičku reprezentaciju algoritama koja se dobija nadovezivanjem i ugnežđavanjem osnovnih grafičkih simbola.

Strukturogramima je moguće predstaviti jedino strukturne algoritme. Mehanizam za predstavljanje nestrukturnih algoritama kod strukturograma ne postoji. Osnovni grafički simboli alternacije, grananja i petlji tipa *for*, *while* i *repeat-until* prikazani su na slici 2.43.

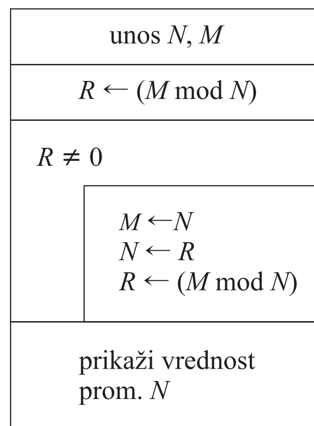


Slika 2.43: Osnovni grafički simboli za predstavljanje algoritama strukturogramima: a) sekvenca, b) alternacija, c) petlja tipa *for*, d) petlja tipa *while*, e) petlja tipa *repeat-until*

Algoritam se može predstaviti strukturogramom nadovezivanjem i ugnežđavanjem osnovnih struktura čija je grafička reprezentacija data na slici 2.43. Reprezentacija algoritma strukturogramom se može koristiti kao dokaz da je algoritam u potpunosti strukturni.

Primer 2.26 (Strukturogram Euklidovog algoritma)**. Zadatak**: Nacrtati strukturogram Euklidovog algoritma.

Rešenje: Euklidov algoritam je tekstualno opisan u primeru 1.6, a strukturni dijagram toka dat je u primeru 2.11 (slika 2.20). Strukturogram sačinjen na osnovu dijagrama toka algoritma sa slike 2.20 prikazan je na slici 2.44. △



Slika 2.44: Strukturogram Euklidovog algoritma

2.5 Pseudokod

Pseudokod je vrsta strukturisanog engleskog ili drugog govornog jezika za opis algoritama, koji omogućava projektantu da se usredsredi na logiku algoritma, bez ulaženja u detalje same sintakse jezika. Istovremeno, opis na pseudokodu treba da bude potpun i da opisuje celu logiku algoritma, tako da implementacija na izabranom programskom jeziku može biti izvršena automatski red-po-red.

Definicija 2.20 (Pseudokod)**.** Pseudokod je neformalni opis operativnih principa algoritma na visokom nivou, koji ima strukturu sličnu strukturi ciljnog programskog jezika.

Reč "neformalni" u definiciji označava da ne postoje precizna pravila za pisanje pseudokoda. Pri opisivanju algoritma na pseudokodu koriste se strukturne konvencije ciljnog programskog jezika, ali je sam opis namenjen za čitanje od strane čoveka, a ne mašine. Pseudokod obično zanemaruje detalje koji nisu od suštinske važnosti za ljudsko razumevanje algoritma, kao što su deklaracije promenljivih (rezervacija memorije za pamćenje vrednosti promenljivih), i sl. Svrha korišćenja pseudokoda je lakše razumevanje semantike algoritma nego što je slučaj sa programskim jezicima, jer se kod pseudokoda ključni principi predstavljaju tekstom bez opterećivanja koda detaljima programskog jezika.

Pseudokod se najčešće koristi u udžbenicima za predstavljanje algoritama i može se reći da je u ovoj vrsti literature pseudokod zastupljeniji od dijagrama toka algoritma. Još jedan razlog koji govori u prilog prethodnom tvrđenju je jednostavnije i brže pisanje pseudokoda u editorima teksta nego crtanje rešenja i ugradnja algoritma u tekst udžbenika u vidu dijagrama toka. S druge strane, dijagram toka je jasniji i pregledniji vid predstavljanja algoritama zbog vizualno jasnijeg načina predstavljanja struktura.

Osnovne strukture koje se mogu predstaviti pseudokodom su takođe sekvenca, alternacija i petlje. Ove strukture se kao i kod ostalih načina za predstavljanje algoritama koriste kao gradivni elementi za predstavljanje složenih algoritamskih struktura.

Sekvenca se u pseudokodu predstavlja kao niz preciznih rečenica ili matematičkih izraza navedenih jedan ispod drugog.

Nekoliko ključnih reči koje se često upotrebljavaju u pseudokodu za označavanje tipične obrade su:

1. Ulaz: READ, OBTAIN, GET,
2. Izlaz: PRINT, DISPLAY, SHOW,
3. Izračunavanje: COMPUTE, CALCULATE, DETERMINE,
4. Inicijalizacija: SET, INIT,
5. Uvećavanje vrednosti za 1: INCREMENT, BUMP,

odnosno na srpskom:

1. Ulaz: UNOS, UČITAJ, PRIBAVI,
2. Izlaz: ŠTAMPAJ, PRIKAŽI,
3. Izračunavanje: IZRAČUNAJ, ODREDI,
4. Inicijalizacija: POSTAVI, INICIJALIZUJ,
5. Uvećavanje vrednosti za 1: INKREMENTIRAJ.

Primer 2.27 (Pseudokod algoritma za zamenu mesta vrednosti dve lokacije)**.**
Zadatak: Napisati algoritam na pseudokodu koji vrši zamenu mesta vrednosti smeštenih u lokacijama A i B. Vrednosti A i B zadaje korisnik. Prikazati vrednosti nakon zamene mesta.

Rešenje: Tekstualni opis algoritma za zamenu mesta vrednosti dveju lokacija dat je u primeru 2.2. Algoritam je moguće predstaviti sekvencom u pseudokodu na sledeći način:

Unos: A, B;
$P = A$;
$A = B$;
$B = P$;
Prikaz: A, B;

Sekvencu čini 5 naredbi, počev od unosa vrednosti u lokacije A i B, preko pomeranja vrednosti preko pomoćne lokacije P, do prikaza sadržaja lokacija. Umesto simbola operatora za dodelu vrednosti '=' moguće je koristiti i ':=', ili $\leftarrow$. $\triangle$

Kao i kod dijagrama toka (slika 2.15), u pseudokodu postoje grananja sa i grananja bez naredbi u "ne" grani, odnosno *if-then-else* i *if-then* grananje. Grananje sa naredbama u "ne" grani u pseudokodu se može predstaviti na sledeći način:

if *uslov* **then**
| *"da" grana*;
else
| *"ne" grana*;
end

Grananje bez *else* dela, odnosno bez naredbi u "ne" grani se može predstaviti na sledeći način:

if *uslov* **then**
| *"da" grana*;
end

Umesto oznake za kraj grananja *end* može se koristi i **endif**, ili **end if**. Kako je pseudokod način predstavljanja algoritma u slobodnoj formi, uz zadržavanje preciznosti i strukture programskog jezika, grananje u pseudokodu može biti predstavljeno i na srpskom na sledeći način:

ako je *uslov* **onda**
"da" grana
u suprotnom
"ne" grana
kraj grananja

Petlje tipa *while*, *repeat-until* i *for* u pseudokodu obično se predstavljaju na sledeći način:

while *uslov* **do**
| *telo petlje*;
end

repeat
| *telo petlje*;
until *uslov*;

for *opis* **do**
| *telo petlje*;
end

Opis u zaglavlju *for* petlje predstavlja tekstualni opis graničnih vrednosti i načina promene vrednosti brojača petlje. Ovde može stajati tekst kao na primer

"za sve vrednosti iz liste", ili, ukoliko je jednostavnije za razumevanje, opis brojača kao kod dijagrama toka $bp = pv, kv, [k]$.

Kao što se iz prethodno navedenih struktura alternacije i svih tipova petlji vidi, svaka osnovna struktura u pseudokodu ima jasno označen početak i kraj. Osnovne strukture se u cilju izgradnje složenijih algoritama mogu nadovezivati i ugnežđavati.

Primer 2.28 (Pseudokod algoritma za određivanje najmanjeg broja u listi)**. Zadatak**: Napisati pseudokod algoritma koji u zadatoj listi brojeva **L** određuje i prikazuje najmanji broj.

Rešenje: Razmotrimo problem na primeru. Neka je zadata lista brojeva

$$\mathbf{L} = \{6, 8, 4, 9, 7, 2, 5, 10\}.$$

Iz primera se jasno vidi da je minimalni broj broj 2. Međutim, algoritam predstavlja precizan postupak koji opisuje kako se do rešenja dolazi, korišćenjem sekvenci, alternacija i petlji, pa se postavlja ključno pitanje: Na koji način "se vidi" da je minimalni broj u datom primeru 2? Odgovor na ovo pitanje je polazna tačka u projektovanju algoritma: potrebno je posmatrati brojeve jedan po jedan sa, na primer, leve strane u desnu i odrediti da li je u međuvremenu pronađen manji broj od onog u koji se trenutno razmatra.

Kako se može proveravati samo po jedan broj u svakom trenutku, pretpostavimo da je prvi broj sa leve strane minimalni, t.j. $min = 6$. Nakon ovoga može se preći na sledeći element

$$\{6, \underline{\mathbf{8}}, 4, 9, 7, 2, 5, 10\}$$

i proveriti da li je manji od onoga što se za sada smatra minimumom, odnosno da li je $8 < min$. Kako ovo nije slučaj, ostaje da je $min = 6$, nakon čega se prelazi na sledeći element

$$\{6, 8, \underline{\mathbf{4}}, 9, 7, 2, 5, 10\}.$$

Uslov je ispunjen za element čija je vrednost 4, pa, uslovno rečeno, treba promeniti mišljenje i usvojiti se da je $min = 4$.

Ovaj postupak potrebno je ponoviti za sledeću vrednost 9, za koju uslov nije ispunjen, kao ni za 7, a nakon toga i za 2, kada će uslov biti ispunjen, kada se usvaja da je $min = 2$. Do kraja liste ostaju vrednosti 5 i 10, obe veće od 2, pa će ostati vrednost $min = 2$.

Prethodno opisani algoritam može se predstaviti pseudokodom na sledeći način:

```
Ulaz: lista L;
min ← prvi element liste L;
for za svaki element L_i iz liste L do
  | if L_i < min then
  |   | min ← L_i;
  | end
end
Prikaz: min;
```

Kao što se iz ovog primera vidi, iako nismo opteretili pseudokod delovima koji bi bili tipični za programski jezik, kao što su inicijalizacija memorijskog prostora za niz vrednosti **L**, indeksiranje niza brojačem petlje i i sl., algoritam je predstavljen na čitljiv način, koji je ujedno veoma blizak programskom jeziku. △

Kontrolna pitanja

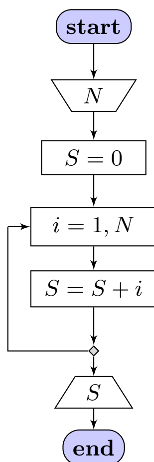
1. Nabrojati i opisati načine za predstavljanje algoritama.
2. Koje su karakteristike tekstualnog načina za predstavljanje algoritama.
3. Neka su $L_i, i = 1, 2, \ldots, N$ elementi niza **L** koji predstavljaju N lokacija u koje je upisan po jedan broj. N je proizvoljan broj koji zadaje korisnik. Dati tekstualni opis na prirodnom jeziku za ciklično pomeranje brojeva za jedno mesto ulevo, tako da vrednost lokacije L_N bude pomerena na lokaciju L_{N-1}, vrednost lokacije L_{N-1} u L_{N-2}, itd. Vrednost lokacije L_1 pomeriti na lokaciju L_N.
4. Nabrojati sve tipove blokova koji se koriste za predstavljanje algoritama dijagramima toka.
5. Problem iz primera 2.13 rešiti upotrebom *repeat-until* petlje.
6. Nacrtati strukturni dijagram toka algoritma i na programskom jeziku C napisati strukturni program koji određuje i prikazuje vrednost π primenom sledećeg iterativnog postupka:

$$\begin{aligned} P_0 &= 3.0 \\ P_{i+1} &= P_i + \sin(P_i), \end{aligned} \tag{2.9}$$

   gde je $i = 0, 1, 2, \ldots$ Izračunavanje prekinuti kada je $|P_{i+1} - P_i| \leq \epsilon$, gde je ϵ zadata tačnost.

7. Isključivo upotrebom petlje tipa *while* nacrtati strukturni dijagram toka algoritma koji određuje i prikazuje faktorijal zadatog broja N. Broj N zadaje korisnik. Smatrati da korisnik zadaje broj N tako da važi $N \geq 1$.

8. Nestrukturni dijagram toka algoritma sa slike 2.38 predstaviti strukturnim ekvivalentom.
9. Nacrtati ekvivalentan dijagram toka algoritma za dijagram toka dat u nastavku, kod koga je petlju tipa *for* linearno transformisana tako da se u telu petlje umesto izraza $S = S + i$ koristi $S = S + 2 \cdot i - 1$, a algoritam i dalje određuje sumu prvih N prirodnih brojeva.



10. Strukturogramom predstaviti algoritam za određivanje faktorijala zadatog broja.
11. Nacrtati strukturogram algoritma koji određuje i prikazuje sve načine na koje je moguće vratiti kusur od N dinara novčanicama u apoenima od 1, 2 i 5 dinara.
12. Na osnovu primera 2.28 u pseudokodu predstaviti algoritam koji u zadatoj listi brojeva **L** određuje i prikazuje najveći broj.
13. Nacrtati strukturni dijagram toka algoritma koji u zadatoj listi brojeva određuje broj koji se pojavljuje najmanji broj puta.
14. Nacrtati strukturni dijagram toka algoritma kojim se za zadati prirodni broj n izračunava i prikazuje suma
$$S = \frac{1!}{\frac{1}{2}} + \frac{2!}{\frac{1}{2} + \frac{1}{3}} + \cdots + \frac{n!}{\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n+1}}.$$
15. Nacrtati strukturni dijagram toka algoritma koji pronalazi i prikazuje N-ti član niza $\mathbb{X}$, čiji se elementi dobijaju sledećim iterativnim potupkom:
$$x_i = x_{i-1} + x_{i-2} + x_{i-3},$$
gde su početne vrednosti elemenata niza $x_0 = 1$, $x_1 = 1$ i $x_2 = 2$. Broj N unosi korisnik sa tastature.

16. Nacrtati strukturni dijagram toka algoritma kojim se za zadati prirodni broj n izračunava i prikazuje suma
$$S = \sum_{i=1}^{n} \frac{n!}{(n-1)!}.$$

17. Nacrtati strukturni dijagram toka algoritma kojim se izračunava i prikazuje vrednost sume:
$$S = 1 + \frac{2}{2!} - \frac{4}{3!} + \frac{8}{4!} - \cdots \pm \frac{2^{i-1}}{i!}.$$

18. Nacrtati strukturni dijagram toka algoritma koji prikazuje sve delioce unetog prirodnog broja N, njihovu sumu, i artimetičku sredinu delilaca. Primer: Za zadato $N = 24$, program treba da prikaže delioce $1, 2, 3, 4, 6, 8, 12, 24$, njihovu sumu 60 i aritmetičku sredinu $60/8 = 7, 5$.

19. Nacrtati strukturni dijagram toka algoritma koji za svaku od m grupa realnih brojeva računa i prikazuje sumu i aritmetičku sredinu brojeva. Na početku programa korisnik zadaje broj grupa, a zatim redom za svaku grupu prvo broj brojeva u toj grupi, a nakon toga i same brojeve. Nakon unosa poslednjeg broja iz svake grupe prikazati sumu i aritmetičku sredinu brojeva te grupe.

20. Nacrtati strukturni dijagram toka algoritma i na programskom jeziku C napisati strukturni program koji od zadatih cifara formira ceo broj. Cifre se zadaju jedna po jedna, počev od cifre najmanje težine. Nakon formiranja broja proveriti da li je formirani broj prost.

3

Osnovni elementi programskog jezika C

Programski jezici, bez obzira da li spadaju u jezike niskog, visokog ili veoma visokog nivoa, zahtevaju pisanje iskaza jezika po strogo definisanoj sintaksi. Ovo je neophodno jer su kompajleri ipak samo računarski programi, koji moraju imati jasno definisan ulaz u vidu izvornog koda kako bi generisali izvršnu verziju programa. Sintaksa jezika opisuje pravilan redosled navođenja elemenata jezika. Postoji nekoliko načina za formalno opisivanje sintaksnih konstrukcija programskih jezika. Ove forme se grubo mogu podeliti na forme koje sintaksu opisuju tekstualno, i forme koje opis sintakse predstavljaju grafički. Najpoznatija forma za opis sintakse jezika iz grupe tekstualnih opisa je Bekus-Naurova forma, a najpoznatija grafička reprezentacija sintakse programskih jezika jesu sintaksni dijagrami.

Na početku ovog poglavlja kratko ćemo kroz par primera predstaviti Bekus-Naurovu formu i sintaksne dijagrame, koje ćemo kasnije u ovom poglavlju koristiti za predstavljanje ispravnih sintaksnih konstrukcija programskog jezika C.

3.1 Formalni opis sintakse programskih jezika

Sintaksne konstrukcije programskih jezika nazivaju se i *gramatikom programskog jezika*. Za opis programskih jezika često se koriste kontekstno-slobodne gramatike. U formalnoj teoriji jezika, kontekstno-slobodna gramatika je gramatika u kojoj se svako gramatičko pravilo može izraziti u obliku

$$V \to w,$$

gde je V simbol koji se definiše, a w je niz definisanih i nedefinisanih simbola, koji se koriste za opis simbola V koji se definiše. Termin "kontekstno-slobodna" predstavlja činjenicu da se V može uvek zameniti sa w bez obzira na kontekst u kome se javlja.

Bekus-Naurova forma (eng. *Backus-Naur Form* - BNF) je konvencija koja se veoma često koristi za zapisivanje pravila kontekstno-slobodnih jezika, kakvi su svi programski jezici, pa i programski jezik C. Proširena Bekus-Naurova forma (eng. *Extended* BNF - EBNF) dodaje određene sintaksičke izraze BNF notaciji i omogućava jednostavniji zapis pravila gramatike.

3.1.1 Bekus-Naurova forma

U BNF se koriste sledeće konvencije za zapisivanje pravila gramatike:

- Umesto simbola $\rightarrow$ koristi se simbol ::=
- Nazivi pomoćnih simbola i simboli koji se definišu se navode među zagradama $\langle$ i $\rangle$
- Vertikalna crta | (izbor) se koristi na desnoj strani pravila i razdvaja moguće opcije koje odgovaraju simbolu sa leve strane. Ukoliko je navedeno više opcija razdvojenih simbolima '|', u konkretnoj upotrebi gramatike bira se jedna od opcija.

Primer 3.1 (Cifra u BNF)**.** Definicija cifre u BNF je

$$\langle cifra \rangle ::= 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \mid 0$$

Pojam koji se definiše je sa leve strane, $\langle cifra \rangle$, a sa desne strane je naveden izbor od 10 mogućnosti, pa po ovoj definiciji cifra može biti $0, 1, \ldots,$ ili 10. Gore navedena definicija čita se: *"cifra može biti 1, ili 2, ili 3, ..., ili 0."* $\triangle$

Primer 3.2 (Neoznačen broj)**.** Koristeći definiciju cifre iz prethodnog primera, neoznačeni broj se u BNF može definisati kao

$$\begin{aligned} \langle neoznacen_ceo_broj \rangle &::= \langle cifra \rangle \mid \langle cifra \rangle \langle neoznacen_ceo_broj \rangle \\ \langle cifra \rangle &::= 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \mid 0 \end{aligned}$$

Navedena definicija glasi: *"neoznačeni broj je samo jedna cifra, ili je cifra koja u nastavku sa svoje desne strane ima neoznačeni broj"*. U ovoj definiciji pojam koji se definiše korišćen je za definiciju samog tog pojma. Ovakva konstrukcija naziva se **rekurzija**, ili rekurzivna definicija.

Po prethodnoj definiciji, jedna mogućnost da se ispravno napiše neoznačeni broj je pisanje bilo koje cifre, ukoliko se uzme prva opcija sa leve strane izbora '|'). Time dobijamo da je neoznačeni broj bilo koji jednocifreni broj.

Ukoliko se posmatra desna strana izbora $\langle cifra \rangle \langle neoznacen_ceo_broj \rangle$, znači da je neoznačeni broj jedna cifra i neoznačeni broj, za koji je prethodno rečeno da može biti jednocifreni broj, što će dati bilo koji dvocifreni broj.

Nastavljajući po istoj logici dalje, neoznačeni broj može biti cifra i dvocifreni broj, što daje sve trocifrene brojeve, itd.

Primer ispravno napisanog neoznačenog broja po gore navedenoj definiciji je 3579 (tri hiljade petsto sedamdeset devet). Pokazaćemo da je ovo ispravno napisan broj koristeći definiciju na sledeći način:

- neoznačen broj 3579 je cifra 3 i neoznačeni broj 579 (po desnoj strani izbora u definiciji);
- neoznačeni broj 579 je cifra 5 i neoznačeni broj 79 (po desnoj strani izbora u definiciji).
- neoznačeni broj 79 je cifra 7 i neoznačeni broj 9 (po desnoj strani izbora u definiciji).
- neoznačeni broj 9 je cifra 9 (po levoj strani izbora u definiciji).

△

Primer 3.3 (Ceo broj)**.** Koristeći definiciju neoznačenih celih brojeva iz prethodnog primera, ceo broj, neoznačeni ili označeni, u BNF se može definisati kao

$$\begin{array}{rcl} \langle \textit{ceo_broj} \rangle & ::= & \langle \textit{neoznacen_ceo_broj} \rangle \\ & & \mid + \langle \textit{neoznacen_ceo_broj} \rangle \\ & & \mid - \langle \textit{neoznacen_ceo_broj} \rangle \end{array}$$

Ova definicija kaže da se ceo broj može pisati kao neoznačeni broj, ili kao neoznačeni broj kome se sa leve strane dodaje znak '+' ili znak '-'. Na primer: 234, +5000, −25, itd. △

Primer 3.4 (BNF za realan broj)**.** Realan broj se može definisati u BNF na sledeći način:

$$\begin{array}{rcl} \langle \textit{realan_broj} \rangle & ::= & \langle \textit{neoznacen_realan_broj} \rangle \\ & & \mid + \langle \textit{neoznacen_realan_broj} \rangle \\ & & \mid - \langle \textit{neoznacen_realan_broj} \rangle \\ \langle \textit{neoznacen_realan_broj} \rangle & ::= & \langle \textit{neoznacen_ceo_broj} \rangle \\ & & \mid \langle \textit{neoznacen_ceo_broj} \rangle . \langle \textit{neoznacen_ceo_broj} \rangle \\ \langle \textit{neoznacen_ceo_broj} \rangle & ::= & \langle cifra \rangle \mid \langle cifra \rangle \langle \textit{neoznacen_ceo_broj} \rangle \\ \langle cifra \rangle & ::= & 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \mid 0 \end{array}$$

Po ovoj definiciji primeri ispravno napisanih realnih brojeva su: 12.05, −5.00, itd. △

3.1.2 Proširena Bekus-Naurova forma

Proširena Bekus-Naurova forma (eng. *Extended Backus-Naur form - EBNF*) ima istu izražajnu moć kao BNF, s tim da su u odnosu na BNF izvršene izmene koje doprinose čitljivosti pravila. EBNF je čitljivija i jednostavnija za upotrebu, jer za razliku od BNF, gde se često koristi rekurzija da bi se izrazile relacije, EBNF koristi **iteraciju**.

Konvencije za EBNF su:

- Pomoćni simboli se zapisuju velikim početnim slovom.
- Završni simboli se zapisuju pod jednostrukim navodnicima ako se sastoje od jednog karaktera, a zadebljanim slovima ako su višeslovni. Npr. '+', **div**, **integer**, itd.
- Oble zagrade (i) se koriste za grupisanje.
- Vitičaste zagrade { i } ograđuju deo koji se ponavlja 0 ili više puta.
- Uglaste zagrade [i] opisuju opcionu konstrukciju (koja može postojati, ali se može i izostaviti).[1]
- Značenje ostalih oznaka je isto kao kod BNF.

Sve konstrukcije date u prethodnim primerima mogu biti izražene i u BNF-u što pokazuje da BNF i EBNF imaju istu izražajnu moć. Sledi nekoliko primera EBNF-a.

Primer 3.5 (EBNF za neoznačen ceo broj)**.**

$$\begin{aligned} NeoznacenCeoBroj \ &::= \ Cifra\ \{Cifra\} \\ Cifra \ &::= \ '1' \mid '2' \mid '3' \mid '4' \mid '5' \mid '6' \mid '7' \mid '8' \mid '9' \mid '0' \end{aligned}$$

Po proširenoj BNF prethodna definicija glasi: *"neoznačeni ceo broj može biri cifra, i 0 ili više cifara iza nje"*. Na ovaj način cifra je definisana bez upotrebe rekurzije kao što je to bio slučaj kod BNF. Definicija cifre je navedena jednom u definiciji neoznačenog broja, tako da se u neoznačenom broju može javiti bar jedna cifra. Vitičaste zagrade sa desne strane definišu da se iza te jedne cifre može javiti 0 ili više cifara. △

Primer 3.6 (EBNF za ceo broj)**.** Uzimajući definicije iz prethodnog primera, ceo broj se u EBNF može difinisati kao

$$CeoBroj \ ::= \ ['+' \mid '-']\ NeoznaceniCeoBroj$$

ili, ukoliko se pravilo iz prethodnog primera za neoznačeni ceo broj ugradi u definiciju, dobijamo pravilo bez da koristimo pomoćni simbol *NeoznaceniCeoBroj*:

$$CeoBroj \ ::= \ ['+' \mid '-']\ Cifra\{Cifra\}$$

Opcije u uglasnim zagradama mogu biti navedene, ili izostavljene, tako da ceo broj predstavlja neoznačeni ceo broj sa eventualnim dodatkom znaka sa leve strane. △

Primer 3.7 (EBNF za realne brojeve)**.**

$$RealanBroj \ ::= \ [\ '+' \mid '-'\ ]\ Cifra\ \{Cifra\}\ '.'\ [Cifra\ \{Cifra\}]$$

Iz definicije realnog broja se vidi da je predznak '+' ili '-' opcioni, kao i razlomljeni deo broja iza decimalne tačke. △

[1] Po nekim konvencijama se koriste sledeći sufiksi: ? umesto [ ], * umesto {}, + sa značenjem "jedno ili više pojavljivanja simbola".

3.1.3 Sintaksni dijagrami

Sintaksni dijagrami opisuju pravila gramatike posredstvom grafova. Imaju istu izražajnu moć kao BNF i EBNF. Konstruišu se na sledeći način:

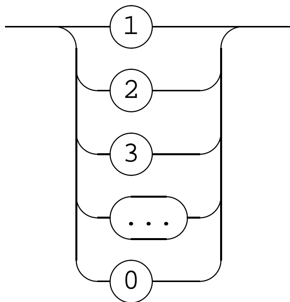
- Svakom pravilu gramatike dodeljuje se po jedan graf koji nosi ime pomoćnog simbola. Ime pomoćnog simbola piše se sa leve strane grafa.
- Čvorovi grafa nose simbole desne strane pravila, a ilustruju se:
  - pravougaonikom ako odgovaraju pomoćnom simbolu gramatike,
  - elipsom, ako predstavljaju završne simbole gramatike.

  Lukovi u grafu povezuju čvorove na način opisan samim gramatičkim pravilom.
- Sintaksno ispravna konstrukcija jezika dobija se obilaskom puteva u grafu krećući se sa leve strane udesno.

Pogledajmo kako izgledaju sintaksni dijagrami nekoliko već pomenutih gramatika.

Primer 3.8 (Sintaksni dijagram za cifru)**.** Sintaksni dijagram koji opisuje cifru prikazan je na slici 3.1. Sintaksno ispravno napisana cifra dobija se krećući se sleva udesno po nekoj od grana sintaksnog dijagrama. Sintaksni dijagram sa slike 3.1 nema povratnih grana, pa se sa leve na desnu stranu može doći preko cifre 1, ili cifra 2, itd. Zbog kompaktnosti prikaza deo cifara od 4 do 9 je izostavljen sa slike 3.1.

Cifra



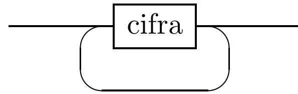
Slika 3.1: Sintaksni dijagram za cifru

△

Primer 3.9 (Sintaksni dijagram za neoznačeni ceo broj)**.** Sintaksni dijagram koji opisuje neoznačeni ceo broj prikazan je na slici 3.2. Krećući se sa leve u desnu

stranu mora se bar jednom preći preko simbola Cifra. Nakon ovoga se može završiti sintaksna konstrukcija, ili se preko povratne grane vratiti i napisati još jedna cifra, itd.

NeoznaceniCeoBroj

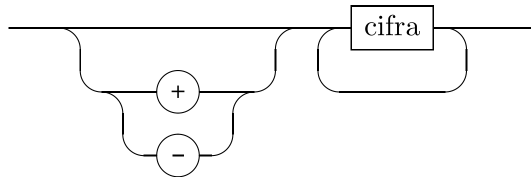


Slika 3.2: Sintaksni dijagram za neoznačeni ceo broj

△

Primer 3.10 (Sintaksni dijagram za ceo broj)**.** Sintaksni dijagram koji opisuje ceo broj prikazan je na slici 3.3. U odnosu na neoznačeni ceo broj iz prethodnog primera, krećući se sleva udesno po ovom sintaksnom dijagramu, može se eventualno preći i preko simbola '+' ili '-'.

CeoBroj

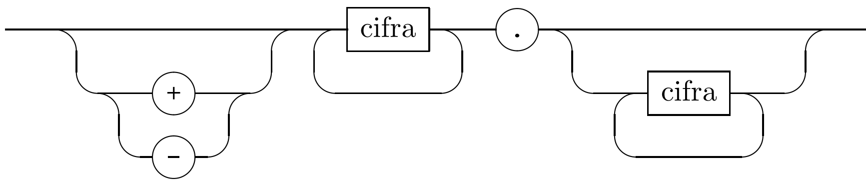


Slika 3.3: Sintaksni dijagram za ceo broj

△

Primer 3.11 (Sintaksni dijagram za realan broj)**.** Sintaksni dijagram koji opisuje realan broj prikazan je na slici 3.4. U odnosu na ceo broj iz prethodnog primera, u ovom sintaksnom dijagramu, krećući se sleva udesno, dodata je decimalna tačka '.' i opciono još jedan neoznačeni broj koji predstavlja razlomljeni deo.

RealanBroj



Slika 3.4: Sintaksni dijagram za realan broj

△

3.2 Programski jezik C

Programski jezik C je programski jezik visokog nivoa, koji spada u grupu proceduralnih programskih jezika.

3.2.1 Razvoj i verzije C kompajlera

Programski jezik C nastao je u periodu između 1969. i 1973. godine. Za tvorca programskog jezika C smatra se Denis Riči (Dennis Ritchie, 1941-2011).

Jezik C je razvijen za potrebe razvoja operativnog sistema UNIX na platformi PDP7[2] u Bell-ovim laboratorijama. Tim programera, Denis Riči, Brajan Kernighan (Brian Kernighan, 1942-) i Ken Tompson (Ken Thompson, 1943-), razvio je programski jezik C da bi operativni sistem UNIX, koji je inicijalno pisan na asembleru za PDP7, preveli za platformu PDP11[3]. Po Denisu Ričiju, godina kada je nastao programski jezik C jeste 1972. godina. Danas je jezik C široko prihvaćen, a na sintaksi jezika C zasnovani su mnogi programski jezici, kao što su C++, C#, Java, PHP, JavaScript, J2ME, i dr.

Programski jezik C nastao je na osnovu programskog jezika B, dok je programski jezik B nastao iz programskog jezika BCPL, pojednostavljivanjem jezičkih konstrukcija. Izmenama jezika B, C je postao jezik sa veoma jednostavnom sintaksom i kratkim sintaksnim konstrukcijama.

Godine 1978. godine Denis Riči i Brajan Kernighan napisali su knjigu *"The C programming language"* kada je C prvi put prikazan širokoj javnosti. Standardizacija programskog jezika C počela je 1983. godine, a 1989. godine je izdata standardizovana verzija C-a poznata kao ANSI-C, ili C89 (ANSI - *American National Standards Institute*). Originalna verzija C-a se često u literaturi označava kao "K&R

[2]PDP7 je mini računar razvijen od strane kompanije DEC (*Digital Equipment Corporation*) 1965. godine.

[3]PDP11 je mini računar koji je kompanija DEC najavila za januar 1970. godine, ali je isporuka krenula u drugoj polovini 1970. Do kraja godine isporučeno je oko 170000 računara.

C", po početnim slovima prezimena autora. Godinu dana kasnije, organizacija za standardizaciju ISO (*International Organization for Standardization*) ratifikovala je isti standad. ISO ratifikacija standarda nosi oznaku godine ratifikacije C90.

Dopuna standarda izvršena je 1999. godine. Ova verzija nosi naziv C99, po godini kada je izvršena dopuna. Bitnije izmene u odnosu na prethodne verzije odnose se na tipove podataka. Još jedna dopuna standarda izvršena je 2011. godine u verziji C11. U ovoj verziji dodata je podrška za konkurentno i paralelno programiranje.

3.2.2 Karakteristike C-a

Programski jezik C je programski jezik visokog nivoa, koji pored ostalog poseduje i operatore bliske jezicima niskog nivoa. Zbog ovoga programi napisani na programskom jeziku C mogu biti brži od programa sa ekvivalentnom funkcionalnošću napisanih na drugim programskim jezicima, a pogodan je i za razvoj širokog spektra aplikacija, počev od razvoja samih operativnih sistema i sistemskog programiranja.

Svojom pojavom programski jezik C skoro je potpuno istisnuo asemblerski jezik. Svi poznatiji operativni sistemi razvijeni su u C-u. Sam kompajler za C na UNIX-u je pisan u C-u.

Programski jezik C po potrebi može biti jezik relativno niskog, ili relativno visokog nivoa, dovoljno je efikasan i dovoljno jezgrovit i izražajan da je pogodniji za upotrebu od mnogih drugih programskih jezika. Sintaksa programskog jezika C je jednostavna, pa je zbog prepoznatljive sintakse ova sintaksa uzeta kao baza mnogih drugih savremenih programskih, proceduralnih i objektno-orjentisanih jezika. Njegova univerzalnost i slobodne forme programiranja čine ga pogodnijim i efikasnijim za široki spektar aplikacija od drugih jezika opšte namene.

Neke od karakteristika programskog jezika C su:

1. Kratke sintaksne konstrukcije - Tim programera koji je razvio jezik C nije imao inicijalnu pretenziju da jezik C bude široko prihvaćen. Motiv je bio što kraći zapis, kako bi pisanje programa bilo brže. Zbog ovoga jezik C u nekim slučajevima može biti izuzetno teško čitljiv.

2. Blizak jezicima niskog nivoa - Određena grupa operatora i naredbi čini jezik C bliskim asemblerskim jezicima. Na primer, ukoliko je potrebno da se dobije na brzini izvršenja, u jeziku C moguće je definisati da se vrednost pamti u registru procesora umesto u operativnoj memoriji.

3. Brzi programi - Programi pisani na programskom jeziku C mogu biti brži u poređenju sa programima pisanim na drugim višim programskim jezicima.

4. Mali broj ključnih reči - Jezik C ima relativno mali broj ključnih reči.

5. Bogat i složen operatorski jezik - Na operatorskom jeziku C-a moguće je veoma kratkim zapisom implementirati relativno složena matematička izračunavanja. Programski jezik C sadrži veliki broj operatora za različite tipove izračunavanja.

6. *Case-sensitive* - Jezik je "osetljiv" (eng. *sensitive*) na velika i mala slova (eng. *case*), pa je važno da li se naredbe, imena promenljivih i drugi elementi jezika pišu malim ili velikim slovima.
7. Slaba tipizacija - Jezik C ima slabe tipove podataka, što znači da se, za razliku od jezika sa jakim tipovima podataka, u okviru istog matematičkog izraza mogu naći promenljive različitog tipa. U ovim slučajevima kompajler automatski vrši transformaciju tipova podataka. Na primer, u Pascal-u nije moguće sabrati ceo broj sa realnim brojem bez prethodnog svođenja na isti tip naredbama za konverziju tipa. Kompajler Pascal-a za ovakve slučajeve javlja sintaksnu grešku. C kompajler automatski razrešava različite tipove podataka.
8. Razvijen sistem funkcija - C poseduje razvijen sistem funkcija. Po sintaksi C-a i glavni program je funkcija, koju poziva operativni sistem i eventualno joj prenosi parametre.

3.2.3 Prevođenje programa

Proces prevođenja programa napisanog na programskom jeziku C prikazan je na slici 3.5. Blok kompajliranja programa na slici 3.5 dat je detaljnije u odnosu na kompajliranje ilustrovano na slici 1.7.

Izvorni program (eng. *source code*) piše se i pamti u tekstualnom fajlu koji obično ima ekstenziju *.c* ili *.cpp*. U ovu svrhu se može koristiti bilo koji editor teksta tipa *Windows*-ovog *Notepad*-a, koji nema napredne funkcije za formatiranje teksta. Integrisana razvojna okruženja obično sadrže i editore.

Proces prevođenja obavlja prevodilac, tako što se program prevodilac pozove i izvrši za izabrani izvorni kod. Ovaj proces je automatski, a pojedini koraci su istaknuti na slici 3.5.

Preprocesiranjem (slika 3.5) eliminišu se makro-naredbe koje program može da sadrži. Izlaz iz ove faze je program na C-u, bez makro-naredbi. Ovakav izvorni kod se uvodi u leksički analizator koji prepoznaje osnovne elemente tipa ključnih reči, operatora, nazive promenljivih, itd. Može se reći da leksički analizator prepoznaje osnovne gradivne elemente i izdvaja ih radi provere da li su poređani onako kako to sintaksa zahteva. Ovi elementi se nazivaju tokeni.

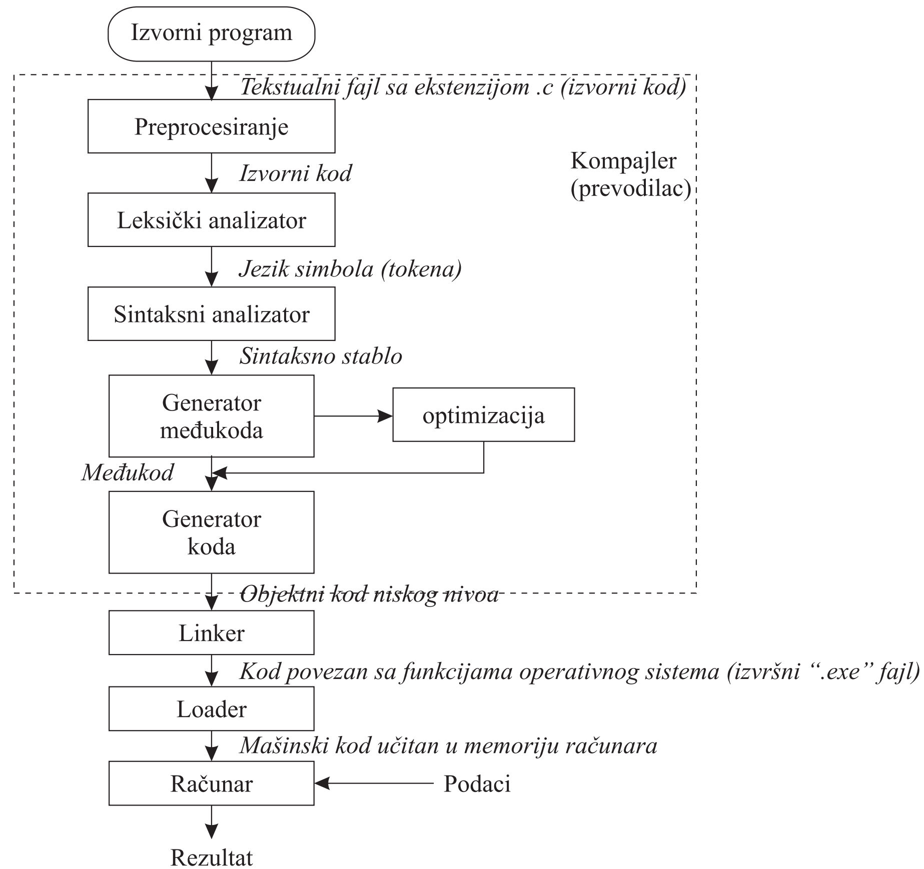
Dobijeni niz tokena se uvodi u sintaksni analizator, koji određuje da li je konstrukcija sintaksno ispravna i generiše među-kod, od koga se dobija kod niskog nivoa. Od izvršnog programa dobijeni kod niskog nivoa deli jedino povezivanje svih objektnih kodova, ukoliko se program sastoji iz više delova (*link*-ovanje).

Primer izlaza iz leksičkog analizatora i ulaza u sintaksni analizator za izraz

$$x = 10;$$

je niz tokena:

promenljiva, znak jednakosti, konstanta, kraj naredbe.



Slika 3.5: Proces prevođenja i izvršavanja programa napisanih na programskom jeziku C

Sintaksni analizator će za ovaj niz odgovoriti da je konstrukcija ispravna i da predstavlja dodelu vrednosti memorijskoj lokaciji koja se interno u programu naziva promenljiva x, i generisaće odgovarajući asemblerski kod.

3.3 Azbuka i tokeni C-a

Program na programskom jeziku piše se povezivanjem osnovnih simbola (karaktera), u koje spadaju slova, cifre i specijalni znaci, u logičke celine. Azbuku programskog jezika C čine karakteri dati u tabeli 3.1.

<table>
<tr><td>Mala slova</td><td>a b c ... z</td></tr>
<tr><td>Velika slova</td><td>A B C ... Z</td></tr>
<tr><td>Cifre</td><td>0 1 2 3 4 5 6 7 8 9</td></tr>
<tr><td>Specijalni karakteri</td><td>+ = _ - ( ) * & % $ # ! < > | . , ; : ” ’ / ? { } [ ] \</td></tr>
<tr><td>Nevidljivi karakteri</td><td>blank (prazno mesto), nova linija, tab (prazna mesta)</td></tr>
</table>

Tabela 3.1: Skup karaktera programskog jezika C

Definicija 3.1 (Token)**.** Token programskog jezika je elementarna celina sačinjena od simbola azbuke koja ima neko značenje u programskom jeziku.

Tokeni mogu biti:

1. ključne reči,
2. identifikatori,
3. separatori,
4. konstante,
5. literali,
6. operatori.

Kao što je prikazano na slici 3.5, kompajler, odmah nakon faze preprocesiranja, prepoznaje elemente programskog jezika (tokene) i proverava u sintaksnom analizatoru da li im je redosled navođenja odgovarajući.

3.3.1 Ključne reči

Definicija 3.2 (Ključne reči)**.** Ključne reči su reči koje u programskom jeziku imaju specijalno značenje.

U ključne reči spadaju naredbe za kontrolu toka izvršenja programa, naredbe za definisanje konstanti, naredbe za definisanje specijalnih delova koda, itd. Jedna od karakteristika programskog jezika C je relativno mali broj ključnih reči. Programski jezik *Ada*, na primer, ima 62 ključne reči, ali to ne znači da su time mogućnosti C-a umanjene. Sve ključne reči programskog jezika C prikazane su u tabeli 3.2.

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Tabela 3.2: Ključne reči programskog jezika C

Napomena: Sve ključne reči u programskom jeziku C pišu se malim slovima. S obzirom na to da je C *case-sensitive* jezik, naredba grananja je **if**, a **IF**, **If** i **iF** nisu među ključnim rečima.

3.3.2 Identifikatori

Definicija 3.3 (Identifikator)**.** Identifikator je sekvenca velikih i malih slova, cifara i karaktera '_' (donja crta), koji nije u skupu ključnih reči i obavezno počinje slovom ili donjom crtom.

Identifikatori se koriste za imenovanje promenljivih u programu, funkcija, simboličkih konstanti, korisničkih tipova podataka i labela.

Definicija 3.4 (Identifikator)**.** U BNF notaciji identifikator se može definisati kao

$$
\begin{aligned}
\langle identifikator\rangle \quad ::= \quad & \langle slovo\rangle \\
& |_\ \langle identifikator\rangle \\
& |\ \langle identifikator\rangle\langle slovo\rangle \\
& |\ \langle identifikator\rangle\langle cifra\rangle \\
& |\ \langle identifikator\rangle_\langle identifikator\rangle \\
& |\ \langle identifikator\rangle_
\end{aligned}
$$

U EBNF notaciji identifikator se može definisati kao

$$Identifikator \quad ::= \quad (Slovo \mid {}'_{}')\{Slovo \mid Cifra \mid {}'_{}'\}$$

Primer 3.12 (Pseudokod za izračunavanje faktorijala broja)**.** Pseudokod algoritma za određivanje faktorijala zadatog celog broja, čiji je dijagram toka prikazan na slici 2.30 je

Ulaz: N;
$F = 1$;
for $i = 2, N$ **do**
| $F = F \cdot i$;
end
Prikaz: F;

Identifikatori u ovom algoritmu su imena promenljivih N, F i i.
△

Na primer, ispravno napisani identifikatori su

x	z_256	_suma	_Iznos
Ime1	prezime	tabela_1	slika_256

a sintaksno neispravno napisani identifikatori su

max + 1	2_main	$krug	235
ime 1	int	for	-zapremina

Iz skupa pogrešno napisanih identifikatora, **max+1** nije identifikator, već izraz, jer sadrži karakter '+', **2_main** počinje cifrom, a **$krug** karakterom '$'; **235** je broj, a ne identifikator; **ime 1** sadrži blank karakter; **int** je ključna reč; **main** je reč rezervisana za sistemsku funkciju, a **-zapremina** je izraz.

Dobra programerska praksa, u cilju povećanja čitljivosti programa, je izabrati mnemoničke identifikatore objekata tako da im je naziv u nekoj vezi za primenom identifikatora (npr. identifikator **povrsina**, koji bi ukazivao na površinu kruga u programu, i sl.).

Kod ranijih verzija kompajlera dužina identifikatora bila je ograničena na 31 znak, ukoliko se identifikator koristi u fajlu u kojem je definisan, i na maksimalno 6 karaktera ukoliko se koristi u eksternim fajlovima. Kod današnjih kompajlera uglavnom ne postoji ograničenje.

3.3.3 Separatori

Definicija 3.5. Separatori su tokeni predstavljeni specijalnim simbolima koji imaju određeno (unapred definisano) sintaksno značenje u programu. Separatori ne označavaju operaciju, već razdvajaju druge tokene u grupe.

Separatori programskog jezika C su:

- { – početak neke programske celine (bloka), ekvivalent je naredbi **begin** (prev. *početak*) u programskom jeziku Pascal.
- } – kraj programske celine (bloka), ekvivalent je naredbi **end** (prev. *kraj*) u programskom jeziku Pascal.
- ; – kraj naredbe. U C-u se ceo program može napisati u jednom redu, a oznaku kraja naredbe ne predstavlja prelazak u novi red, nego separator ';'.
- (– pocetak liste parametara funkcije (ukoliko je u funkciji separatora, ako je u matematičkom izrazu, onda je znak za otvorenu zagradu).
-) – kraj liste parametara funkcije.
- // – linijski komentar, deo koda koji kompajler preskače. Kada kompajler naiđe na dve crte "//" preskočiće i neće pokušati da prevodi red počev od znaka za komentar pa do kraja reda. Koristi se u cilju dokumentovanja programa tako što se dodaju tekstualni opisi zbog kasnijeg lakšeg snalaženja.
- /* */ – blokovski komentar. Početak komentara '/*' i kraj komentara '*/' ne moraju biti u istoj liniji. Ovakvim blokom je moguće napisati komentar koji zahvata više od jednog reda. Obično specijalizovani editori komentare ispisuju zelenom bojom, kako bi se razlikovali od "ostatka" programa.

Primer 3.13 (Program na C-u za određivanje faktorijala broja)**.** Program na programskom jeziku C za određivanje i prikazivanje faktorijala zadatog broja N, sa odgovarajućim komentarima u kodu dat je u nastavku[4].

```
1  /* Ovo je program za odredjivanje faktorijala
2  zadatog broja. Promenljive su:
3  N - ulazni parametar (zadati broj)
4  i - brojac petlje
5  F - prom. u kojoj pamtimo medjurezultate za faktorijal */
6  #include "stdio.h"
7  main()
8  {
9      int N, F, i;       // ovde deklarisemo promenljive
10     scanf("%d",&N);    // unos broja sa tastature
11     F = 1;
12     for (i = 2; i <=N; i++) // for petlja po i od 2 do N
13         F =  F * i;    // prethodnu vrednost mnozimo sa i
14     printf("%d\n",F); // prikaz faktorijala
15 }
```

△

3.3.4 Konstante

Definicija 3.6 (Konstante)**.** Konstante su konkretne vrednosti čija je vrednost napisana direktno u okviru programskog koda.

Konstante mogu biti:

1. celobrojne,
2. realne, i
3. znakovne (karakter konstante).

Celobrojne konstante

BNF notacija celobrojnih konstanti data je u primeru 3.3, EBNF notacija u primeru 3.6, a sintaksni dijagram u primeru 3.10.

Definicija 3.7 (Celobrojna konstanta u C-u)**.** EBNF notacija celobrojne konstante u programskom jeziku C je

$$CeoBroj \quad ::= \quad [\ \mathbf{0x}\ |\ '0'\ ]\ ['+'\ |\ '-']\ Cifra\ \{Cifra\}\ [\ l\ |\ u\ ]$$

[4]Linije programskog koda označene su brojevima sa leve strane. Ovi brojevi nisu sastavni deo programa, već će biti korišćeni za referenciranje iz teksta na određenu liniju koda.

Kao dodatak definiciji celobrojne konstante iz prethodnog poglavlja, u programskom jeziku C celobrojna konstanta može biti bez prefiksa, ili, pored prefiksa za znak, može imati i prefiks **0x**, ili **0**. Takođe, celobrojna konstanta eventualno može imati sufiks **u** ili **l**.

Prefiks konstante u C-u označava brojevni sistem. Konstante bez prefiksa su u dekadnom brojevnom sistemu, prefiks **0x** označava da je broj dat u heksadekadnom brojevnom sistemu, dok prefiks **0** označava oktalni brojevni sistem. Konstanta u C-u napisna kao 010 je zapravo dekadni broj 8, jer prefiks 0 označava da se radi o oktalnom sistemu. Konstanta $0xFF$ je ceo broj 255.

Sufiks **u** ili **l** govori kompajleru kakva memorijska reprezentacija broja treba biti, pa 'u' govori da se radi o neoznačenom celom broju (***unsigned***). Sufiks **l** govori da se radi o broju sa velikim brojem cifara, pa će za ovakav broj biti rezervisano više bajtova nego obično (***long***).

Primer 3.14 (Ilustracija celobrojne konstante)**. Zadatak**: Napisati strukturni program na C-u koji vrednost zadatog celog broja uvećava za 100. Prikazati uvećanu vednost broja.

Rešenje:

```
1 #include "stdio.h"
2 main()
3 {
4     int ZadBr, NoviBr;       // promenljive
5     scanf("%d",&ZadBr);    // unos broja sa tastature
6     NoviBr = ZadBr + 100;  // 100 je celobrojna konstanta
7     printf("%d\n",NoviBr);
8 }
```

Broj 100 u liniji 6 je celobrojna konstanta. △

Realne konstante

Realne konstane su opisane u primerima 3.4, 3.7 i 3.11. Realna konstanta u EBNF notaciji je

$$RealanBroj \quad ::= \quad [\ '+' \ |\ '-'\ ]\ \{Cifra\}\ .\ \{Cifra\}$$

Ovakva konstanta se naziva konstanta u **fiksnom zarezu**. Primer konstante u fiksnom zarezu je 1.0, -5.2, +45.01, itd. Vitičaste zagrade u EBNF označavaju ponavljanje elementa 0 ili više puta, što znači da se celobrojni, ili realni deo konstante mogu izostaviti. Ukoliko se izostave, podrazumeva se vrednost 0. Na primer, konstanta -.5 je ekvivalentna zapisu -0.5, ili +5. je ekvivalentno zapisu +5.0.

Realna konstanta u C-u može biti zadata i u **pokretnom zarezu**. EBNF realne konstante u pokretnom zarezu je

$$\begin{aligned} PokretniZarez &::= [' +' \mid ' -'] \ Mantisa \ (E \mid e) \ Eksponent \\ Mantisa &::= RealanBroj \\ Eksponent &::= CeoBroj \end{aligned}$$

Primer konstante u pokretnom zapisu je -.2E+4, što predstavlja zapis broja

$$-0.2 \cdot 10^{+4}.$$

Kako je u EBNF definiciji naglašeno, svejedno je da li se koristi malo ili veliko slovo *'e'*.

Znakovne konstante

Znakovna konstanta je bilo koji simbol koji se može otkucati na tastaturi ili bilo koji specijalni simbol za posebne namene. Ove konstante se nazivaju i karakteri i pišu se između jednostrukih znakova navoda ('). Na primer: *'a'*, *'A'*, *'9'*, ili *'!'*, itd. U EBNF notaciji karakter konstanta je

$$\begin{aligned} Karakter &::= Slovo \mid Cifra \mid Interpunkcije \mid SpecijalniKarakteri \\ Slovo &::= 'a' \mid 'b' \mid \ldots \mid 'A' \mid \ldots \mid 'Z' \\ Cifra &::= '0' \mid '1' \mid \ldots \mid '9' \\ Interpunkcije &::= '!' \mid '?' \mid '.' \mid ',' \mid \ldots \\ SpecijalniKarakteri &::= '\backslash n' \mid '\backslash t' \mid '\backslash v' \mid '\backslash b' \mid '\backslash f' \mid '\backslash r' \mid '\backslash "' \mid '\backslash '' \mid '\backslash nnn' \end{aligned}$$

Pored simbola koji se mogu naći na tastaturi, postoje i tzv. "nevidljivi", odnosno specijalni karakteri. Na primer, novi red koji se dobija pritiskom na taster *enter* je nevidljivi karakter, za koji bez obzira postoji karakter konstanta, itd. Ovi karakteri imaju svoju funkciju, a konstante koje se koriste za označavanje ovih karaktera se sastoje od dva simbola: specijalnog simbola '\' i slova koje predstavlja oznaku specijalnog karaktera.

Na primer, oznaka "nevidljivog" karaktera za prelazak u novi red je n, ali da bi se ovaj specijalni karakter razlikovao od "običnog" simbola za slovo n ispred oznake se dodaje karakter '\'. Konstanta za specijalni karakter za prelazak u novi red tako postaje '\n'. Karakter '\' koji prethodi oznaci specijalnog karaktera naziva se i *escape* karakter, zato što signalizira kompajleru da "izađe iz uobičajenih okvira" i slovo koje sledi posmatra kao oznaku specijalnog karaktera.

Zbog posebne funkcije koju karakter '\' ima u C-u, konstanta za ovaj upravo ovaj karakter je '\\'. Iz istog razloga konstante za apostrof i znak navoda su ' \' ' i '\" '.

Konstante za specijalne karaktere u C-u su:

- '\n' – prelazak u novi red,

- '\t' – horizontalna tabulacija (taster *ta* na sastaruri),
- '\v' – vertikalna tabulacija,
- '\b' – brisanje karaktera (eng. *backspace*),
- '\f' – nova strana (eng. *page break* ili *form feed*),
- '\r' – povratak na početak reda (eng. *carriage return*)[5],
- '\" ' – navodnik,
- '\' ' – apostrof,
- '\nnn' – trocifreni kod *nnn* bilo kog karaktera iz ACSII tabele (na primer, '\065').

3.3.5 Literali

Definicija 3.8 (Literali)**.** Literali su konstantni nizovi karaktera.

Literali se pišu se između znakova navoda (") i obično se koriste za prikaz rečenica na govornom jeziku. EBNF za literale je:

$$\begin{aligned} Literal &::= \ '\," \,' \{Karakter\} \ '\," \,' \\ Karakter &::= \ Slovo \mid Cifra \mid Interpunkcije \mid SpecijalniKarakteri \end{aligned}$$

Editori razvojnih okruženja obično literale prikazuju crvenom bojom.

Primer 3.15 (Pseudokod algoritma za određivanje parnosti zadatog broja)**.** Pseudokod algoritma za određivanje parnosti broja dat je u nastavku. Da li je broj paran ili ne određuje se ispitivanjem ostatka deljenja zadatog broja sa 2.

Ulaz: N;
if $N \mod 2 = 0$ **then**
    Prikaz: "Broj je paran.";
else
    Prikaz: "Broj je neparan.";
end

Literali u ovom algoritmu su nizovi karaktera *"Broj je paran."* i *"Broj je neparan."*. △

Napomena: Literal se u programu mora završiti u onoj liniji programa u kojoj je započet. Pisanje literala u više redova nije sintaksno ispravno.

Bez obzira što se literal ne može pisati u više redova, problem višerednih poruka je rešiv pomoću specijalnih karaktera. Literal može sadržati i specijalne karaktere. Sledeća poruka bi zbog specijalnog karaktera '\n' bila prikazana u dva reda:

```
"Ovo je prvi red.\nOvo je drugi red"
```

[5] *Carriage Return* (CR) se često koristi kod matričnih štampača u kombinaciji sa prelaskom u novi red.

3.3.6 Operatori i izrazi

Definicija 3.9 (Operator)**.** Operator je simbol koji se koristi za označavanje operacije koje je potrebno izvršiti nad operandima.

Primer operatora je operator sabiranja '+', koji ima dva operanda, operator oduzimanja '-', koji ima dva operanda, itd. Za operatore koji imaju dva operanda kaže se da su *binarni operatori*, a za operatore koji imaju jedan operand kaže se da su *unarni operatori*.

U osnovne operatore C-a spadaju:

- **=** – operator dodele vrednosti; binarni operator kod koga sa leve strane stoji identifikator promenljive kojoj se dodeljuje vrednost, a sa desne konstanta, promenljiva ili izraz čija se vrednost određuje.
- **+,-,*,/,%** – binarni aritmetički operatori za sabiranje, oduzimanje, množenje, deljenje i određivanje ostatka pri deljenju, respektivno.
- **||, &&, ==, !=** – binarni relacioni operatori kojima se formiraju logički izrazi: logičko ILI, logičko I, jednako i različito, respektivno.
- **++, −−** – unarni operatori, pišu se sa leve ili desne strane promenljive i promenljivoj uz koju stoje povećavaju (++), odnosno smanjuju vrednost (−−) za 1. Ukoliko su sa leve strane promenljive, prvo se modifikuje vrednost promenljive pa se nakon toga izvršavaju ostale operacije u izrazu. Ukoliko je operator sa desne strane promenljive, prvo će se izvršiti sve ostale operacije u izrazu pa će se tek tada modifikovati vrednost promenljive. Operator ++ se naziva operator inkrementiranja, a operator −− operator dekrementiranja.

Od operatora, konstanti i promenljivih mogu se formirati izrazi. Malim zagradama '(' i ')' je moguće definisati prioritet izvršavanja operatora u izrazu. Srednje i velike zagrade se ne koriste u izrazima, ali je dozvoljena višestruka upotreba malih zagrada. O podrazumevanim prioritetima i detaljima vezanim za operatore biće više reči u narednom poglavlju. Kao operator za dodelu vrednosti u C-u koristi se simbol '=', a BNF opis izraza dodele vrednosti promenljivoj je

$$
\begin{aligned}
&< dodela_vrednosi > \quad ::= \\
&\quad < promenljiva > \quad = \quad < promenljiva > | < konstanta > | < izraz >;
\end{aligned}
$$

Promenljiva kojoj se dodeljuje vrednost piše se sa leve strane simbola '=', a sa desne strane se može naći konstanta, izraz, ili promenljiva čija se vrednost dodeljuje.

Primer 3.16 (Dodela vrednosti)**.** Neka su x, y i z promenljive u programskom jeziku C. Tada izrazi

$$
\begin{aligned}
x &= 10; \\
y &= x; \\
z &= x + (y * 10) + 4;
\end{aligned}
$$

predstavljaju izraze dodele vrednosti promenljivama x, y i z, redom. Sa leve strane operatora dodele vrednosti nalazi se promenljiva kojoj se dodeljuje vrednost konstante u prvom slučaju. U drugom slučaju se promenljivoj y dodeljuje vrednost promenljive x. U trećem slučaju se promenljivoj z dodeljuje vrednost izraza. Naravno, pre dodele vrednosti računar obavlja sve operacije iz izraza, određuje vrednost izraza i tek nakon toga dodeljuje vrednost.△

3.4 Osnovna struktura C programa

U programskom jeziku C program se piše u okviru bloka čiji početak i kraj označavaju separatori '{' i '}'. Pošto se u programskom jeziku C navedeni separatori mogu koristiti i kao početak i kraj drugih blokova unutar programa (grananja, petlje, itd), kao što će kasnije biti pokazano, program pre separatora '{' koji označava početak programa ima i oznaku "ulazne tačke" od koje kreće izvršenje - *main*() (*eng. main* - glavni).

Primer 3.17. Kompletan i sintaksno ispravan program koji je moguće prevesti C kompajlerom, i koji prilikom pokretanja na izvršenje neće ništa prikazati jer ne sadrži nijednu naredbu između oznake za početak i kraj programa je:

```
1 main()
2 {
3
4 }
```

Na slici 1.10 prikazan je izgled integrisanog radnog okruženja "Microsoft Visual Studio 2010", sa kreiranim projektom (levi deo radnog prostora okruženja), dodatim jednim fajlom sa izvornim programom i unetim programom iz ovog primera (desni deo radnog prostora okruženja). △

Osnovne elemente programa na C-u uočićemo na sledećem primeru.

```
1 main()
2 {
3       printf("Hello␣World!\n");
4 }
```

Ovaj program na ekranu prikazuje poruku *Hello World*[6] na sledeći način:

```
Hello World!
```

[6]"Hello world" (u prevodu "Zdravo, svete") je ustaljena rečenica koja se prikazuje na izlaznom medijumu prilikom pokretanja demonstrativnih programa nekog programskog jezika. Otuda i ime ovakvih programa "*Hello World* programi". Ovaj primer se prvi put pojavio u knjizi "*The C Programming Language*", autora Brajana Kernigana i Denisa Ričija, izdatoj 1978. godine.

U listingu prethodnog programa **main** i **printf** su identifikatori. Identifikatori *main* i *printf* su imena funkcija. Identifikator *main* informiše sistem o tome gde treba započeti izvršavanje programa, a "()" označava da se u ovom pozivu *main* ne koriste dodatni argumenti, koji se inače mogu navesti između zagrada i biće obrađeni kasnije. U C programu obavezna je jedna i samo jedna funkcija *main*, jer program ne može započeti izvršenje sa dva ili više mesta istovremeno.

Identifikator *printf* je identifikator funkcije. Parametar funkcije *printf* je literal koji se navodi u okviru zagrada '(' i ')' i predstavlja tekst koji će biti ispisan na ekranu. Specijalni karakter za prelazak u novu liniju '\n' dodat je kako bi kursor nakon prikaza prešao u novi red, tako da bi program eventualni sledeći prikaz obavio u novoj liniji, odmah ispod tekuće.

Sledeći program u odnosu na prethodni primer uvodi dodatne elemente u osnovnu strukturu programa.

```
1 main()
2 {
3     int razlika;
4
5     razlika = 100 - 90;
6     printf("Razlika␣100-90=%d\n", razlika);
7 }
```

Iskaz iz trećeg reda programa

int razlika;

predstavlja naredbu deklaracije, kojom se u memoriji rezerviše prostor u kom se može zapamtiti jedan ceo broj (*int*). Ovoj memorijskoj lokaciji se dodeljuje identifikator *razlika* kao simboličko ime pomoću koga je moguće kasnije pristupati toj lokaciji. Iskaz u 5 liniji programa (razlika=100-90;) je iskaz kojim se promenljivoj *razlika* dodeljuje vrednost izraza sa desne strane operatora '='.

Funkcija *printf* se koristi u složenijem obliku, sa dva argumenta: literalom **"Razlika 100-90=%d\n"** i promenljivom **razlika**, odvojena zarezom. Literal sadrži specijalnu oznaku '%d'. Funkcija *printf*, analizirajući navedeni literal, pronalazi oznaku '%d', pri čemu karakter 'd' prihvata kao **konverzioni karakter**. Umesto konverzionog karaktera funkcija prikazuje vrednost promenljive koja je navedena iza literala. Funkcija *printf* ima promenljiv broj argumenata. Prvi argument je obavezno literal, koji određuje format prikaza, a za njim slede promenljive odvojene zarezom. Za svaki konverzioni karakter treba da postoji po jedna promenljiva koja će se prikazati na njegovom mestu. Prethodni program prikazuje:

```
Razlika 100-90=10
```

Četvrti red programa ostavljen je prazan zbog čitljivosti. Nepisano pravilo "lepog pisanja" u C-u je pomeranje u desno (uvlačenje tasterom *tab* na tastaturi) svih linija u bloku između separatora { i }. Time se oslikava struktura programa i jasno se vidi koja programska struktura se nalazi u okviru koje strukture. Iz tog

razloga su iskazi u linijama 3, 5 i 6, pomereni udesno u odnosu na separatore u linijama 2 i 7.

Razmaci (*space* i *tab*) između tokena jezika C i prazni redovi ne utiču na proces kompajliranja i mogu se dodavati proizvoljno. Ceo program u C-u je moguće napisati u jednom redu, ali u tom slučaju program ne bi bio čitljiv. Kaže se da programski jezik C nije pozicioni jezik, kao na primer FORTRAN. Program 3.1 je potpuno korektan i sintaksno ispravan C program.

Program 3.1

```
1  main(
2    )  {int v1,
3   v2      ;
4    v1=100; v2=
5   90;
6          printf("Ovaj program je korektan\n"
7  );
8     printf("v1=%d, v2=%d\n",
9     v1,
10    v2)
11    ;}
```

Izlaz programa 3.1

Ovaj program je korektan
v1=100, v2=90

Jedini izuzetak je da se literal, započet u jednom redu, u tom redu mora i završiti (linije 6 i 8 u primeru 3.1). Program 3.2 je isti program kao 3.1, s tim da je napisan čitljivije.

Program 3.2

```
1  /* Ovaj program je pregledniji od
2  programa iz prethodnog primera */
3  main()
4  {
5      int v1,v2;  // deklaracija dve celobrojne promenljive
6
7      v1=100;     // dodela konstante promenljivoj
8      v2=90;      // isto -||-
9      // slede prikazi
10     printf("Ovaj program je korektan\n");
11     printf("v1=%d, v2=%d\n",v1,v2);
12 }
```

Definicija 3.10 (Struktura C programa)**.** Struktura C programa je sledeća:

$$
\begin{aligned}
Program \quad ::= \quad & [\ Preprocesorske_direktive\ ] \\
& [\ Korisnicke_funkcije\ ] \\
& [\ Promenljive\ ] \\
& [\ Konstante\ ] \\
& [\ Tipovi\ ] \\
& \mathbf{main}() \\
& \{ \\
& \quad TeloPrograma \\
& \}
\end{aligned}
$$

Pre identifikarora *main* se eventualno mogu naći i drugi elementi kao što su preprocesorske direktive, korisničke funkcije, promenljive i konstante, i novi korisnički tipovi podataka, o čemu će biti reči u kasnijim poglavljima.

3.5 Deklaracija promenljivih i konstanti

Kako su algoritmi uglavnom neopterećeni detaljima same implementacije, kod algoritama nije potrebno voditi računa o načinu smeštanja podataka i rezervaciji prostora u memoriji računara. Pojednostavljeno rečeno, u trenutku kada je potrebna promenljiva, u algoritmu se jednostavno navede njeno simboličko ime, bez prethodne deklaracije. Kod nekih programskih jezika moguće je koristiti promenljive baš na ovaj način. Naime, kada se javi potreba za pamćenjem nekog podatka programer navodi simboličko ime, što je znak kompajleru da odvoji memorijski prostor za tu promenljivu. Kompajler na osnovu konteksta u kom se javlja simboličko ime u takvim slučajevima, odlučuje o tipu podatka i organizuje memorijski prostor. Ovo nije slučaj sa programskim jezikom C. U C-u je neophodno izvršiti eksplicitnu deklaraciju svih promenljivih koje se koriste u programu.

Deklaracija promenljvih

Programski jezik C spada u grupu jezika koji zahtevaju eksplicitnu deklaraciju promenljivih pre njihovog korišćenja. Kao što je u definiciji 2.4 rečeno (strana 34), u kontekstu programiranja, promenljiva je **simboličko ime** memorijske lokacije, koje se uvodi zbog pojednostavljenja pristupa memorijskoj lokaciji.

Definicija 3.11 (Deklaracija)**.** Deklaracija je iskaz kojim se vrši rezervacija memorijskog prostora i vezivanje identifikatora u vidu simboličkog imena za rezervisanu lokaciju.

Postoje statička i dinamička rezervacija (alokacija) memorijskog prostora.

Definicija 3.12 (Statička alokacija)**.** Statička alokacija memorijskog prostora je alokacija kod koje se u toku pisanja programa tačno definišu podaci i veličina potrebnog prostora, koji u toku izvršenja programa nije moguće menjati.

Definicija 3.13 (Dinamička alokacija)**.** Dinamička alokacija memorijskog prostora je alokacija kod koje se zauzeće prostora i veličina zauzetog prostora mogu menjati u toku izvršenja programa.

Deklaracijom se definiše tip podataka koji će se čuvati u rezervisanom prostoru i simboličko ime koje će se koristiti za pristup. Ovi podaci se ne mogu menjati u toku izvršenja programa. Od navedenog tipa zavisi veličina rezervisanog prostora i memorijska reprezentacija vrednosti (jedinični, dvojični komplement, IEEE754, i sl.).

EBNF notacija statičke deklaracije promenljive je

$$\begin{aligned} \langle deklaracija \rangle \quad ::= & \\ & \langle tip \rangle\ \langle identifikator \rangle\ [\text{'='}\ \langle konstanta \rangle] \\ & [\{\ \text{','}\ \langle tip \rangle\ \langle identifikator \rangle\ [\text{'='}\ \langle konstanta \rangle]\}]\ ; \\ \langle tip \rangle \quad ::= & \ \mathbf{int} \mid \mathbf{float} \mid \mathbf{char} \end{aligned}$$

gde su *int*, *float* i *char* osnovni tipovi podataka za pamćenje celih, realnih brojeva i karaktera, respektivno. Drugim rečima, deklaracija na početku ima tip promenljive, koji može biti *int*, *float*, *char*, iza čega sledi identifikator (naziv promenljive) i opciono znak '=' sa inicijalnom vrednošću zadatom u vidu konstante. Opciono, kako je navedeno u EBNF opisu statičke deklaracije, jedna naredba deklaracije može obuhvatiti i više promenljivih. U ovom slučaju se pojedinačne deklaracije odvajaju zarezom. Ukoliko deklaracija sadrži više od jedne promenljive, sve promenljive u okviru te deklaracije su istog tipa.

Program 3.3 ilustruje moguće načine deklaracije promenljive na osnovu prethodno date EBNF notacije.

Program 3.3

```
1  main()
2  {
3      int x;
4      int y,z;
5      int k=6;
6      float m,n=-.5,p;
7      char S='a';
8
9      printf("vrednost␣prom.␣k␣je␣%d.\n",k);
10 }
```

Izlaz programa 3.3

vrednost prom. k je 6.

U liniji 3 programa 3.3 statički je deklarisana promenljiva x kao celobrojna promenljiva. U jednom deklaracionom iskazu moguće je deklarisati proizvoljan broj promenljivih, što je ilustrovano u liniji 4, a dato kao opcioni deo (zagrade []) koji se može ponavljati 0 ili više puta (zagrade {}) u EBNF notaciji. U okviru deklaracije moguće je dodeliti promenljivoj inicijalnu vrednost. Ovo je ilustrovano u linijama 5, 6 i 7. U 7. liniji je karakter promenljivoj S dodeljena znakovna (karakter) konstanta *'a'*.

Zauzimanje i oslobađanje memorije u toku izvršenja programa, odnosno dinamička alokacija, moguća je pozivom funkcija za dinamičku alokaciju programskog jezika C, koje će biti obrađen u poglavlju 6.6.3.

Definicija 3.14 (Inicijalizacija prilikom deklaracije)**.** Inicijalizacija prilikom deklaracije je dodela vrednosti promenljivoj u iskazu u kom se promenljiva deklariše, što ima za posledicu upis inicijalne vrednosti u promenljivu odmah nakon rezervacije memorijskog prostora za nju.

Na osnovu primera programa 3.3 vidi se da se eksplicitna deklaracija promenljivih vrši u okviru samog bloka programa *main* i da ne postoji poseban blok za deklaraciju.

U zavisnosti od pozicije gde se deklaracija u programu može naći, postoji mala razlika u odnosu na verzije kompajlera ANSI C i C99. **ANSI C** kompajler zahteva da deklaracije budu navedene na samom početku bloka, pre prve izvršne naredbe programa. **C99** i noviji kompajleri dozvoljavaju da deklaracije budu u bilo kom delu programa, uključujući i u okviru grananja i petlji. Kod C99 kompajlera, ukoliko je promenljiva deklarisana u petlji, promenljiva će postojati u memoriji samo dok traje izvršenje te petlje, nakon čega se memorija oslobađa, bez obzira da li se program završio u celosti ili ne.

Napomena: U programskom okruženju *Microsoft VisualStudio* dostupna su oba kompajlera. Ukoliko je izvorni kod snimljen u fajlu sa ekstenzijom *.c* biće pozvan ANSI C kompajler, a ukoliko je ekstenzija *.cpp* biće pozvan C99/C++ kompajler.

Program 3.4 je sintaksno ispravan u C99 kompajleru, ali je sintaksno neispravan u ANSI C kompajleru, jer postoji deklaracija i nakon prve izvršne naredbe koja je navedena u 6. redu.

Program 3.4

```
1  main()
2  {
3      int x;
4      int y,z;
5
6      x = 10;
7
8      int k=6;
9      printf("vrednost prom. k je %d.\n",k);
10 }
```

Deklaracija konstanti

U C-u je pored promenljivih moguće deklarisati i konstante i dodeliti im simboličko ime. Za razliku od promenljivih, za konstante se ne odvaja memorijski prostor, već kompajler na svim mestima u izvornom programskom kodu na kojima se nalazi simboličko ime konstante direktno postavlja njenu vrednost. Konstantama u okviru programa nije moguće dodeliti vrednost.

EBNF deklaracije konstante je

$$\langle deklaracija_konstante\rangle \ ::= \ \textbf{const}\ \langle tip\rangle\ \langle identifikator\rangle = \langle konstanta\rangle;$$
$$\langle tip\rangle \ ::= \ \textbf{int} \mid \textbf{float} \mid \textbf{char}$$

Vrednost iza simbola '=' je obavezna. Na primeru programa 3.5 ilustrovana je upotreba konstanti sa simboličkim imenom.

Program 3.5

```
1 main()
2 {
3     const float Pi=3.1415;
4     float R = 5., Obim;
5     Obim = 2*R*Pi;
6     printf("O=%f\n",Obim);
7 }
```

Iskaz sa funkcijom *printf* u 6. redu listinga programa 3.5 sadrži konverzioni karakter '%f' za prikaz realnih brojeva.

3.6 Standardni ulaz i izlaz

Promenljive u programu mogu dobiti vrednost na jedan od dva načina:

1. dodelom vrednosti konstante, promenljive, ili izraza,
2. unosom sa tastature ili nekog drugog spoljašnjeg uređaja.

Dodelu konstante ili vrednosti izraza promenljivoj moguće je izvršiti pomoću operatora dodele vrednosti '='. Unos sa tastature ili nekog drugog spoljašnjeg uređaja obično se naziva *ulaz*, dok se prikaz, štampanje rezultata i sl. naziva *izlaz*. Programski jezik C nema operatore ni naredbe u vidu ključnih reči za ulaz i izlaz.

Da bi se koristili ulaz i izlaz potrebno je preprocesorskim direktivama uključiti dodatne biblioteke funkcija koje sadrže funkcije za ulaz i izlaz.

C kompajler dolazi sa velikim brojem standardnih biblioteka. Svaka biblioteka je poseban fajl na disku sa ekstenzijom *.h* (eng. *header file*, prev. *header* - zaglavlje), koji sadrži veliki broj dodatnih funkcija koje mogu značajno proširiti upotrebljivost

jezika. Neke od ovih biblioteka sadrže funkcije za ulaz i izlaz, neke sadrže dodatne matematičke funkcije, a postoje i biblioteke koje sadrže funkcije za iscrtavanje tačaka i geometrijskih oblika na ekranu, za kontrolu zvučne kartice, slanje paketa preko računarske mreže, itd.

Da bi se u proces kompajliranja programa uključio neki "header fajl" koristi se preprocesorska direktiva **#include**, uz navođenje imena fajla. Preprocesorske direktive se navode na samom početku programa, pre identifikatora *main*, kako je opisano definicijom 3.10 (def. strukture C programa na strani 104).

"Header fajl" koji sadrži funkcije za ulaz i izlaz je **stdio.h**. Naziv biblioteke *"stdio"* je skraćenica od engleskih reči **ST**andard **I**nput **O**utput, ili u prevodu, standardni ulaz i izlaz. Naziv header fajla se može pisati između znakova navoda (" ") ili između znakova < i >.

Napomena: Prethodni primeri kod kojih je korišćena funkcija *printf* prilikom kompajliranja će javiti sintaksnu grešku da je funkcija *printf* nepoznata, sve dok se pre identifikatora *main* ne napiše preprocesorska direktiva **#include "stdio.h"**.

Pored funkcija za ulaz sa tastature i prikaz vrednosti na monitoru računara, *stdio.h* sadrži i funkcije pomoću kojih je moguće poslati vrednost promenljive na serijski ili paralelni port računara. Za slučaj paralelnog porta, svaki pin porta će odgovarati jednoj binarnoj vrednosti promenljive koja se šalje na port, a vrednost bi se mogla izmeriti voltmetrom (+5V ili 0V). Ovo daje mogućnost upravljanja drugih uređaja pomoću PC računara. Takođe su dostupne i funkcije za očitavanje statusa portova, kojima se može dobiti povratna informacija o statusu kontrolisanog uređaja. Sem funkcija za standardni ulazi i izlaz, ostale funkcije nisu obuhvaćene ovim udžbenikom.

3.6.1 Standardni ulaz

Sintaksa iskaza kojim se poziva funkcija za unos podataka sa standardnog ulaza je

$$\langle Unos \rangle \quad ::= \quad \textbf{scanf}\,(\langle format \rangle, \langle lista_adresa_promenljivih \rangle)\,;$$

Funkcija *scanf* je blokirajuća funkcija. To značu da će se prilikom izvršenja funkcije *scanf* program blokirati i ostati na toj liniji koda sve dok korisnik ne unese sve šta se zahteva formatom, ma koliko vremena korisniku treba za unos. Tek nakon unosa, program će nastaviti izvršenje programa dalje.

Format je literal kod koga se između znakova navoda (") piše onoliko pojedinačnih formata koliko je promenljivih potrebno učitati jednim *scanf* iskazom. Format govori kompajleru da li sa tastature očekuje unos celobrojne, realne ili karakter konstante i na koji način. Format svake promenljive se u okviru literala $\langle format \rangle$ opisuje **pojedinačnom ulaznom konverzijom**. Pojedinačna, zbog toga što svaka promenljiva u listi ima svoj format unosa, a "konverzija" zato što ova oznaka govori kompajleru kako da interpretira (konvertuje) niz karaktera koji korisnik unosi sa tastature: celobrojni podatak, znakovni podatak, i sl.

Definicija 3.15 (Pojedinačna ulazna konverzija)**.** Sintaksa pojedinačne ulazne konverzije je

$$\langle pojedinacna_ulazna_konverzija \rangle ::= \%[\langle w \rangle][\ \mathbf{h} \mid \mathbf{l} \mid \mathbf{L}\ ]\langle tip_konverzije \rangle$$

$$\langle w \rangle ::= \langle ceo_broj \rangle$$
$$\langle tip_konverzije \rangle ::= \mathbf{d} \mid \mathbf{u} \mid \mathbf{o} \mid \mathbf{x} \mid \mathbf{i} \mid \mathbf{f} \mid \mathbf{e} \mid \mathbf{g} \mid \mathbf{c}$$

Prethodna definicija opisana rečima glasi: pojedinačna ulazna konverzija, koja se nalazi u okviru literala formata unosa, počinje znakom '%', nakon koga opciono sledi ceo broj ($\langle w \rangle$), nakon koga opciono sledi jedno od slova h, l ili L. Na kraju konverzije nalazi se tip konverzije koji je jedno slovo iz skupa d, u, o, x, i, f, e, g, ili c.

Primer 3.18 (Sintaksno ispravno pisanje ulaznog formata)**.** S obzirom da su ceo broj $\langle w \rangle$ i slova h, l i L opciona, pojedinačna ulazna konverzija može se po definiciji napisati na sledeći način:

```
%d
%f
```

Dodavanjem opcionih elemenata dobijaju su sledeći zapisi:

```
%4d
%8hf
```

Primer literala koji sadrži jednu pojedinačnu ulaznu konverziju je

```
"%d",
```

a primer literala sa tri pojedinačne ulazne konverzije je

```
"%d%f%d".
```

Naravno, ukoliko format sadrži literal sa tri pojedinačne ulazne konverzije, u listi adresa promenljivih iskaza *scanf* moraju se naći adrese tri promenljive, kojima će se dodeliti vrednosti otkucane na tastaturi.
$\triangle$

Opcioni celobrojni parametar označen sa $\langle w \rangle$ u definiciji označava maksimalni dozvoljeni broj znakova za podatak. Na primer, pojedinačna konverzija "%4d" označava da će korisnik uneti broj ne duži od 4 cifre.

Prefiks **h** se koristi ispred celobrojnih konverzija i označava da se vrši konverzija ulaznog znakovnog niza u kratki ceo broj koji u memoriji zauzima 2 bajta, za razliku od celih brojeva koji zauzimaju 4B. Prefiks **l** se koristi ispred celobrojnih i realnih

tipova konverzija i označava da se vrši konverzija ulaznog znakovnog niza u dugi ceo broj od 8 bajtova, ili u realni broj u dvostrukoj tačnosti, a prefiks L se koristi samo ispred realnih tipova i označava da se vrši konvertovanje ulaznog znakovnog niza u binarni podatak koji zauzima duplo više mesta od brojeva sa dvostrukom preciznošću.

Tipovi konverzije označavaju kog tipa je podatak koji se unosi, jer ceo broj otkucan na tastaturi kao 20 može se protumačiti kao 20, ili 16, ili 32, u zavisnosti od toga da li se broj 20 čita kao dekadni, oktalni, ili heksadekadni broj. Tipovi konverzije su:

- d - očekuje se unos dekadnog broja
- u - očekuje se unos neoznačenog celog dekadnog broja
- o - očekuje se unos oktalnog celog broja
- x - očekuje se unos heksadecimalnog celog broja
- i - očekuje se unos celog broja određenog načinom na koji ga je korisnik uneo broj, po definiciji 3.7 (strana 96.). Ako korisnik unese broj koji ima vodeću 0, npr. 020, broj će se uzeti kao broj u oktalnom sistemu, a ukoliko unese npr. 0x20 uzeće se kao heksadekadni.
- f - očekuje se unos realnog broja u fiksnom zarezu
- e - očekuje se unos realnog broja u pokretnom zarezu
- g - očekuje se unos realnog broja u fiksnom ili pokretnom zarezu u zavisnosti od formata unetog broja
- c - očekuje se unos jednog karakter podatka sa tastature
- s - očekuje se unos stringa (više karaktera).

Primer 3.19 (Unos brojeva sa tastature)**. Zadatak**: Nacrtati dijagram toka i napisati program na programskom jeziku C koji od korisnika zahteva unos dva cela i jednog realnog broja i upisuje ih u odgovarajuće promenljive. Smatrati da je jedna od celobrojnih promenljivih maksimalno četvorocifreni broj.

Rešenje:

```
1 #include "stdio.h"
2 main()
3 {
4     int N,broj2;
5     float x;
6     scanf("%d%4d%f",&N,&broj2,&x)
7 }
```

Po sintaksi funkcije za unos *scanf*, funkcija iza formata zahteva onoliko **adresa** promenljivih koliko pojedinačnih ulaznih konverzija format sadrži. Adresa promenljive dobija se tako što se ispred identifikatora promenljive dodaje unarni operator ’&’, na način prikazan u 6. liniji primera. △

Definicija 3.16 (Referenca)**.** Referenca je unarni operator C-a koji može da stoji sa leve strane identifikatora promenljive i vraća adresu memorijske lokacije koja je dodeljena promenljivoj. Operator referenciranja je '&'.

Razlog za navođenje reference na adresu na kojoj se promenljiva nalazi, a ne same vrednosti promenljive biće razmotren u kasnijim poglavljima.

Primer 3.20. Sledeći program ispisuje adresu memorije u heksadekadnom formatu ("%x") na kojoj se nalazi promenljiva p (&p), kao i samu vrednost promenljive p u dekadnom formatu ("%d").

```
1 #include "stdio.h"
2 main()
3 {
4     int p=50;
5     printf("%x%d",&p,p)
6 }
```

Potrebno je napomenuti da će se adresa koju prikazuje program u ovom primeru razlikovati prilikom svakog ponovnog pokretanja programa, jer je mala verovatnoća da operativni sistem pri ponovnom pokretanju programa dodeli istu adresu istoj promenljivoj, naročito ako je vremenski period između pokretanja programa dug. △

3.6.2 Standardni izlaz

Sintaksa iskaza kojim se poziva funkcija iz biblioteke *stdio.h* za prikaz podataka na standardnom izlazu (na monitoru) je

$$\langle prikaz \rangle \quad ::= \quad \textbf{printf}\ (\langle format \rangle\ [, \langle lista_parametara \rangle])\ ;$$

$$\langle lista_parametara \rangle \quad ::= \quad \{\langle promenljiva \rangle | \langle konstanta \rangle | \langle izraz \rangle\}$$

Format je literal koji se prikazuje. Format može, ali i ne mora da sadrži pojedinačne izlazne konverzije. Ova funkcija prikazuje literal, a ukoliko literal sadrži pojedinačne izlazne konverzije umesto ovih konverzija se na tim mestima u literalu prikazuju vrednosti parametara iz liste. Lista može da sadrži promenljive, konstante i izraze razdvojene zarezom. Način prikaza vrednosti određuje navedena pojedinačna izlazna konverzija. Broj pojedinačnih konverzija treba da bude jednak (mada može da bude i manji), broju parametara u listi. Umesto prve konverzije ispisaće se vrednost prvog parametra iz liste, umesto druge konverzije vrednost drugog parametra, itd. Ukoliko format ne sadrži pojedinačne konverzije, lista parametara se ne navodi, što je i razlog za pisanje ove liste u formi opcionog parametra funkcije *printf* u gore navedenoj definiciji (zagrade '[' i ']').

Napomena: Funkcija *printf* zahteva navođenje imena promenljivih u listi, za razliku od funkcije *scanf*, koja zahteva reference na promenljive. Lista je niz identifikatora promenljivih, konstanti i izraza odvojenih zarezom.

Pojedinačna izlazna konverzija, kao i pojedinačna ulazna konverzija počinje karakterom '%', može imati opcione parametre, nakon kojih sledi jedan od konverzionih karaktera d, u, o, x, i, f, e, g, c i s, sa istim značenjem kao kod ulaza.

Definicija 3.17 (Pojedinačna izlazna konverzija)**.** Sintaksa pojedinačne izlazne konverzije je

$$\langle poj._izl._konv.\rangle ::= \%[-][+][\#][\langle w\rangle[.\langle d\rangle]][\mathbf{h} \mid \mathbf{l} \mid \mathbf{L}]\langle tip_konv.\rangle$$

$$\langle w\rangle ::= \langle ceo_broj\rangle$$

$$\langle d\rangle ::= \langle ceo_broj\rangle$$

$$\langle tip_konv.\rangle ::= \mathbf{d} \mid \mathbf{u} \mid \mathbf{o} \mid \mathbf{x} \mid \mathbf{i} \mid \mathbf{f} \mid \mathbf{e} \mid \mathbf{g} \mid \mathbf{c}$$

Značenje opcionih parametara '-', '+', '#', $\langle w\rangle$ i $\langle d\rangle$ je sledeće:

- '−' - ukoliko je naveden, ispis se vrši tako da je vrednost poravnata uz levu ivicu dela predviđenog za prikaz; ukoliko nije naveden, ispis se vrši uz desno poravnanje u odnosu na broj mesta predviđenih za ispis (npr. "%6d" će dvocifreni broj ispisati sa desnim poravnanjem, tako što će prvo ispisati 4 blanko karaktera, a zatim i dve cifre broja);
- '+' - prikaz znaka vrednosti i u slučaju pozitivnih brojeva, ukoliko se ne navede za pozitivne brojeve se ne ispisuje prefiks '+';
- '#' - ispisuje oznaku brojevnog sistema za celobrojne promenljive (vodeću nulu za oktalne, 0x za heksadekadne);
- $\langle w\rangle$ - označava maksimalan broj karaktera predviđen za prikaz, uključujući celi i razlomljeni deo broja, decimalnu tačku i znak;
- $\langle d\rangle$ - samo za realne brojeve, označava broj mesta za prikaz decimalnog dela broja. Ukoliko se ne navede realni brojevi u fiksnom zarezu se prikazuju na desetak i više decimala (zavisi od kompajlera).

Literal koji opisuje format prikaza može da sadrži specijalne karaktere poput karaktera '\n' (novi red), '\t' (prazan prostor *tab*, tabulacija), itd.

Primer 3.21. Zadatak: Napisati program na C-u koji:
a) prikazuje vrednosti celobrojnih promenljivih a i b, konstante 30, kao i zbir promenljivih $a + b$. Promenljivama prethodno dodeliti proizvoljne vrednosti. Brojeve prikazati jedan ispod drugog. Za ispis svakog broja predvideti 5 mesta. Sve vrednosti, sem zbira poravnati uz desnu ivicu i prikazati znak broja. Prilikom prikaza

zbir $a + b$ poravnati levo.
b) prikazuje vrednosti 2 realne promenljive, jednu ispod druge, tako da se za svaku promenljivu prikazuju samo dve decimale, a ukupan broj mesta predviđen za ispis je 10.
c) prikazuje vrednosti dve karakter promenljive, u istom redu, odvojene blanko znakom. Promenljivama dodeliti proizvoljne vrednosti.
d) prikazuje vrednosti dve karakter promenljive, u istom redu, odvojene zarezom. Promenljivama dodeliti proizvoljne vrednosti.
e) vrednost izraza a+b, u obliku rečenice "Vrednost izraza ... je ...".

Rešenje:

```
1  #include "stdio.h"
2  main()
3  {
4      int a=100,b=-4;
5      float x,y;
6      char s,r;
7      printf("%+5d\n%+5d\n%+5d\n%-+5d",a,b,30,a+b);  // prikaz pod a)
8      x=3.1415;
9      y=-123.1;
10     printf("%10.2f\n%10.2f\n",x,y);    // prikaz pod b);
11     s = 'A';
12     r = 'h';
13     printf("%c␣%c\n",s,r);    // prikaz pod c);
14     printf("%c,%c\n",s,r);    // prikaz pod d);
15     printf("Vrednost␣izraza␣a+b,␣gde␣je␣a=%d,␣a␣b=%d,␣je␣%d",a,b,a+b); // pod e)
16 }
```

Izlaz iz programa prikazan je na slici 3.6.

```
C:\Users\vciric\documents\visual studio 2010\Projects\AIP\Debug\AIP.exe
 +100
   -4
  +30
+96        3.14
   -123.10
A h
A,h
Vrednost izraza a+b, gde je a=100, a b=-4, je 96
```

Slika 3.6: Primer upotrebe funkcije *printf*

△

3.7 Osnovne algoritamske strukture C-a

Osnovne algoritamske strukture koje se mogu implementirati u C-u su:

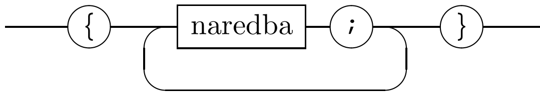
1. sekvenca,
2. alternacija, i
3. petlje.

Sekvenca se u programskom jeziku implementira *blokom naredbi*.

Definicija 3.18 (Blok naredbi C-a)**.** Blok naredbi je programska struktura koja implementira algoritamsku strukturu sekvence, kod koje se naredbe izvršavaju sukcesivno tako da naredna naredba započinje izvršenje tek nakon okončanja izvršenja prethodne naredbe.

Blokom naredbi vrši se logičko grupisanje naredbi. Naredbe u bloku naredbi se međusobno odvajaju znakom ';', a mogu se zbog čitljivosti pisati svaka naredba u posebnom redu. Blok naredbi počinje znakom '{' i završava se znakom '}'. Sintaksni dijagram bloka naredbi prikazan je na slici 3.7.

BlokNaredbi



Slika 3.7: Sintaksni dijagram bloka naredbi

Ukoliko se program prevodi C99 ili novijim kompajlerom, u okviru svakog bloka je moguće deklarisati promenljive. Za ove promenljive će se rezervisati mesto u memoriji i postojaće samo dok traje izvršenje bloka. Kada se završi i poslednja naredba bloka, mesto u memoriji rezervisano za ove promenljive se oslobađa, a pristup promenljivoj van bloka je sintaksno neispravan. Ovakve promenljive se nazivaju lokalnim promenljivama za blok.

Sledeći program ilustruje grupisanje naredbi u blok naredbi.

Program 3.6

```
1  main()
2  {
3      int x,y;
4      int z = 10;
5      {
6          int a,b;
7          a = 5;
8          b = a + z;
9      }
10     z = 5;
11 }
```

Naredbe 6, 7 i 8 grupisane su u blok koji počinje vitičastom zagradom u liniji 5, a završava se u liniji 9. Potrebno je napomenuti da u ovom primeru uvođenje bloka suštinski ne menja izvršenje programa, već je blok uveden radi ilustracije. Blok od 5. do 9. linije programa 3.6 sadrži dve lokalne promenljive[7], deklarisane u liniji 6. Memorija zauzeta ovim promenljivama će biti oslobođena nakon 9. linije, dok će memorija zauzeta promenljivama deklarisanim u liniji 3 biti oslobođena nakon završetka programa u 11. liniji.

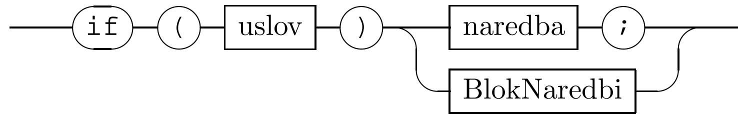
Pravu upotrebnu vrednost blokovi dobijaju uz naredbe za kontrolu toka izvršenja programa, kojima je moguće definisati, na primer, ponavljanje bloka naredbi. Naredbama za kontrolu izvršenja programa implementiraju se algoritamske strukture alternacije i petlje. Strukture alternacije u C-u su *if-then*, *if-then-else* i *switch* struktura, a pelje *while*, *repeat-until* i *for*.

[7]Pisanje lokalnih promenljivih u bloku je sintasno neispravno u ANSI C kompajleru, dok C99 i noviji kompajleri dozvoljavaju ovakve konstrukcije.

3.7.1 *if-then* i *if-then-else*

Ključna reč za naredbu grananja u C-u je *if*. Sintaksni dijagram grananja tipa *if-then* u C-u prikazan je na slici 3.8.

IfThen



Slika 3.8: Sintaksni dijagram grananja tipa *if-then*

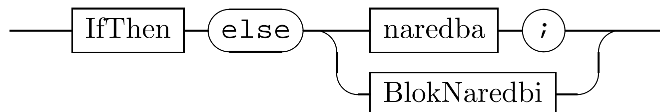
Sintaksni dijagram sa slike 3.8, opisan rečima, kaže da se iza ključne reči *if* u okviru malih zagrada '(' i ')' piše uslov grananja, nakon koga sledi naredba završena simbolom ';', ili blok naredbi. Uslov grananja u C-u može biti celobrojna promenljiva, ili celobrojni ili logički izraz. U slučaju celobrojne promenljive ili celobrojnog izraza uslov nije ispunjen ako je vrednost promenljive, odnosno izraza 0, a jeste ispunjen za bilo koju drugu vrednost.

Naredba ili blok naredbi čine telo grananja, odnosno "da" granu (definicija 2.7, strana 40). Telo grananja će se izvršiti ukoliko je uslov grananja ispunjen, a ukoliko nije telo će se preskočiti i izvršenje će nastaviti od prve naredbe koja se nalazi iza grananja.

Napomena: U telu grananja u C-u može se naći samo jedna naredba. Ukoliko je potrebno u telo grananja smestiti više od jedne naredbe, ove naredbe se moraju grupisati u vidu bloka. Imajući u vidu sintaksu bloka naredbi, ovo praktično rečeno znači da, ukoliko je u telu samo jedna naredba, mogu, ali i ne moraju se pisati zagrade '{' i '}'. Ukoliko se telo sastoji od više naredbi, ove zagrade su obavezne. Ovo važi za sve upravljačke strukture C-a, a ne samo za grananja.

Uz dodatak ključne reči *else*, grananje tipa *if-then* postaje grananje tipa *if-then-else*. Sintaksni dijagram *if-then-else* strukture prikazan je na slici 3.9.

IfThenElse



Slika 3.9: Sintaksni dijagram grananja tipa *if-then-else*

Zaključak iz prethodne napomene za pisanje naredbe ili bloka naredbi sa ili bez zagrada '{' i '}' važi i za *else* granu.

Program 3.7 ilustruje tri pomenuta načina zadavanja uslova grananja: celobrojnom promenljivom, celobrojnim izrazom i logičkim izrazom.

Program 3.7

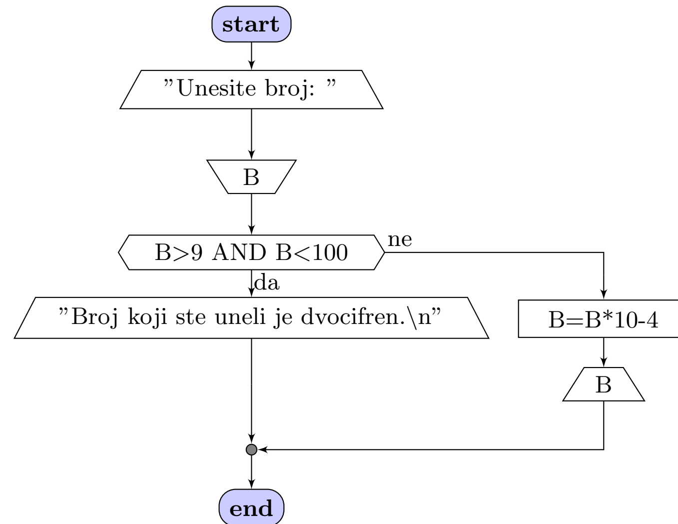
```
1  #include "stdio.h"
2  main()
3  {
4      int x = 2, y = 1;
5      if (x)
6         printf("Prvi␣uslov␣je␣ispunjen.\n");
7      if (x-2*y)
8          print("drugi:␣da\n");
9      else
10         print("drugi:␣ne\n");
11     if ( x > y )
12         printf("x␣je␣vece␣od␣y");
13     else
14         printf("y␣je␣vece␣od␣x");
15 }
```

Izlaz programa 3.7

Prvi uslov je ispunjen.
drugi: ne
x je vece od y
.

Primer 3.22 (Primer grananja)**. Zadatak**: Nacrtati strukturni dijagram toka algoritma i na programskom jeziku C napisati strukturni program koji određuje da li je uneta celobrojna vrednost dvocifreni broj. Ukoliko jeste, prikazati odgovarajuću poruku. Ukoliko nije, broj pomnožiti sa 10 i od broja oduzeti 4, a zatim prikazati tako dobijenu vrednost.

Rešenje:




```
1  #include "stdio.h"
2  main()
3  {
4      int B;
5      printf("Unesite broj: ");
6      scanf("%d",&B);
7      if (B > 9 && B < 100 )
8          printf("Broj koji ste uneli je dvocifren.\n");
9      else
10     {
11         B = B * 10 - 4;
12         printf("B=%d\n",B);
13     }
14 }
```

S obzirom na to da "da" grana sadrži samo jedan iskaz, nije neophodno otvarati i zatvarati blok (linije 7 i 8). Kako "ne" grana sadrži dve naredbe, neophodno je ove naredbe grupisati u blok (linije 10, 11, 12 i 13).

Izlazi koje program prikazuje kada se program izvrši sa ulaznim vrednostima 13 i 102 (za dva pokretanja programa) su:

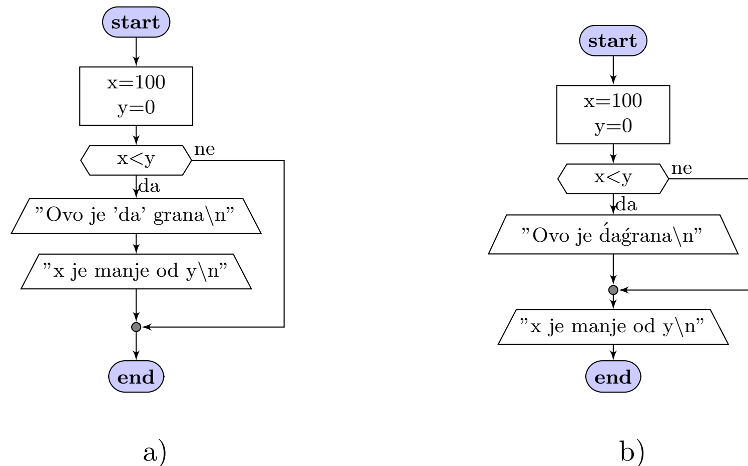
```
Unesite broj: 13
Broj koji ste uneli je dvocifren.

Unesite broj: 102
B=1016
```

△

Primer 3.23 (Izostavljanje bloka za slučaj više naredbi)**.** U ovom primeru ilustrovan je slučaj pogrešnog izostavljanja separatora za početak i kraj bloka, za slučaj kada "da" grana sadrži više od jedne naredbe.

Uzmimo primer algoritma sa slike 3.10a. Algoritam prikazan na slici 3.10a sadrži grananje, koje u "da" grani sadrži dve naredbe. Ukoliko se program napiše bez navođenja separatora za blok '{' i '}', smatra se da je samo jedna naredba u "da" grani, pa umesto algoritma sa slike 3.10a, dobijamo algoritam sa slike 3.10b. Može se reći da je ovako napisan program sintaksno ispravan, ali semantički neispravan, jer ne implementira željeni algoritam.



Slika 3.10: Primer upotrebe bloka u *if-then* strukturi: a) sa ispravnim blokom, b) sa neispravnim blokom

```
1  #include "stdio.h"
2  main()
3  {
4      int x = 100, y = 0;
5      if (x < y)
6          printf("Ovo je 'da' grana\n");
7          printf("x je manje od y\n");
8  }
```

Ovaj program na izlazu prikazuje:

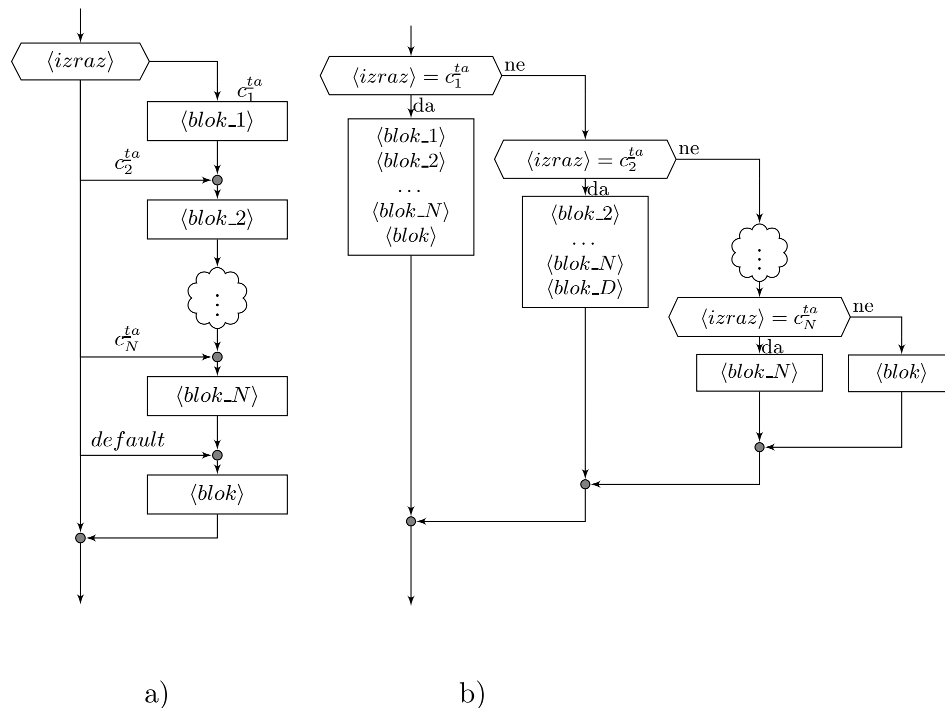
```
x je manje od y
```

Bez obzira što je početak naredbe u liniji 7 uvučen u odnosu na *if* iz 5. linije, kao što je ranije rečeno, programski jezik C nije pozicioni jezik, što znači da je prelazak

u novi red i uvučeno pisanje pojedinih linija samo "stil lepog pisanja", ali ne utiče na sintaksu jezika. △

3.7.2 *switch*

Switch struktura je poseban tip algoritamskih struktura, koji ne spada u osnovne algoritamske strukture, jer je ovu strukturu moguće implementirati ugnježdavanjem više grananja na odgovarajući način. Algoritamska reprezentacija *switch* strukture data je na slici 3.11a, a odgovarajući ekvivalentni algoritam, implementiran ugneždavanjem alternacija, dat je na slici 3.11.



Slika 3.11: Algoritamska reprezentacija: a) *switch* strukture, b) odgovarajuće kombinacije ugnježdenih grananja

Struktura *switch* radi tako što najpre odredi vrednost zadatog izraza, ili uzima vrednost zadate promenljive koja se po sintaksi može naći umesto izraza, i u zavisnosti od vrednosti skače na određeni deo koda u okviru same strukture. Svaki blok u okviru strukture se vezuje za određenu konstantnu vrednost, pa ako je vrednost izraza jednaka konstanti zadatoj za neki blok, skače se na taj blok. Izvršenje dalje teče počev od tog bloka pa do kraja strukture. Broj blokova koji je moguće zadati

je proizvoljan, i zavisi koje vrednosti izraza programer želi pokriti ovom strukturom. Ukoliko dobijena vrednost nije među navedenim konstantama, izvršiće se podrazumevani (*default*) blok (slika 3.11).

Podrazumevani (*default*) blok je moguće izostaviti. Ukoliko je ovaj blok izostavljen, a vrednost izraza nije među navedenim konstantama, neće se izvršiti nijedan blok ove strukture, a izvršenje programa će nastaviti od prve izvršne naredbe nakon *switch* strukture.

Sintaksa *switch* strukture je

$$
\begin{array}{ll}
Switch & ::= \\
& \textbf{switch}(\langle izraz\rangle|\langle promenljiva\rangle) \\
& \{ \\
& \quad \textbf{case}\ \langle konstanta_1\rangle : \quad \langle blok_naredbi_1\rangle; \\
& \quad \textbf{case}\ \langle konstanta_2\rangle : \quad \langle blok_naredbi_2\rangle; \\
& \quad \ldots \\
& \quad \textbf{case}\ \langle konstanta_N\rangle : \quad \langle blok_naredbi_N\rangle; \\
& \ [\ \textbf{default} : \langle blok_naredbi\rangle;\ ] \\
& \}
\end{array}
$$

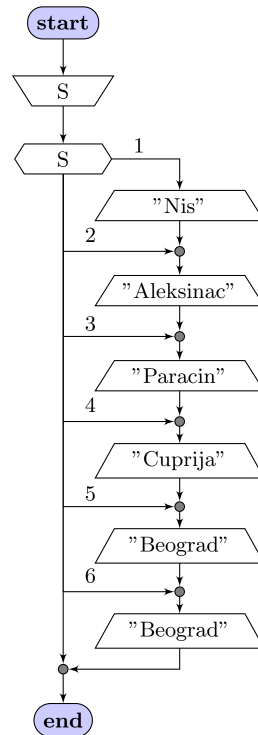
Primer 3.24 (Program koji prikazuje spisak autobuskih stanica)**. Zadatak**: Nacrtati strukturni dijagram toka algoritma i na programskom jeziku C napisati strukturni program koji za zadatu stanicu između Niša i Beograda ispisuje sve autobuske stanice, počev od zadate stanice do Beograda. Stanice su numerisane brojevima na sledeći način: Niš - 1, Aleksinac - 2, Paraćin - 3, Ćuprija - 4, Jagodina - 5, Beograd - 6.

Rešenje: Rešenje predstavlja direktnu implementaciju *switch* strukture. Rešenje je dato u nastavku.

```
1  #include "stdio.h"
2  main()
3  {
4      int S; // oznaka stanice
5      printf("unesite oznaku stanice: ");
6      scanf("%d",&S);
7      switch(S)
8      {
9          case 1 : printf("Nis\n");
10         case 2 : printf("Aleksinac\n");
11         case 3 : printf("Paracin\n");
12         case 4 : printf("Cuprija\n");
13         case 5 : printf("Jagodina\n");
14         case 6 : printf("Beograd\n");
15     }
16 }
```

Program prilikom izvršenja, za unetu vrednost 3, daje izlaz prikazan u nastavku.

```
unesite oznaku stanice: 3
Paracin
Cuprija
Jagodina
Beograd
```



Kako je u ovom rešenju izostavljen podrazumevani deo (*default*), ukoliko se unese vrednost različita od 1,2,3,4,5 ili 6 program neće prikazati ništa. △

Primer 3.25 (Program koji prikazuje spisak autobuskih stanica, II način)**. Zadatak**: Na programskom jeziku C napisati strukturni program koji za zadato početno slovo stanice između Niša i Beograda, ispisuje sve autobuske stanice počev od zadate stanice do Beograda. Prilikom ispisa naziva stanica, stanice numerisati brojevima tako da zadata stanica ima oznaku 1, druga 2, itd.

Rešenje: U odnosu na prethodni primer, u ovom primeru *switch* struktura vrši selekciju bloka počev od koga će krenuti izvršenje, na osnovu zadate karakter promenljive. Blokovi su označeni karakter konstantama, a uveden je i dodatni brojač koji se u svakom bloku inkrementira, kako bi pokazao redni broj stanice.

```
1  #include "stdio.h"
2  main()
3  {
4      char S; // oznaka stanice
5      int b = 1; // numeracija stanice
6      printf("unesite pocetno slovo stanice: ");
7      scanf("%c",&S);
8      switch(S)
9      {
10         case 'n' : printf("%d. Nis\n",b);
11             b++;
12         case 'a' : printf("%d. Aleksinac\n",b);
13             b++;
14         case 'p' : printf("%d. Paracin\n",b);
15             b++;
16         case 'c' : printf("%d. Cuprija\n",b);
17             b++;
18         case 'j' : printf("%d. Jagodina\n",b);
19             b++;
20         case 'b' : printf("%d. Beograd\n",b);
21     }
22 }
```

Program za unet karakter 'c' na izlazu prikazuje:

```
unesite pocetno slovo stanice: c
1. Cuprija
2. Jagodina
3. Beograd
```

U slučaju *switch* strukture, po sintaksi blokove naredbi je moguće odvajati separatorima '{' i '}', ali nije neophodno. Odvajanje nije neophodno, jer je početak i kraj svakog bloka nedvosmisleno određen *case* naredbom. △

3.7.3 *while*

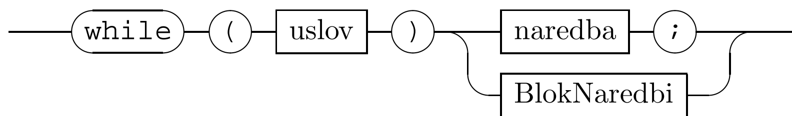
Sintaksa *while* petlje u C-u opisana u EBNF notaciji je [8]

$$
\begin{aligned}
While \;\; ::= & \\
& \textbf{while} \;\; '(' \langle uslov \rangle ')' \\
& \quad (\langle naredba \rangle \; ';' \,) \mid \langle blok \rangle
\end{aligned}
$$

Sintaksni dijagram *while* petlje prikazan je na slici 3.12.

[8]Sintaksa *while* petlje opisana rečima glasi: nakon ključne reči *while*, u malim zagradama koje su obavezne, navodi se uslov, pa naredba završena simbolom ';', ili blok naredbi.

While



Slika 3.12: Sintaksni dijagram *while* petlje

Naredba, odnosno blok naredbi predstavlja telo petlje. Telo petlje se izvršava sve dok je uslov ispunjen. Uslov *while* petlje može biti, kao i kod grananja, celobrojna promenljiva, celobrojni izraz, ili logički izraz. U slučaju celobrojne promenljive ili celobrojnog izraza uslov nije ispunjen ako je vrednost 0, a jeste ispunjen za bilo koju drugu vrednost.

Primer upotrebe *while* petlje dat je u okviru programa 3.8.

Program 3.8

```
1  #include "stdio.h"
2  main()
3  {
4      int x = 10;
5      while (x)
6      {
7          printf("%d,␣",x);
8          x = x - 1;
9      }
10     printf("\n");
11     while (x < 4)
12         printf("%d,␣",x++);
13 }
```

Izlaz programa 3.8

10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
0, 1, 2, 3,

.

Program iz primera 3.8 u okviru prve petlje ispisuje vrednosti promenljive x, uz dekrementiranje (smanjivanje za jedan) vrednosti promenljive svaki put kada se vrednost prikaže (linija 8). Petlja se izvršava sve dok vrednost promenljive x ne dostigne vrednost 0. Potrebno je naglasiti da se za ovu vrednost telo petlje neće izvršiti, jer je po uslovu x treba da bude strogo veće od 0. Nakon petlje izvršiće se prelazak u novi red (linija 10).

Druga petlja prikazuje i inkrementira (povećava za jedan) vrednost promenljive x, počev od zatečene vrednosti 0. Prikaz i inkrement su implementirani u istom iskazu (linija 12). Zarezi se prikazuju jer je tako definisano formatom u *printf* naredbama u linijama 7 i 12. Po zadatom formatu prikaza, iza svakog prikazanog

broja se prikazuje i zarez, uključujući zarez i iza poslednjeg prikazanog broja u redu.

Primer 3.26 (Konverzija broja iz dekadnog u binarni brojevni sistem)**. Zadatak**: Nacrtati strukturni dijagram toka algoritma i na programskom jeziku C napisati strukturni program koji vrši konverziju unetog celog dekadnog broja u binarni brojevni sistem. Cifre binarnog brojevnog sistema prikazati počev od cifre najmanje težine odvojene zarezima.

Rešenje: Konverziju nekog broja u binarni brojevni sistem moguće je obaviti tako što se broj podeli sa 2, a ostatak deljenja prikaže kao binarna cifra najmanje težine, nakon čega se rezultat deljenja ponovo deli sa 2, a ostatak uzima kao binarna cifra. Deljenje se obavlja sve dok se ne dobije vrednost 0. Pokazaćemo ovaj postupak na primeru konverzije dekadnog broja 23.

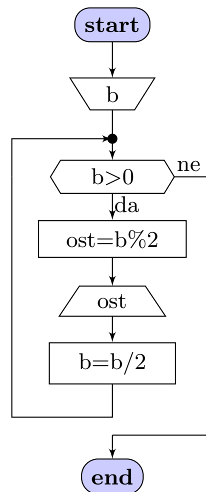
```
B=23
23/2 = 11, ostatak 1
11/2 =  5, ostatak 1
 5/2 =  2, ostatak 1
 2/2 =  1, ostatak 0
 1/2 =  0, ostatak 1
B=(10111)
```

Napomena: Ukoliko su operandi operatora deljenja u C-u celi brojevi i rezultat će biti ceo broj, koji se dobija odbacivanjem eventualnog razlomljenog dela rezultata. Rezultat deljenja brojeva 10/4 u C-u je 2.

Dijagram toka prethodno opisanog algoritma prikazan je na slici 3.13. Odgovarajući program napisan na programskom jeziku C dat je u nastavku.

```
1  #include "stdio.h"
2  main()
3  {
4      int b;   // broj koji se konvertuje
5      int ost; // pomocna prom. za ostatak
6      printf("unesite␣broj:␣");
7      scanf("%d",&b);
8      printf("B=");
9      while (b > 0)
10     {
11         ost = b % 2;
12         printf("%d,␣",ost);
13         b = b / 2;
14     }
15 }
```

Za uneti broj 23, program daje sledeći izlaz:



Slika 3.13: Dijagram toka algoritma za konverziju broja iz dekadnog u binarni brojevni sistem

```
unesite broj: 23
B=1, 1, 1, 0, 1,
```

Potrebno je napomenuti da su cifre binarnog broja prikazane u obrnutom redosledu, odnosno počev od cifre najmanje težine. △

3.7.4 *do-while*

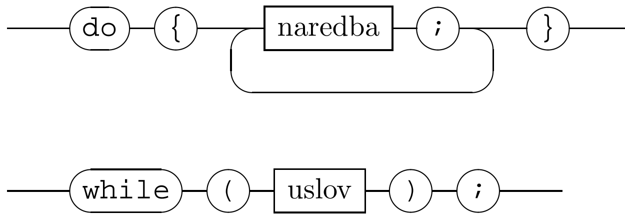
Sintaksa *repeat-until* petlje u C-u opisana u EBNF notaciji je [9]

$$
\begin{array}{lll}
\mathit{Repeat} & ::= & \\
& & \textbf{do} \\
& & '\{' \\
& & \quad \{\langle \mathit{naredba} \rangle\ ';'\ \} \\
& & '\}' \\
& & \textbf{while}\ \ '('\langle \mathit{uslov} \rangle')'\ ';'
\end{array}
$$

Sintaksni dijagram prikazan je na slici 3.14.

[9] Sintaksa *repeat-until* petlje opisana rečima glasi: nakon ključne reči *do* u okviru vitičastih zadrada koje su obavezne navodi se jedna ili više naredbi sa simbolom ';' na kraju, pa ključna reč *while*, nakon koje sledi uslov u malim zagradama i simbol ';'.

While



Slika 3.14: Sintaksni dijagram *do-while* petlje

Telo *do-while* petlje se izvršava sve dok uslov nije ispunjen. Uslov petlje takođe može biti celobrojna promenljiva, celobrojni izraz, ili logički izraz.

Primer 3.27 (Dijagram toka Euklidovog algoritma korišćenjem *do-while* petlje)**.**
Zadatak: Napisati program na C-u za Euklidov algoritam dat u primeru 2.15.

Rešenje: Program napisan na osnovu algoritma čiji je dijagram toka ilustrovan na slici 2.15, dat je u nastavku.

```
/* printf naredbe su dodate u odnosu na
originalni algoritam kako bi se video
svaki korak u izvrsenju ovog programa */
#include "stdio.h"
main()
{
    int M,N,R;
    printf("unesite dva broja: ");
    scanf("%d%d",&M,&N);
    printf("M=%d\n",M);
    printf("N=%d\n",N);
    do
    {
        R = M % N;
        printf("R=%d\n",R);
        M = N;
        printf("M=%d\n",M);
        N = R;
        printf("N=%d\n",N);
    }
    while(R != 0);
    printf("NZD je %d.\n",M);
}
```

Program za unete brojeve 15 i 10 daje sledeći izlaz:

```
unesite dva broja: 15 10
```

```
M=15
N=10
R=5
M=10
N=5
R=0
M=5
N=0
NZD je 5.
```

△

3.7.5 *for*

Sintaksa *for* petlje u C-u u EBNF notaciji je [10]

$$
\begin{aligned}
For \quad ::= & \\
& \textbf{for} \ '(' \ \langle izraz1 \rangle \ ';' \ \langle izraz2 \rangle \ ';' \ \langle izraz3 \rangle \ ')' \\
& \ (\langle naredba \rangle ';' \ ) \ | \langle blok \rangle
\end{aligned}
$$

Nakon ključne reči *for*, u okviru malih zagrada nalaze se tri izraza odvojena simbolima ';'. Prvi izraz, $\langle izraz1 \rangle$, služi za **inicijalizaciju brojača** petlje. Drugi izraz, $\langle izraz2 \rangle$, sadrži **uslov za kraj** petlje. Treći izraz, $\langle izraz3 \rangle$, opisuje **promenu brojača** nakon svake iteracije petlje. Po ovom opisu, *for* petlja u C-u predstavlja *while* petlju sa brojačem prikazanu na slici 2.35 (strana 63):

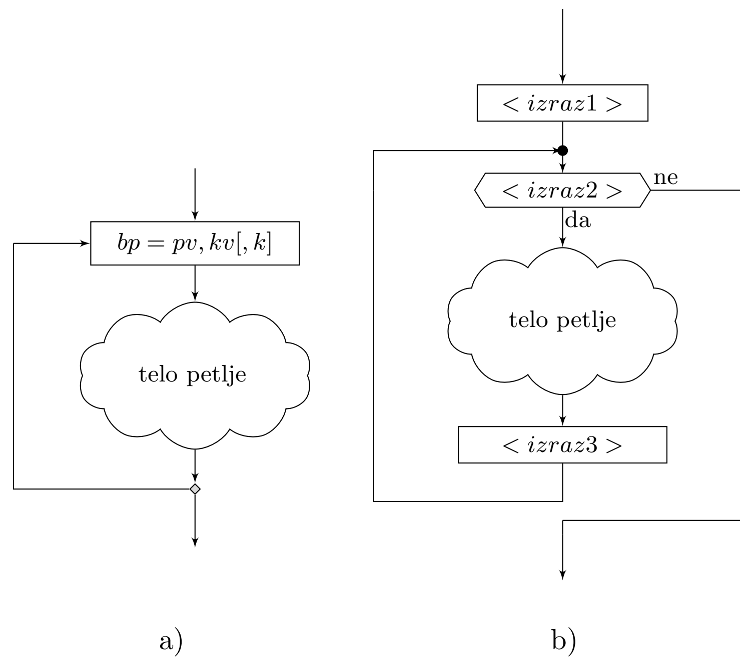
1. $\langle izraz1 \rangle$ se izvršava pre petlje i inicijalizuje brojač;
2. telo petlje se izvršava dok je uslov definisan izrazom $\langle izraz2 \rangle$ ispunjen; za slučaj da je kao $\langle izraz2 \rangle$ navedena celobrojna promenljiva ili celobrojni izraz, uslov je ispunjen ako $\langle izraz2 \rangle$ ima vrednost različitu od nule;
3. nakon završetka jedne iteracije tela petlje izvršava se $\langle izraz3 \rangle$, koji menja vrednost brojača petlje, nakon čega se izvršenje vraća na početak petlje gde se ponovo ispituje uslov.

Oznaka za *for* petlju u dijagramu toka algoritma prikazana je na slici 3.15a, a princip rada *for* petlje u C-u prikazan je na slici 3.15b.

Četiri primera upotrebe *for* petlje data su u okviru listinga programa 3.9. Kako petlje nisu ugneždene, za sve petlje je korišćen isti brojač i. Prve tri petlje "broje unapred", dok poslednja petlja "broji unazad", t.j. petlja ima negativan korak.

Petlja u liniji 6 programa 3.9 inicijalizuje brojač na vrednost $i = 1$, a izvršava se sve dok je $i < 10$. Izraz za promenu vrednosti brojača je $i = i+1$, čime se dobija petlja po i od 1 do 9 sa korakom $k = 1$.

[10] Sintaksa *for* petlje opisana rečima glasi: nakon ključne reči *for* u okviru malih zadrada, koje su obavezne, navode se tri izraza razdvojena simbolima ';', a zatim ili naredba nakon koje sledi ';', ili blok naredbi.



Slika 3.15: Dijagram toka algoritma koji odgovara *for* petlji u C-u: a) uobičajena reprezentacija, b) pozicija izraza u toku izvršenja.

U liniji 10 je takođe definisana petlja sa korakom $k = 1$, s tim da je u ovoj liniji $\langle izraz3 \rangle$ dat u skraćenom obliku pomoću operatora za inkrementiranje vrednosti promenljive i++.

Program 3.9

```
1  #include "stdio.h"
2  main()
3  {
4      int i;
5      // petlja unapred od 1 do 9: i=1 do 9
6      for (i = 1; i < 10; i = i + 1)
7          printf("%d, ",i);
8      printf("\n");
9      // petlja unapred od -5 do 5: i=-5 do 5
10     for (i = -5; i <= 5; i++)
11         printf("%d, ",i);
12     printf("\n");
13     // petlja unapred od 0 do 19 sa korakom 3: i=0 do 19, korak 3
14     for (i = 0; i < 20; i = i + 3)
```

```
15          printf("%d, ",i);
16      printf("\n");
17      // petlja unazad po i od 10 do 2, sa korakom -1
18      for (i = 10; i > 1; i--)
19          printf("%d, ",i);
20  }
```

Izlaz programa 3.9

1, 2, 3, 4, 5, 6, 7, 8, 9,
-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5,
0, 3, 6, 9, 12, 15, 18,
10, 9, 8, 7, 6, 5, 4, 3, 2,

.

Linija 14 sadrži petlju kod koje $\langle izraz3 \rangle$ povećava brojač za 3, čime se dobija petlja sa korakom $k = 3$. Inicijalizacija, uslov petlje i promena vrednosti brojača petlje u liniji 18 ukazuju na petlju sa granicama $pv = 10, kv = 2$ i korakom $k = -1$.

Iskazi u linijama 8, 12 i 16 su dodati kako bi se razdvojio prikaz iz različitih petlji u posebne redove.

Ukoliko se umesto ANSI-C koristi C99 ili noviji kompajler, kako je ranije pomenuto, promenljive je moguće deklarisati na bilo kom mestu u programu, a ne samo na početku pre prve izvršne naredbe. Kod C99 i novijih kompajlera brojač petlje je moguće istovremeno deklarisati i inicijalizovati u okviru $\langle izraz1 \rangle$. Ovo je pokazano u primeru programa 3.10. U ovom slučaju promenljiva brojača petlje postoji samo dok se petlja izvršava, a memorija zauzeta brojačem se oslobađa nakon završetka petlje.

Program 3.10

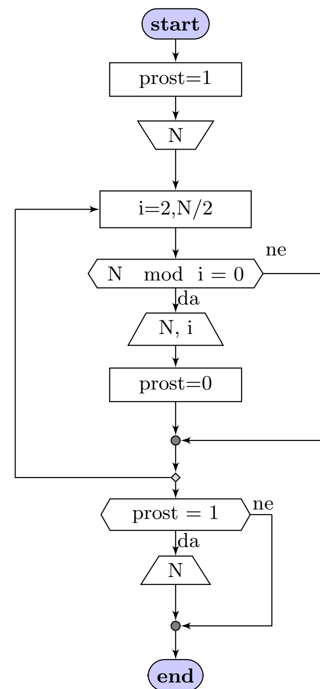
```
1  #include "stdio.h"
2  main()
3  {
4      for (int i = 1; i < 10; i++)
5          printf("%d, ",i);
6  }
```

Primer 3.28 (Prosti brojevi). **Zadatak**: Nacrtati strukturni dijagram toka algoritma i na programskom jeziku C napisati strukturni algoritam koji ispituje da li je zadati broj prost. Broj je prost ukoliko je deljiv jedino jedinicom i samim sobom. Ukoliko broj nije prost, prikazati sve brojeve kojim je zadati broj deljiv.

Rešenje: Kako je svaki celi broj deljiv jedinicom i samim sobom bez ostatka, korisna informacija o tome da li je broj prost ili ne može se dobiti ispitivanjem da li je zadati broj deljiv bez ostatka i nekim drugim brojem, osim jedinicom i samim sobom. Jedno moguće rešenje je izvršiti proveru da li je broj deljiv brojem 2, pa zatim brojem 3, 4, itd, sve do polovine vrednosti zadatog broja. Preko polovine

nema potrebe ispitivati, jer broj N svakako ne može biti deljiv brojevima većim od $N/2$. Ukoliko je zadati broj N deljiv bar jednim brojem od 2 do $N/2$, sledi da nije prost.

Dijagram toka algoritma je prikazan na slici 3.16.



Slika 3.16: Dijagram toka algoritma za ispitivanje da li je uneti broj prost

Na početku ovog algoritma pretpostavljeno je da je broj prost (promenljiva $prost = 1$). Ukoliko se utvrdi da je deljiv bar jednim brojem od 2 do $N/2$ pretpostavka će se promeniti (promenljiva $prost = 0$), nakon čega bez obzira na deljivost ostalim brojevima, ne postoji mogućnost postavljanja promenljive *prost* na 1 u okviru petlje. To znači da ukoliko se vrednost promenljive *prost* zbog deljivosti broja N jednim brojem promeni na 0, više se neće vratiti na 1.

Program na C-u za algoritam sa slike 3.16 je dat u nastavku.

```
1  #include "stdio.h"
2  main()
3  {
4      int N, prost = 1,i;
5      printf("Unesite␣broj␣koji␣zelite␣ispitati:␣");
6      scanf("%d",&N);
7      for (i = 2; i < N/2; i++)
```

```
 8      {
 9          if ( N % i == 0)
10          {
11              printf("Broj %d je deljiv brojem %d.\n",N,i);
12              prost = 0;
13          }
14      }
15      if ( prost == 1)
16      printf("Broj %d je prost.\n",N);
17  }
```

Rezultati izvršenja sa različitim ulaznim vrednostima dati su u nastavku.

```
C:\primer.exe
Unesite broj koji zelite ispitati: 17
Broj 17 je prost.

C:\primer.exe
Unesite broj koji zelite ispitati: 23
Broj 23 je prost.

C:\primer.exe
Unesite broj koji zelite ispitati: 18
Broj 18 je deljiv brojem 2.
Broj 18 je deljiv brojem 3.
Broj 18 je deljiv brojem 6.
```

Potrebno je napomenuti da je optimalnije koristiti *while* petlju za ispitivanje da li je broj prost ili ne, i prekinuti izvršenje petlje čim se utvrdi da je broj deljiv nekim brojem. Ukoliko se utvrdi da je broj deljiv samo jednom brojem od 2 do $N/2$, nema razloga ispitivati dalje. U ovom primeru korišćenje *for* petlje bilo je neophodno zbog dela teksta zadatka koji glasi: *"Ukoliko broj nije prost prikazati sve brojeve kojim je zadati broj deljiv"*. △

3.8 Naredbe bezuslovnog skoka

Naredbe bezuslovnog skoka su naredbe čijim se izvršenjem bezuslovno "skače" na neki drugi deo programa i izvršenje se nadalje nastavlja od naredbe na koju se prešlo bezuslovnim skokom. Bezuslovnim menjanjem toka izvršenja programa može se bezuslovno preći na naredbu koja se može naći na proizvoljnoj udaljenosti pre ili posle same naredbe bezuslovnog skoka. Ovim se može uneti nestrukturnost u algoritam, pa je, ukoliko je cilj projektovati strukturni algoritam, ove naredbe potrebno pažljivo koristiti.

U C-u postoje tri naredbe bezuslovnog skoka:

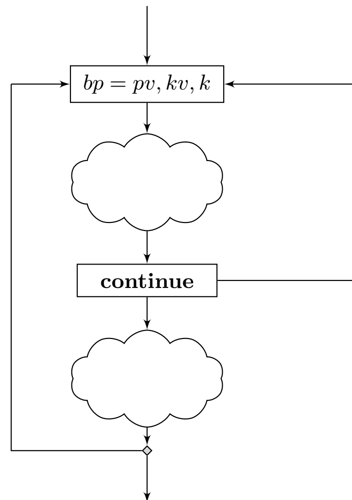
1. continue,

2. break, i
3. goto.

Naredbe *continue*, *break* i *goto* su ključne reči programskog jezika C.

3.8.1 Naredba *continue*

Kada program u toku izvršenja naiđe na naredbu *continue* (prev. *nastavi*), ukoliko se ova naredba nalazi u petlji, izvršenje trenutne iteracije petlje će se prekinuti i nastaviće od naredne iteracije petlje. Ovo je ilustrovano dijagramom toka algoritma na slici 3.17.



Slika 3.17: Dijagram toka petlje koja u sebi sadrži naredbu *continue*

Na primer, petlja u liniji 4 programa 3.11 sadrži *continue* naredbu koja se izvršava ako je ispunjen uslov da je brojač petlje $i = 2$. Posledica ovoga je da se u drugoj iteraciji petlje ne izvršava deo tela petlje koji se nalazi iza naredbe *continue*, što se vidi iz priloženog izlaza.

Program 3.11

```
1 #include "stdio.h"
2 main()
3 {
4     for (int i = 1; i <= 3; i++)
5     {
6         printf("Pocetak␣%d.␣iteracije\n",i);
7         if ( i == 2)
```

```
8               continue;
9           printf("Kraj %d. iteracije\n\n",i);
10      }
11      printf("Kraj programa\n");
12  }
```

Izlaz programa 3.11

Pocetak 1. iteracije
Kraj 1. iteracije

Pocetak 2. iteracije
Pocetak 3. iteracije
Kraj 3. iteracije
Kraj programa

3.8.2 Naredba *break*

Kada program u toku izvršenja naiđe na naredbu *break* (prev. *prekid*), ukoliko se ova naredba nalazi u petlji, izvršenje petlje se prekida i nastavlja se sa prvom naredbom neposredno iza petlje. Ovo je ilustrovano dijagramom toka algoritma na slici 3.18.

Na primer, petlja u liniji 4 programa 3.12 sadrži *break* naredbu koja se izvršava ako je ispunjen uslov da je brojač petlje $i = 2$. Posledica ovoga je da se izvršenje petlje završava na polovini druge iteracije petlje, u trenutku izvršenja naredbe *break*, što se vidi iz priloženog izlaza.

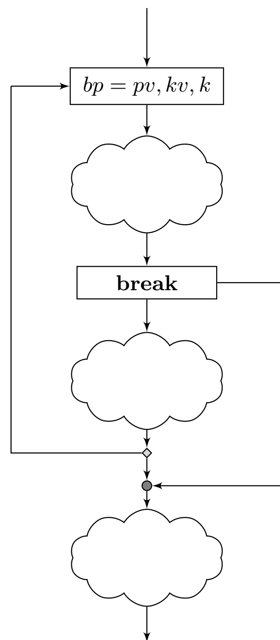
Program 3.12

```
1   #include "stdio.h"
2   main()
3   {
4       for (int i = 1; i <= 3; i++)
5       {
6           printf("Pocetak %d. iteracije\n",i);
7           if ( i == 2)
8               break;
9           printf("Kraj %d. iteracije\n\n",i);
10      }
11      printf("Kraj programa\n");
12  }
```

Izlaz programa 3.12

Pocetak 1. iteracije
Kraj 1. iteracije

Pocetak 2. iteracije



Slika 3.18: Dijagram toka petlje koja u sebi sadrži naredbu *break*

Kraj programa

3.8.3 Naredba *goto*

Pozivanjem naredbe *goto* moguće je "skočiti" na bilo koju liniju programa. Ova naredba je naziv dobila od engleskih reči *Go To*, što u prevodu znači "*idi*", ili "*pređi na...*". Linija programa na koju se "skače" na početku se označava labelom. Labela je proizvoljno zadati identifikator, koji se završava znakom ':'. Sintaksna definicija labele i naredbe *goto* kojom se "skače" na definisanu labelu je:

$$\langle labela \rangle \ ::= \ \langle identifikator \rangle ' : ' \ [\ \langle naredba \rangle\ ]$$

$$\langle bezuslovni_skok \rangle \ ::= \ \textbf{goto}\ \langle labela \rangle\ ';'$$

Program može sadržati proizvoljan broj labela i naredbi *goto*. Potrebno je napomenuti da nije sintaksno neispravno u programu imati labelu na koju se ne skače nijednom *goto* naredbom.

Primer 3.29 (Ilustracija *goto naredbe*). Program dat u ovom primeru ilustruje sintaksu i upotrebu *goto* naredbe. Program zahteva od korisnika da unese dvocifreni broj. Ukoliko uneti broj nije dvocifren, izvršenje se naredbom *goto* iz 11. linije koda vraća na labelu *opet* u 5. liniji koda, kako bi se od korisnika ponovo zahtevao unos.

```
1  #include "stdio.h"
2  main()
3  {
4      int N;
5  opet:          // labela na koju se moze "skociti" goto naredbom
6      printf("Unesite dvocifren broj: ");
7      scanf("%d",&N);
8      if ( N < 10 || N > 100 )
9      {
10         printf("Broj koji ste uneli nije dvocifren\n");
11         goto opet;     // bezuslovni skok na labelu
12     }
13     printf("Hvala sto ste uneli dvocifreni broj.\n");
14 }
```

Primer izlaza koji prikazuje program dat je u nastavku.

```
Unesite dvocifren broj: 1
Broj koji ste uneli nije dvocifren
Unesite dvocifren broj: 123
Broj koji ste uneli nije dvocifren
Unesite dvocifren broj: 999
Broj koji ste uneli nije dvocifren
Unesite dvocifren broj: 50
Hvala sto ste uneli dvocifreni broj.
```

Dijagram toka algoritma prikazan je na slici 3.19.

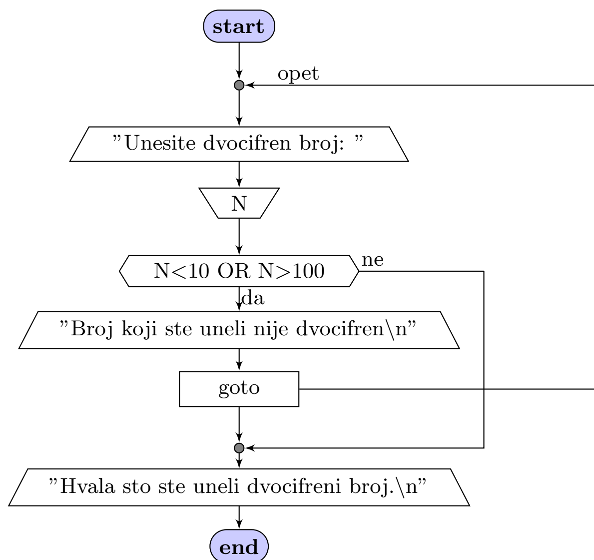
Napomena: Prethodni program i dijagram toka sa slike 3.19 su nestrukturni. Naredbom *goto* kreiran je alternativni izlaz iz strukture grananja, tako da ova struktura ima jedan ulaz i dva potencijalna izlaza (regularni, i preko *goto*), što je u suprotnosti sa definicijom strukturnih algoritama. Problem iz ovog primera moguće je relativno jednostavno rešiti na strukturni način upotrebom petlje *do-while*.

△

Bez obzira što naredbe bezuslovnog skoka, kako im i ime kaže, prelaze na zadatu naredbu bez ispitivanja uslova, u praktičnoj primeni ove naredbe se obično nalaze u okviru alternacije (kao u primeru 3.29), gde dobijaju pravu upotrebnu vrednost.

3.8.4 Primena naredbi bezuslovnog skoka

Naredbama bezuslovnog skoka moguće je implementirati bilo koju algoritamsku strukturu, bilo strukturnu ili nestrukturnu. Skokovi na proizvoljne lokacije u pro-



Slika 3.19: Dijagram toka algoritma za programa sa naredbom bezuslovnog skoka

gramu omogućavaju veliku fleksibilnost programeru, ali i unose rizik od pojave semantičkih grešaka koje se teško otklanjaju, jer se program teško prati.

Upotreba naredbi skoka je posebno izražena u asemblerskim jezicima, gde su naredbe bezuslovnog i naredbe uslovnog skoka često jedine dostupne naredbe za kontrolu toka izvršenja programa. Kombinovanjem naredbi bezuslovnog skoka i strukture alternacije[11] moguće je implementirati bilo koju petlju.

Naredbe bezuslovnog skoka, konkretno u C-u, nalaze primenu i kod modifikacije toka *switch* strukture.

Formiranje strukturne petlje naredbom bezuslovnog skoka

Kombinovanjem grananja i naredbi bezuslovnog skoka moguće je implementirati petlje tipa *while*, *do-while* i *for*. S obzirom na to da je jednostavnije implementirati petlju tipa *do-while*, razmotrićemo prvo ovaj slučaj.

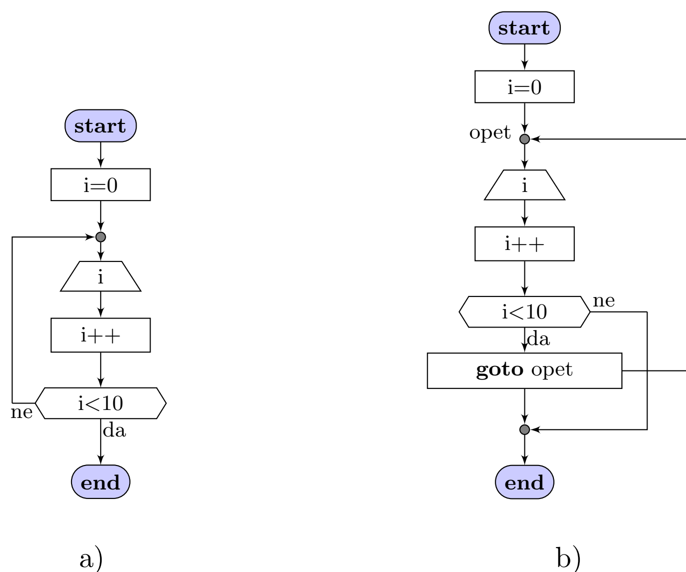
Dijagram toka petlje tipa *do-while* prikazan je na slici 2.24 (strana 50). Ova petlja spada u bezuslovne petlje kod koje se telo petlje ponavlja sve dok se uslov petlje ne ispuni. Uslov petlje nalazi se na kraju petlje, nakon poslednje naredbe tela petlje.

[11]Za alternaciju se u asamblerskim jezicima koriste naredbe uslovnog skoka.

Petlju tipa *do-while* moguće je implementirati naredbama bezuslovnog skoka tako što se sa kraja tela petlje naredbom bezuslovnog skoka prelazi na početak petlje. Da bi se obezbedilo da se petlja završi u zavisnosti od uslova kao kod *do-while*, naredba bezuslovnog skoka stavlja se u "da" granu alternacije, čiji uslov praktično postaje uslov petlje.

Primer 3.30 (Implementacija petlje tipa *do-while* naredbama bezuslovnog skoka)**.** Implementaciju *do-while* petlje naredbama bezuslovnog skoka pokazaćemo na primeru dijagrama toka algoritma sa slike 3.20a. Na slici 3.20a dat je jednostavan algoritam koji prikazuje vrednost promenljive i sve dok ova vrednost ne dostigne vrednost 10. Početna vrednost je $i = 0$.

Odgovarajuća implementacija naredbama bezuslovnog skoka prikazana je na slici 3.20b. Algoritam sa slike 3.20, nakon prolaska kroz inicijalizaciju promenljive i, prolazi kroz labelu za povratak, što nema uticaja na izvršenje, pa se izvršenje nastavlja prikazom i inkrementom promenljive, koje predstavljaju telo petlje.



Slika 3.20: Dijagram toka algoritma petlje tipa *do-while*: a) regularne petlje, b) petlje implementirane naredbama bezuslovnog skoka

Kako se kod *do-while* petlje uslov ispituje nakon tela petlje, i algoritam sa slike 3.20b na ovom mestu ispituje uslov, ali alternacijom. Ukoliko je uslov alternacije ispunjen, izvršiće se skok na labelu *opet*. Ako uslov nije ispunjen preskočiće se naredba skoka i izvršenje će nastaviti od prve naredne naredbe nakon alternacije, odnosno nakon petlje.

Potrebno je naglasiti da je sama alternacija sa slike 3.20b na nestrukturan način uključena u algoritam, jer ima jednu ulaznu i dve izlazne grane: regularnu i preko *goto* naredbe.

Program za algoritam prikazan dijagramom toka na slici 3.20b dat je u nastavku.

```
1  #include "stdio.h"
2  main()
3  {
4      int i = 0;
5  opet:
6      printf("%d, ",i);
7      i++;
8      if (i<10)
9          goto opet;
10 }
```

Program daje sledeći izlaz:

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

△

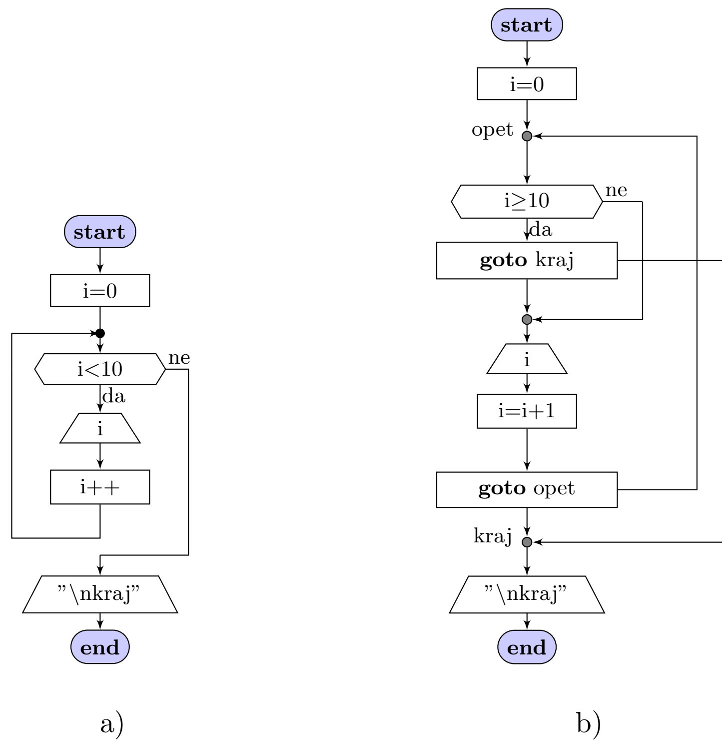
Naredbama bezuslovnog skoka moguće je na sličan način implementirati i *while* petlju. Kod *while* petlje uslov je na početku petlje, a sa kraja petlje se izvršenje bezuslovno vraća na početak petlje, neposredno pre trenutka provere uslova petlje, kako je to ilustrovano na slici 2.17 (strana 43).

Primer 3.31 (Implementacija petlje tipa *while* naredbama bezuslovnog skoka)**.** Implementaciju *while* petlje naredbama bezuslovnog skoka pokazaćemo na primeru algoritma čiji je dijagram toka prikazan na slici 3.21a.

Kao i u prethodnom primeru, program prikazuje redom brojeve od 0 do 9, tako što na početku inicijalizuje brojač na 0, a zatim ga povećava za 1 uz ponovno prikazivanje. Ovaj proces se obavlja sve dok je vrednost promenljive $i < 10$.

Odgovarajući algoritam koji naredbama bezuslovnog skoka implementira algoritam, prikazan na slici 3.21a, dat je na slici 3.21b. Algoritam sa slike 3.21b nakon inicijalizacije promenljive i ispituje uslov, jer je kod petlje tipa *while* uslov na početku. Uslov je negiran u odnosu na uslov sa slike 3.21a, pa ako uslov kao takav nije ispunjen (originalni uslov je ispunjen), preskočiće se naredba bezuslovnog skoka i počeće izvršenje petlje. Da uslov nije negiran naredba bezuslovnog skoka bi se trebalo naći u "ne" grani, a ne u "da" grani alternacije sa slike 3.21b, što je takođe izvodljivo.

Na kraju petlje je bezuslovni skok na početak petlje neposredno pre ispitivanja uslova (labela *opet*). Ukoliko je uslov ispunjen skočiće se na labelu koja se nalazi neposredno nakon tela petlje, kako bi se nastavilo sa izvršavanjem naredbi nakon petlje.



Slika 3.21: Dijagram toka algoritma petlje tipa *while*: a) regularne petlje, b) petlje implementirane naredbama bezuslovnog skoka

Potrebno je naglasiti da je alternacija sa slike 3.21 nestrukturna, jer ima jednu ulaznu i dve izlazne grane: regularnu i preko *goto* naredbe. Program za algoritam čiji je dijagram toka dat na slici 3.21b dat je u nastavku.

```
1  #include "stdio.h"
2  main()
3  {
4      int i = 0;
5  opet:
6      if (i>=10)
7          goto kraj;
8      printf("%d, ",i);
9      i++;
10     goto opet;
11 kraj:
12     printf("\nkraj");
13 }
```

Program na izlazu prikazuje sledeće vrednosti:

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
kraj
```

△

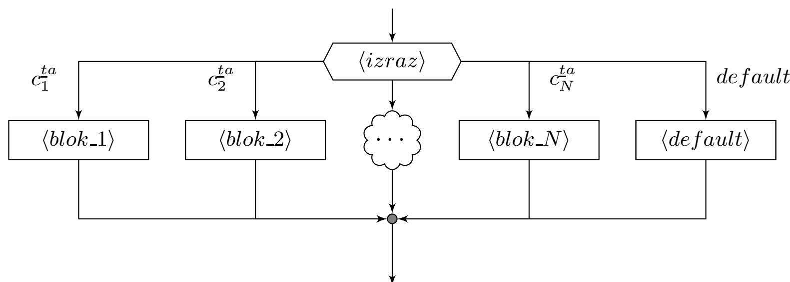
Petlje tipa *for* moguće je implementirati naredbama bezuslovnog skoka na isti način kako je to urađeno na slici 3.21b za petlje tipa *while*. Promenljiva i bi u ovom slučaju bila brojač petlje. Naredba $i = 0$ sa slike 3.21b definisala bi inicijalnu vrednost brojača, odnosno početak petlje. Uslov $i \geq 10$ je negacija uslova petlje, a naredba $i = i + 1$ na kraju tela petlje je promena vrednosti brojača, odnosno korak petlje, u ovom slučaju $k = 1$. Ovo je i opisano kao transformacija petlje tipa *for* u petlju tipa *while* i prikazano na slici 2.35 na strani 63.

Kako mašinski jezici uglavnom za kontrolu toka imaju samo naredbe bezuslovnog skoka i naredbe alternacije, kompajleri viših programskih jezika programe na ovim jezicima prevode u izvršni program na sličan način kako je to pokazano na slikama 3.20 i 3.21. Neki asemblerski jezici imaju i naredbe koje kombinuju alternaciju i bezuslovni skok. Ovakve naredbe se nazivaju **naredbama uslovnog skoka**.

Pisanje programa direktno na asemblerskim jezicima, bez upotrebe viših programskih jezika svodi se na pisanje programa po algoritmima se slika 3.20 i 3.21.

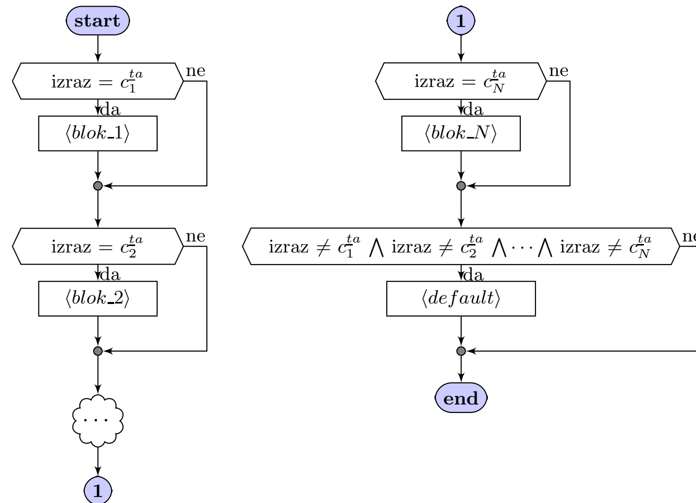
Modifikovana *switch* struktura

Kombinovanjem regularne *switch* strukture prikazane na slici 3.11 (strana 120) i naredbi bezuslovnog skoka može se dobiti algoritamska struktura koja je mnogo češća u programiranju od same *switch* strukture sa slike 3.11. Ova struktura prikazana je na slici 3.22, a odgovarajući dijagram toka ekvivalentnog algoritma implementiranog alternacijama dat je na slici 3.23.



Slika 3.22: Modifikovana *switch* struktura

Kako je na slici 3.23 prikazano, struktura sa slike 3.22 proverava vrednost izraza (ili promenljive) i ako je vrednost jednaka konstanti c_1^{ta} izvršiće se blok ⟨*blok_1*⟩. Ako je vrednost izraza jednaka konstanti c_2^{ta}, izvršiće se blok ⟨*blok_2*⟩, itd. Ukoliko vrednost nije među navedenim konstantama, a postoji podrazumevani blok ⟨*default*⟩, tada će se izvršiti podrazumevani blok. Nakon izvršenja određenog bloka, izvršenje se nastavlja sa prvom narednom naredbom iza ove strukture. Ukoliko podrazumevani blok ⟨*default*⟩ ne postoji, i vrednost izraza nije među navedenim konstantama, neće se izvršiti nijedan blok i izvršenje će preći na prvu narednu naredbu iza ove strukture.



Slika 3.23: Ekvivalentni algoritam modifikovanoj *switch* strukturi implementiran alternacijama

Napomena: U programskom jeziku Pascal *switch* struktura se naziva *case* i u osnovi ima izgled strukture prikazane na slici 3.22, a ne originalne *switch* strukture sa slike 3.11.

Struktura sa slike 3.22 se u programskom jeziku C dobija tako što se naredba bezuslovnog skoka *break* doda na kraj svakog bloka. Sintaksa modifikovane *switch*

strukture je

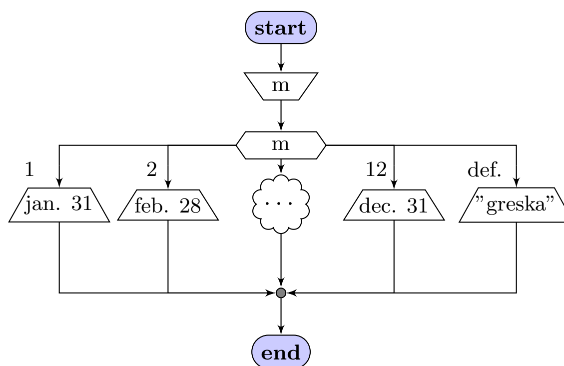
$$
\begin{array}{ll}
Switch & ::= \\
& \textbf{switch}(\langle izraz\rangle|\langle promenljiva\rangle) \\
& \{ \\
& \quad \textbf{case}\ \langle konstanta_1\rangle : \quad \langle blok_naredbi_1\rangle; \\
& \qquad\quad \textbf{break}; \\
& \quad \textbf{case}\ \langle konstanta_2\rangle : \quad \langle blok_naredbi_2\rangle; \\
& \qquad\quad \textbf{break}; \\
& \quad \ldots \\
& \quad \textbf{case}\ \langle konstanta_N\rangle : \quad \langle blok_naredbi_N\rangle; \\
& \qquad\quad \textbf{break}; \\
& \ [\ \textbf{default} : \langle blok_naredbi\rangle;\ ] \\
& \}
\end{array}
$$

Naredba *break* prekida izvršenje cele strukture nakon izvršenja bloka na koji je izvršenje prešlo na osnovu vrednosti izraza. Time se dobija struktura sa slike 3.22. Potrebno je napomenuti da naredba *break* nije neophodna nakon podrazumevanog $\langle default\rangle$ bloka, mada nije greška ni da se navede, jer će se izvršenje strukture svakako završiti nakon ovog bloka.

Napomena: Nije obavezno dodavanje naredbe *break* na kraj svakog bloka. Ako se *break* izostavi iz nekog bloka dobiće se kombinacija algoritamskih struktura sa slika 3.11 i 3.22.

Primer 3.32 (Primer modifikovane *switch* strukture)**. Zadatak**: Nacrtati strukturni dijagram toka algoritma i na programskom jeziku C napisati strukturni program koji za zadati redni broj meseca u godini prikazuje ime meseca i broj dana u mesecu.

Rešenje: Algoritam rešenja datog problema predstavlja direktnu primenu modifikovane *switch* strukture na dati problem. Dijagram toka algoritma i program na programskom jeziku C dati su u nastavku.



```
1  #include "stdio.h"
2  main()
3  {
4      int m;
5      printf("Unesite redni broj meseca u godini: ");
6      scanf("%d",&m);
7      switch(m)
8      {
9          case 1: printf("januar, ima 31 dan.");
10             break;
11         case 2: printf("februar ima 28 dana.");
12             break;
13         // ...
14         case 12: printf("decembar, ima 31 dan.");
15         break;
16         default: printf("pogresan unos.");
17     }
18 }
```

△

3.9 Preprocesorske direktive

Preprocesorske direktive su direktive (smernice, naredbe) programskog jezika kojima je moguće uticati na kompajler i sam proces prevođenja programa. Kompajler na početku procesa prevođenja programa proverava da li program sadrži preprocesorske direktive i deluje na osnovu njih (slika 3.5, strana 92). Ova faza kompajliranja se naziva preprocesiranje. Komponenta kompajlera koja vrši preprocesiranje naziva se **preprocesor**.

Ulaz u fazu preprocesiranja je izvorni kod C programa. Izlaz iz ove faze je takođe izvorni program na C-u, ali sa razrešenim direktivama i izmenjenim kodom na osnovu navedenih preprocesorskih direktiva.

Postoje tri kategorije preprocesorskih direktiva:

1. direktive za definisanje simboličkih konstanti i makroa,
2. direktive za uključivanje biblioteka, i
3. direktive za uslovno prevođenje programa.

Zavisno od tipa preprocesorske direktive, preprocesorske direktive mogu se naći na početku programa, ili bilo gde u programu. Po pravilu, direktive za uključivanje biblioteka nalaze se na samom početku programa, dok se ostale dve kategorije direktiva mogu naći bilo gde u programu.

Svaka preprocesorska direktiva počinje simbolom '#' za kojim sledi ključna reč. Iza ključne reči direktive navode se parametri direktive. Svaka direktiva mora biti

napisana u jednoj programskoj liniji, bez obaveze navođenja tačke i zareza na kraju linije. Prethodno opisana sintaksa preprocesorske direktive u EBNF je

⟨*preproc._dorektiva*⟩ ::=
#⟨*kljucna_rec*⟩ ⟨*parametri*⟩

⟨*kljucna_rec*⟩ ::= **include** | **define** | **undef** | **if** | **ifdef** | **endif** | **ifndef** | **else** | **elif**

3.9.1 Simboličke konstante i makroi

Ukoliko se neki delovi koda često koriste u programu, te delove moguće je zameniti simboličkim konstantama koje su jednostavnije i kraće za pisanje.

Definicija 3.19 (Simbolička konstanta)**.** Simbolička konstanta je identifikator čije se pojavljivanje u programu u fazi preprocesiranja zamenjuje zadatim konstantnim nizom simbola.

Ključna reč za definisanje simboličkih konstanti u C-u je `#define`. Sintaksa definicije simboličke konstante je sledeća:

⟨*simbolicka_konstanta*⟩ ::=
#define ⟨*identifikator*⟩ ⟨*niz_simbola*⟩

U fazi preprocesiranja se proverava sintaksna ispravnost identifikatora po definiciji 3.4 (strana 94), ali ne i ispravnost niza simbola. Preprocesor svako pojavljivanje identifikatora u programu jednostavno zamenjuje navedenim nizom simbola, pa program sa tim izmenama prosleđuje u naredni stepen prevodioca.

Napomena: Sintaksna ispravnost niza simbola proverava se od strane prevodioca na mestu na kom je preprocesor umetnuo ovaj niz u program umesto pronađenog identifikatora konstante, a ne na mestu definicije simboličke konstante.

Definicija simboličke konstante može se naći bilo gde u programu, a konstanta se može koristiti od mesta definicije pa nadalje. Uobičajena praksa je da se simboličke konstante navedu na početku programa.

Primer 3.33 (Upotreba simboličkih konstanti)**.** Na početku sledećeg programa definisane su dve simboličke konstante: `Pi` i `e`. Ove konstante koriste se na nekoliko mesta u programu.

```
1 #define Pi    3.14159265
2 #define e     2.71828183
3 #include "stdio.h"
4 main()
5 {
```

```
6       float R1=1, R2=e, O1, O2; // poluprecnik i obim kruga
7       O1 = 2*R1*Pi;
8       O2 = 2*R2*Pi;
9   }
```

Ekvivalentni program bez simboličkih konstanti, koji preprocesor prosleđuje u narednu fazu kompajliranja dat je u nastavku.

```
1   main()
2   {
3       float R1=1, R2=2.71828183, O1, O2;
4       O1 = 2*R1*3.14159265;
5       O2 = 2*R2*3.14159265;
6   }
```

Potrebno je napomenuti da prilikom definisanja simboličkih konstanti nije naveden tip podataka. Za ovim nema potrebe, jer preprocesor jednostavno zamenjuje konstantu na mestima pojavljivanja njenog identifikatora, a tip i ostali sintaksni elementi se proveravaju tamo gde je konstanta upotrebljena. △

Nepisano je pravilo i često se sreće da se imena simboličkih konstanti u programu zadaju velikim slovima, iako sintaksa jezika ne zahteva ovo. Na taj način je programeru koji gleda kod jasno da li se radi o promenljivoj ili simboličkoj konstanti, bez da se vraća na početak programa i proverava.

Primer 3.34 (Primer skraćivanja izraza upotrebom simboličkih konstanti)**.** Simboličkim konstantama moguće je definisati bilo koji niz simbola, a ne samo numeričke konstante. Ovo je ilustrovano sledećim primerom.

```
1   #define PRIKAZI    printf("Rezultat␣je␣%d.",x);
2   #include "stdio.h"
3   main()
4   {
5           int x = 10;
6           PRIKAZI
7   }
```

Kada se u 6. liniji koda pri preprocesiranju naiđe na identifikator simboličke konstante $PRIKAZI$, umesto identifikatora se postavlja kompletan niz simbola iz definicije konstante, uključujući i tačku-zarez na kraju definicije. Ovaj program daje sledeći izlaz:

```
Rezultat je 10.
```

△

Definicija 3.20 (Makroi)**.** Makroi su simboličke konstante sa mogućnošću zadavanja vrednosti prethodno definisanim parametarima u okviru konstante.

Makroi su slični simboličkim konstantama, s tim da se definiciji makroa pridodaju parametari koji se pri korišćenju makroa zamenjuju konkretnim vrednostima. Kaže se da su makroi simboličke konstante sa parametrima. Sintaksa definicije makroa je:

$$\langle makro \rangle \quad ::= \quad \#\textbf{define}\ \langle identifikator \rangle\ '('\ \langle parametri \rangle\ ')'\ \langle niz_simbola \rangle$$

$$\langle parametri \rangle \quad ::= \quad \langle identifikator \rangle\ \{\ ,\ \langle identifikator \rangle\ \}$$

Makro se u programu koristi tako što se iza identifikatora makroa u malim zagradama navedu vrednosti razdvojene zarezom, po jedna za svaki parametar. Preprocesor tada u nizu simbola iz definicije makroa pronalazi parametre i zamenjuje ih vrednostima iz poziva makroa.

Primer 3.35 (Definicija i upotreba makroa)**.** Na početku sledećeg programa definisan je makro *prikazi*. Ovaj makro ima jedan parametar a.

```
1 #define PRIKAZI(a)    printf("Rezultat je %d.\n",a);
2 #include "stdio.h"
3 main()
4 {
5     int x=3,y=1;
6     PRIKAZI(x)
7     PRIKAZI(y)
8 }
```

Poziv makroa iz 6. i 7. linije zamenjuje se nizom simbola

```
printf("Rezultat je %d.\n",a);
```

iz definicije, s tim da se umesto parametra a navode konkretne vrednosti koje stoje u okviru malih zagrada u pozivu makroa. Prethodni program nakon preprocesiranja postaje:

```
1 main()
2 {
3     int x=3,y=1;
4     printf("Rezultat je %d.\n",x);
5     printf("Rezultat je %d.\n",y);
6 }
```

Program daje sledeći izlaz:

```
Rezultat je 3.
Rezultat je 1.
```

△

Uklanjanje definicija

U bilo kojoj fazi izvršenja programa se prethodno uvedena definicija simboličke konstante ili makroa može ukloniti. Preprocesorska direktiva za uklanjanje definicije je `#undef`, nakon koje se navodi identifikator simboličke promenljive ili makroa koji se uklanja:

$$\langle uklanjanje_def.\rangle \quad ::= \quad \#\mathbf{undef}\ \langle identifikator\rangle$$

Simbolička promenljiva ili makro, čija je definicija ukinuta, prestaje da važi počev od linije u kojoj je definicija ukinuta.

Primer 3.36 (Uklanjanje definicija simboličkih konstanti i makroa)**.** Sledeći program ilustruje uklanjanje definicije makroa za kvadriranje zadatog broja $kv(x)$, definisanog u 1. liniji programa. Direktiva za uklanjanje definicije navedena je u 7. liniji. Kompajler će javiti sintaksnu grešku u 8. liniji ovog programa da identifikator kv nije poznat, što nije slučaj sa 6. linijom programa.

```
1 #define kv(x) x*x
2 #include "stdio.h"
3 main()
4 {
5         int R=2, Z=3, S;
6         S = kv(R);
7 #undef kv(x)
8         S = kv(Z);
9 }
```

△

3.9.2 Uključivanje biblioteka u program

Preprocesorska direktiva kojom se u program mogu uključiti druge biblioteke je `#include`.

Ova direktiva navodi se na početku programa i signalizira kompajleru da je u proces prevođenja potrebno uključiti biblioteku koja je navedena kao parametar direktive. U program se može uključiti više biblioteka, s tim da se za svaku biblioteku navodi posebna `#include` direktiva.

Biblioteke su posebni fajlovi na disku koji imaju svoje ime i obično imaju ekstenziju *.h*. Ove biblioteke se nazivaju i *header* fajlovi (*header* - prev. *zaglavlje*), pa otuda i ekstenzija *.h*.

Svrha postojanja biblioteka je da se jednom napisan programski kod koristi u više programa, čime se proces razvoja aplikacija može pojednostaviti i ubrzati. Biblioteke mogu biti standardne biblioteke koje se isporučuju uz kompajler, kao na primer biblioteka *stdio.h*, ili korisnički definisane biblioteke. Korisnički definisana biblioteka nalazi se u posebnom fajlu koji piše programer za potrebe svog (svojih)

programa. Ovako napisana biblioteka se po potrebi može uključiti u jedan ili više programa preprocesorskom direktivom `#include` uz navođenje imena fajla koje je programer zadao.

Biblioteke mogu da sadrže **definicije simboličkih konstanti i makroa**, definicije novih **tipova podataka**, kao i **definicije funkcija**.

Standardna biblioteka koja je do sada korišćena, i koja pored ostalog sadrži i funkcije *printf* i *scanf* za prikaz i unos, nosi naziv *stdio.h.* Što se sintakse tiče, naziv header fajla može se pisati između znakova navoda (" ") ili između znakova < i >. Razlika je samo u skupu putanja na disku gde će kompajler tražiti biblioteku.

Sledeći program uključuje biblioteku za standardni ulaz i izlaz, i biblioteku sa dodatnim matematičkim funkcijama, o kojoj će biti reči u kasnijim poglavljima.

```
1 #include "stdio.h"
2 #include "math.h"
3 void main()
4 {
5         ...
```

Sledeći primer ilustruje način definisanja i upotrebe korisničke biblioteke.

Primer 3.37 (Primer definisanja i upotrebe korisničke biblioteke)**.** Neka su u fajlu "*moje_konstante.h*" definisane simboličke konstante na sledeći način:

```
1 #define Pi    3.14159265
2 #define e     2.71828183
3 #define Plank 6.6260693e-34
```

Tada ovaj fajl predstavlja biblioteku čiji se sadržaj može koristiti u bilo kom programu uključivanjem pomoću preprocesorske direktive `#include`. Primer programa koji koristi prethodno definisanu biblioteku dat je u nastavku.

```
1 #include "stdio.h"
2 #include "moje_konstante.h"
3 main()
4 {
5     double Dirak;
6
7     Dirak = Plank / 2*Pi;
8     printf("Dirakova konstanta je %e.", Dirak);
9 }
```

△

3.9.3 Uslovno prevođenje programa

Preprocesorske direktive za uslovno prevođenje programa omogućavaju uključivanje ili uklanjanje određenih delova programa iz procesa prevođenja programa.

Definicija 3.21 (Uslovno prevođenje programa)**.** Uslovno prevođenje programa je koncept po kome programer može da definiše delove programa koji će u zavisnosti od okolnosti biti uključeni u prevođenje programa, ili će biti u potpunosti isključeni iz procesa prevođenja.

Za uslovno prevođenje koriste se sledeće direktive:

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

Preprocesorska direktiva `#ifdef` definiše početak, a `#endif` kraj dela programa koji će biti uključen u prevođenje ukoliko je simbolička konstanta koja se navodi kao parametar `#ifdef` direktive prethodno definisana, odnosno koji će kompajler preskočiti u suprotnom. Sintaksa je sledeća:

$$\langle \textit{uslovno_prev.} \rangle ::= \begin{array}{l} \quad \langle \textit{deo_programa_koji_se_prevodi} \rangle \\ \#\textbf{ifdef} \ \langle \textit{simbolicka_konstanta} \rangle \\ \quad \langle \textit{uslovni_deo_programa} \rangle \\ \#\textbf{endif} \\ \quad \langle \textit{deo_programa_koji_se_prevodi} \rangle \end{array}$$

Primer 3.38 (Uslovno prevođenje programa)**.** Često se javlja potreba da se prilikom razvoja i pisanja programa prikazuju i međurezultati, a da se ovi prikazi nakon završetka programa što jednostavnije uklone iz programa. Sledeći program ilustruje kako je ovaj problem elegantno moguće rešiti uslovnim prevođenjem programa.

```
1  #define PROVERA
2
3  #include "stdio.h"
4  void main()
5  {
6      int N=10, F=1, i;
7      for (i = 2; i < N; i++)
8      {
9          F = F * i;
10 #ifdef PROVERA
11         printf("%d\n",F);
12 #endif
13     }
14     printf("%d\n",F);
15 }
```

Na početku programa definisana je simbolička konstanta $PROVERA$, a prikaz međurezultata umetnut je u deo programa koji se uslovno prevodi, ukoliko je ova konstanta definisana. Kako ovi međurezultati nisu potrebni u finalnoj verziji programa, već samo krajnji rezultat, jednostavno uklanjanje definicije simboličke konstante rezultovaće da kompajler pri prevođenju programa ne prevodi delove programa u okviru bloka `#ifdef PROVERA`.
△

Preprocesorska direktiva `#ifndef` ima istu sintaksu i efekat kako i `#ifdef`, s tom razlikom da će u slučaju `#ifndef` deo programa biti preveden ako simbolička konstanta **nije** definisana.

Direktiva `#else` se može navesti između preprocesorskih direktiva `#ifdef` i `#endif`, pri čemu deli ovaj deo programa na dva dela: deo koji će se kompajlirati kada je konstanta definisana, i deo koji neće. Ova direktiva se takođe može naći i između `#ifndef` i `#endif`.

Umesto kombinacije direktiva `#else` i `#ifdef`, može se koristiti direktiva `#elif`, na sledeći način:

```
1  #define sk_A
2  #define sk_B
3  main()
4  {
5          ...
6  #ifdef sk_A
7          ... // deo koji ce biti preveden ako je def. sk_A
8  #elif sk_B
9          ... // bice preveden ako nije def. sk_A, ali jeste sk_B
10 #else
11         ... // bice preveden ako nije ni sk_A ni sk_B
12 #endif
13         ...
14 }
```

Preprocesorska direktiva koja omogućava definisanje i ispitivanje složenijih uslova u cilju uslovnog prevođenja je `#if`. Sledeći primer ilustruje upotrebu ove preprocesorske direktive.

Primer 3.39 (Uslovno prevođenje programa)**.** Kao primer upotrebe uslovnog prevođenja programa može se navesti i potreba da se program napiše tako da se jednostavno može preneti sa jedne platforme na drugu i na njoj kompajlirati (npr. sa *Windows* operativnog sistema na *Linux*, *Android*, ili dr.). U ovom sučaju, specifični delovi programa koji se razlikuju od operativnog sistema do operativnog sistema mogu se definisati kao delovi koji se uslovno prevode i mogu se uključivati po potrebi.

Ovaj koncept ilustrovan je sledećim programom.

```
1  #define system LINUX
```

```
2
3  #if system == WINDOWS
4      #include "windows.h"
5  #endif
6  #if system == LINUX
7      #include "linux.h"
8  #endif
9  #if system == ANDROID
10     #include "android.h"
11 #endif
12
13 main()
14 {
15     ...
16 }
```

△

Kontrolna pitanja

1. Nabrojati konvencije koje se koriste za zapisivanje pravila gramatike kod BNF, EBNF i sintaksnih dijagrama.
2. Napisati definiciju heksadekadne cifre u BNF notaciji.
3. Napisati definiciju kompleksnog broja ($Re + i \cdot Im$) u EBNF notaciji.
4. Nacrtati sintaksni dijagram kompleksnog broja ($Re + i \cdot Im$).
5. Kako se po sintaksi pravilno pišu identifikatori u C-u? Koja je njihova namena?
6. Koje od navedenih reči mogu biti identifikatori u programskom jeziku C?

```
kapa, For, 3vrsta, D23, _beta, switch,
yBaR_, %delta, *sigma, !stop
```

7. Nabrojati i objasniti namenu specijalnih karaktera u C-u.

8. Šta je određeno deklaracijom promenljive?

9. Upotrebu funkcije *scanf* objasniti na primeru unosa jednog realnog i jednog celog broja sa tastature. Objasniti format pojedinačne ulazne konverzije.

10. Upotrebu funkcije *printf* objasniti na primeru prikaza jednog realnog i jednog celog broja. Objasniti format pojedinačne izlazne konverzije.

11. Šta će prikazati na ekranu sledeći poziv funkcije *printf*?

```
int x = 4;
float y = .9;
printf("Vdednost x=%-3d,\na y je %-+5.2f",x,y);
```

12. Šta će na ekranu prikazati sledeći deo koda?

```
printf("%4d%4d%4d\n",10,1,-2);
printf("%4d%4d%4d\n",123,321,0);
printf("%4d%4d%4d\n",-1,15,2);
```

13. Nacrtati sintaksni dijagram *for* petlje.

14. Šta prikazuje sledeći deo koda?

```
1       int i,j;
2       for (i = 1; i <= 5; i++)
3       {
4           if (i == 2)
5               continue;
6           for (j = 1; j <= 10; j++)
7           {
8               if (j == 3)
9                   continue;
10              if (j == 6)
11                  break;
12              printf("(%d,%d)\n", i,j);
13          }
14      }
15
```

15. Nacrtati strukturni dijagram toka algoritma i napisati strukturni program umesto nestrukturnog rešenja problema iz primera 3.29.

16. Napisati makro za određivanje kuba zadatog broja.

17. Napisati program na programskom jeziku C koji u slučaju da je zadata simbolička konstanta "FA" određuje i prikazuje faktorijal zadatog broja N, a u slučaju da nije, određuje i prikazuje zbir prvih N prirodnih brojeva.

18. Na programskom jeziku C napisati strukturni program koji za svaku od m grupa realnih brojeva računa i prikazuje sumu i aritmetičku sredinu brojeva. Na početku programa korisnik zadaje broj grupa, a zatim redom za svaku grupu prvo broj brojeva u toj grupi, a nakon toga i same brojeve. Nakon unosa poslednjeg broja iz svake grupe prikazati sumu i aritmetičku sredinu brojeva te grupe.

19. Napisati strukturni program koji određuje i prikazuje sve proste brojeve od 1 do n. Vrednost n zadaje korisnik.

20. Na programskom jeziku C napisati program koji od korisnika traži da unese pozitivan ceo broj, a zatim određuje i prikazuje najveći prost broj, manji ili jednak unetom broju.

4

Promenljive, tipovi podataka i operatori

U prethodnom poglavlju bilo je reči o osnovnim elementima programskog jezika C. Detaljno su razmotreni osnovni elementi i implementacija algoritamskih struktura za kontrolu toka izvršenja programa. Pored projektovanja toka izvršenja algoritma, podjednako je važno razmotriti i podatke koje je potrebno pamtiti za vreme izvršenja programa.

U ovom poglavlju razmotrićemo tipove podataka i operacije koje je moguće izvoditi nad pojedinim tipovima podataka.

4.1 Osnovni tipovi podataka i operatori

Kao što je definicijom 2.4 precizirano (strana 34), promenljiva u programiranju je simboličko ime za lokaciju u memoriji u kojoj je moguće pamtiti vrednosti i čitati ih iz nje. Memorijska lokacija može pamtiti binarni niz nula i jedinica određene dužine.

Za promenljivu je neophodno znati format zapisa u memoriji, odnosno kako "tumačiti" upisane nule i jedinice. Naime, ako kažemo da je sadržaj šesnaestobitne lokacije sa simboličkim imenom A

$$A = (0011\ 1000\ 0000\ 0010)_2,$$

to nije dovoljno. Potrebno je reći u kom formatu je zapis. Ukoliko je celobrojni zapis, tada navedene nule i jedinice predstavljaju dekadni broj 14338. Ukoliko je A realni broj sa 4-bitnim eksponentom i 12-bitnom mantisom u dvojičnom komplementu, tada se radi o broju $-2046 \cdot 10^3$ (ako je baza 10). Naravno, ako je mantisa u nekom drugom formatu, rezultat, odnosno tumačenje sadržaja memorijske lokacije bi bilo drugačije.

U ranijim poglavljima je bilo reči o potrebi za deklaracijom promenljivih u programskim jezicima (poglavlje 3.5, strana 104). Deklaracijom promenljive određuje se:

1. simboličko ime (identifikator) za pristup lokaciji,
2. format zapisa vrednosti promenljive u memoriji,
3. skup vrednosti koje promenljiva može imati, i
4. skup operacija koje se mogu izvršiti (koje su dozvoljene) nad promenljivom.

Definicija 4.1 (Tip promenljive)**.** Tip promenljive određuje: (1) format zapisa u memoriji, (2) skup vrednosti koje promenljiva može imati, i (3) skup operacija koje se mogu izvršavati nad podatkom.

Kod nekih programskih jezika nije neophodno eksplicitno navođenje promenljivih koje će se koristiti u programu, već kompajler sam određuje tip i automatski deklariše promenljivu na osnovu konteksta gde se promenljiva upotrebljava.

Definicija 4.2 (Implicitna deklaracija promenljivih)**.** Implicitna deklaracija promenljivih je mogućnost kompajlera da automatski deklariše promenljivu iz izraza u kom se koristi, a da informacije o tipu dobije iz konteksta izraza.

Ukoliko se kod programskih jezika sa implicitnom deklaracijom u izrazu javi identifikator promenljive, koja se do tog trenutka nije javila ni u jednoj prethodnoj naredbi tog programa, niti postoji ekplicitna deklaracija, automatski se iz konteksta izraza izvlači o kakvoj se promenljivoj radi (celobrojna, realna, znakovna, i sl.), i u tom trenutku se rezerviše prostor za nju, pa tek tada određuje vrednost izraza. Primer jezika sa implicitnom deklaracijom je *Basic* (čita se: "Bejzik").

Napomena: Kod algoritma, kao što se iz poglavlja 2 može zaključiti, koristi se implicitna deklaracija promenljivih. Na početku algoritma nije neophodno navesti listu promenljivih koje će biti korišćene i njihove tipove.

Primer 4.1. Sledeći program napisan je u Bejziku. U programu se implicitno uvodi promenljiva A tako što joj se dodeljuje vrednost izraza $3 + 5$. Nakon toga uvodi se promenljiva C, a za njom i B u izrazu "B=A*C". Potrebno je napomenuti da pre korišćenja nije eksplicitno izvršena deklaracija, a promenljive sam kompajler implicitno definiše i iz konteksta izraza deklariše kao celobrojne.

```
1  Function primer
2      A=3+5
3      C=10
4      B=A*C
5  End Function
```

Definicija 4.3 (Eksplicitna deklaracija promenljivih)**.** Eksplicitna deklaracija promenljive je naredba u programu koja kompajleru eksplicitno daje informaciju o tipu promenljive i njenom simboličkom imenu.

Programski jezik C je **jezik sa eksplicitnom deklaracijom** promenljivih. C ne podržava implicitnu deklaraciju, iako se za algoritamsku reprezentaciju programa u C-u često koristi implicitna deklaracija.

Napomena: Sintaksa C-a za deklaraciju osnovnih tipova promenljivih data je u poglavlju 3.5 na strani 104.

Osnovni tipovi podataka koji su podržani u C-u su:

1. **numerički**,
2. **logički**, i
3. **znakovni**.

Tipom podataka određeni su i operatori koji se u izrazima mogu primenjivati nad operandima (promenljivama).

Po broju operanada operatori se dele na

1. **unarne** – operatori sa jednim operandom,
2. **binarne** – operatori sa dva operanda, i
3. **ternarne** – operatori sa tri operanda.

Po funkcionalnosti, operatori u C-u dele se na:

1. aritmetičke operatore,
2. operatore za rad sa bitovima,
3. logičke operatore,
4. relacione operatore,
5. operator dodele vrednosti,
6. operator grananja,
7. *sizeof* operator,
8. *comma* operator,
9. operator za konverziju tipa, i
10. operatori referenciranja i dereferenciranja.

U narednim poglavljima razmotrićemo osnovne tipove podataka i operatore koji se mogu primenjivati nad njima.

4.1.1 Numerički tipovi podataka

U osnovne numeričke tipove podataka spadaju:

1. celobrojni, i
2. realni podaci.

Celobrojni podaci

Celobrojni podaci (eng. **integer**, čita se *intedžer*, ili skraćno *int*) mogu uzimati vrednosti iz dva skupa: skupa prirodnih brojeva $\mathbb{N}_0 = \{0, 1, 2, \ldots\}$, i skupa celih brojeva $\mathbb{Z} = \{\ldots - 2, -1, 0, 1, 2, \ldots\}$. Programski jezik C poseduje nekoliko tipova podataka za pamćenje celih brojeva. Oni se međusobno razlikuju po tome da li se mogu pamtiti vrednosti iz skupa $\mathbf{Z}$, ili samo vrednosti iz skupa $\mathbf{N}_0$, kao i po broju bitova koji se koriste za binarnu reprezentaciju, što definiše veličinu opsega brojeva koji se mogu predstaviti.

Definicija 4.4 (Označeni i neoznačeni brojevi)**.** Neoznačenim brojem u programiranju naziva se celobrojni podatak za koji se smatra da je pozitivan i u memoriji se ne pamti, niti je predviđeno mesto za pamćenje znaka.
Označeni brojevi su celi brojevi za koje se pamti oznaka da li su pozitivni ili negativni.

Broj bajtova za pamćenje celobrojnih podatka varira od 1 do 8, zavisno od konkretnog celobrojnog tipa. Tako, celobrojni podatak može biti osmobitni (1 bajt), pa sve do 64-bitnog (8 bajtova), ili više.

Skup vrednosti koje celobrojni tip podržava određen je memorijskom reprezentacijom broja koji se pamti. Za predstavljanje celih neoznačenih brojeva koristi se jednostavno binarno kodiranje broja. Ako je broj cifara binarnog broja manji od veličine alociranog memorijskog prostora tada se broj sa leve strane dopunjuje nulama.

Primer 4.2. Dekadni broj $(25)_{10}$ predstavljen u binarnom brojevnom sistemu je $(11001)_2$. Ukoliko se za pamćenje ovog podatka koristi tip podataka za neoznačene cele brojeve veličine 2 bajta, u memoriji će broj biti predstavljen na sledeći način[1]:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	1	**0**	**0**	**1**

△

Opseg brojeva koje je moguće zapamtiti u promenljivoj u formatu neoznačenih celih brojeva je

$$\text{od } 0 \text{ do } 2^b - 1,$$

gde je b broj bitova predviđenih za reprezentaciju broja.

Za pamćenje označenih brojeva koriste se:

1. prosto kodiranje znaka,

2. kodiranje sa pomerajem,

[1]Bit najmanje težine obično se piše zdesna, a oznaka težine svakog bita 2^i piše se iznad cifre, navođenjem samo eksponenta i. Kod ne-težinskih reprezentacija oznaka i predstavlja redni broj bita u memorijskoj lokaciji.

3. nepotpuni komplement (jedinični komplement), i
4. potpuni komplement (dvojični komplement.)

Kod prostog kodiranja znaka, bit najveće težine u binarnoj reprezentaciji se koristi za kodiranje znaka, a ostalih $b-1$ bitova se koriste za direktno kodiranje apsolutne vrednosti broja. Opseg brojeva koje je moguće zapamtiti u promenljivoj sa označenim tipom predstavljenim prostim kodiranjem je

$$\text{od} \quad -\left(2^{b-1}-1\right) \quad \text{do} \quad 2^{b-1}-1,$$

gde je b broj bitova predviđenih za reprezentaciju broja [5].

Kod kodiranja sa pomerajem, vrednost V se transformiše kao:

$$V_p = V + q^{n-1} - 1,$$

gde je q osnova brojevnog sistema. Opseg vrednosti koje je moguće zapamtiti je isti kao i kod prostog kodiranja znaka. Na primer, ukoliko imamo 1 bajt na raspolaganju, reprezentacija dekadnog broja $(3)_{10}$ bila bi $V_p = 3+2^7-1 = (130)_{10} = (10000010)_2$, a reprezentacija broja $(-3)_{10}$ bila bi $V_p = -3+2^7-1 = (124)_{10} = (01111100)_2$.

Jedinični komplement za kodiranje pozitivnih brojeva koristi prosto kodiranje broja, a za kodiranje negativnih brojeva komplementira se svaka cifra binarne reprezentacije apsolutne vrednosti broja. Da li je broj pozitivan ili negativan zna se na osnovu cifre najveće težine (0 - pozitivan, 1 - negativan). Na primer, $(3)_{10} = (00000011)$, a $(-3)_{10} = (11111100)_2$. Opseg brojeva kod jediničnog komplementa isti je kao i opseg brojeva kod prostog kodiranja znaka.

Razlika između jediničnog i dvojičnog komplementa je u negativnom opsegu brojeva. Zbog načina kodiranja jedinični komplement za broj 0 ima dva koda (vidi [5]). Dvojični komplement je isti kao i jedinični, uz razliku da se apsolutnoj vrednosti negativnih brojeva dodaje 1 da bi se izbegao problem postojanja dva koda za nulu. Na primer, $(3)_{10} = (00000011)$, a $(-3)_{10} = (11111100)_2 + 1 = (11111101)_2$. Opseg brojeva koje je moguće zapamtiti u promenljivoj sa označenim tipom predstavljenim dvojičnim komplementom

$$\text{od} \quad -2^{b-1} \quad \text{do} \quad 2^{b-1}-1.$$

U dvojičnom komplementu se može kodirati jedan negativan broj više u odnosu na jedinični komplement (broj -2^{b-1}).

Ključne reči za deklaraciju celobrojnih tipova podataka u programskom jeziku C su **char**, **short**, **int** i **long**, ispred kojih može stajati ključna reč **signed** ili **unsigned**.

Celobrojni tip u EBNF notaciji može se predstaviti na sledeći način:

⟨*celobrojni_tip*⟩ ::=
[**signed** | **unsigned**] (**char** | **short** | **int** | **long**)

Opcionim ključnim rečima *signed* i *unsigned* eksplicitno se naglašava da se radi o označenim (eng. *signed*), ili o neoznačenim brojevima (eng. *unsigned*). Tipovi *char*, *short*, *int* i *long* određuju broj bitova *b* u binarnoj reprezentaciji broja.

Standardi koji opisuju programski jezik C **ne definišu** tačan broj bajtova za tipove promenljivih, već minimalan broj bajtova. Svaki C kompajler mora da podrži najmanje onoliko bajtova koliko je definisano standardom, a može imati i više. Sa druge strane, C standard ne definiše ni memorijsku reprezentaciju, jer različiti procesori mogu raditi sa različitim reprezentacijama, pa zato za svaki tip procesora (preciznije, arhitekturu) postoji poseban C kompajler. Zbog toga, kada je standardom definisano da se u 8-bitnoj memorijskoj lokaciji pamte označeni brojevi, standard kaže da je minimalni opseg od -127 do 127, a ne od -128 do 127, kako bi se obuhvatile sve mogućnosti zbog razlike koja postoji između jediničnog i dvojičnog komplementa. Ukoliko je u konkretnom kopajleru za konkretan procesor implementiran dvojični komplement, opseg će biti od -128 do 127, što svakako pokriva opseg koji zahteva standard.

U tabeli 4.1 prikazani su **svi celobrojni tipovi** programskog jezika C. Za svaki tip dat je minimalni broj bajtova po standardu, da li se radi o označenom broju ili ne, i minimalni i maksimalni broj iz opsega brojeva, koji se može predstaviti tipom po standardu. Pored ovih podataka, uporedo je dat i broj bajtova u okruženju *Microsoft Visual Studio 2010*. Ovaj broj se razlikuje jedino u slučaju tipa *int*.

Potrebno je napomenuti da je počev od specifikacije standarda C99 uveden i dodatni tip *long long* za ekstremno velike brojeve. Podaci o ovom tipu takođe su dati u tabeli 4.1.

Primer 4.3 (Broj bajtova u memorijskoj reprezentaciji)**.** Opseg brojeva se razlikuje od kompajlera do kompajlera, a koliko tačno bajtova zauzima određeni podatak moguće je dobiti pomoću **sizeof**. Sledeći kod prikazuje koliko bajtova zauzimaju promenljive tipa *char* i *int*.

```
1  #include "stdio.h"
2  main()
3  {
4      int x;
5      char c;
6      int bx,bc; // promenljive koje ce pamtiti broj bajtova
7      bx = sizeof(x);
8      bc = sizeof(c);
9          printf("x zauzima %d, a c zauzima %d bajt.\n", bx, bc);
10 }
```

Izlaz programa 4.0

x zauzima 4, a c zauzima 1 bajt.

.

Ukoliko se program prevede i izvrši u okruženju *Microsoft Visual Studio 2010* daće izlaz koji je prikazan. Kao što se iz izlaza može zaključiti, tip *char* je u skladu sa standardom, kao i za tip *int*, s tim da tip *int* nudi mnogo veći opseg brijeva. △

Tip	Min br. bajtova	Označeni / Neoznačeni	Min.	Max.	Br. bajtova VS2010
char	1	Oz ili Ne	-127 ili 0	127 ili 255	1
signed char	1	Oz	-127	127	1
unsigned char	2	Oz	0	255	1
short short int signed short signed short int	2	Oz	-32.767	32.767	2
unsigned short unsigned short int	2	Ne	0	65.535	2
int signed int	2	Oz	-32.767	32.767	4
unsigned unsigned int	2	Ne	0	65.535	4
long long int signed long signed long int	4	Oz	-2.147. 483.647 ($= -2^{31}$)	2.147. 483.647 ($= 2^{31}$)	4
unsigned long unsigned long int	4	Ne	0	4.294. 967.295 ($= 2^{32}$)	4
long long long long int signed long long signed long long int	8	Oz	-9.223. 372.036. 854.775.807 ($= -2^{63}$)	-9.223. 372.036. 854.775.807 ($= 2^{63}$)	8
unsigned long long unsigned long long int	8	Ne	0	18.446. 744.073. 709.551. 615 ($= 2^{64}$)	8

* Oz – označen ceo broj u dvojičnom komplementu
* Ne – neoznačen ceo broj

Tabela 4.1: Celobrojni tipovi programskog jezika C

Celobrojnim promenljivama moguće je dodeljivati vrednosti celobrojnih konstanti (vidi poglavlje 3.3.4 na strani 96 i definiciju 3.7). Interesantno je razmotriti šta se dešava sa promenljivom ukoliko joj se dodeli vrednost koja je van opsega. Ova situacija razmotrena je u sledećem primeru.

Primer 4.4 (Dodela vrednosti van opsega promenljive)**.** U sledećem programu označenim i neoznačenim promenljivama tipa *char* dodeljivane su različite vrednosti van opsega i prikazivani su rezultati.

```
1 #include "stdio.h"
```

```
 2 main()
 3 {
 4     signed char sc;
 5     unsigned char uc;
 6     sc = 129;
 7     uc = 257;
 8     printf("oznaceni␣umesto␣129␣ima␣vrednost␣%d,\na␣neoznaceni␣umesto␣257␣ima␣
       vrednost␣%d\n",sc,uc);
 9     sc = -129;
10     uc = -1;
11     printf("oznaceni=%d\nneoznaceni=%d",sc,uc);
12 }
```

Izlaz programa 4.0

oznaceni umesto 129 ima vrednost -127,
a neoznaceni umesto 257 ima vrednost 1
oznaceni=127
neoznaceni=255

Program je prevođen u okruženju *Microsoft Visual Studio 2010*. Kao što se može videti, sintaksno je ispravno dodeliti vrednosti van opsega, ali konkretna vrednost koja će biti upisana u memorijsku lokaciju zavisi od prevodioca. U slučaju označenog broja iz linije 6, vrednost 129 je prevedena u binarni zapis kao $uc = (10000001)_2$, pa je jedinica na mestu bita najveće težine prilikom prikaza protumačena kao znak. Kako je vrednost koja je prikazana na izlazu -127, može se zaključiti da je broj zapamćen u dvojičnom komplementu. △

Realni podaci

Promenljive realnog tipa mogu uzimati vrednosti iz skupa realnih brojeva. Najčešće se realni brojevi pamte u formatu sa pokretnim zarezom (eng. *floating point*)[2]. Broj predstavljen u pokretnom zarezu je broj napisan u obliku

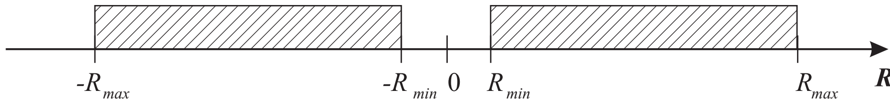
$$V = m \cdot q^e,$$

gde je V vrednost broja, m mantisa sa svojim celobrojnim i razlomljenim delom, q osnova brojevnog sistema, a e celobrojni eksponent. Da bi se broj zapamtio u pokretnom zarezu dovoljno je memorisati eksponent e i mantisu m, jer je osnova q ista kod svih brojeva konkretnog procesora (obično $q = 2$ ili $q = 10$), pa je nema potrebe pamtiti, već se podrazumeva.

Opseg brojeva koje je moguće predstaviti realnim tipom prikazan je na slici 4.1. Jedna osobina koja proizlazi iz načina predstavljanja realnih brojeva u pokretnom zarezu je da je razlika između dva uzastopna broja koje je moguće predstaviti manja

[2]Pored reprezentacije u pokretnom zarezu postoji i tzv. reprezentacija u *fiksnom zarezu* (eng. *fixed point*) [5]

što je apsolutna vrednost broja manja, a veća ukoliko su brojevi po apsolutnoj vrednosti veći. Zbog ovoga se govori o preciznosti reprezentacije broja u binarnom zapisu. Preciznost je veća, odnosno rastojanje između brojeva je manje ukoliko su brojevi bliži nuli. Takođe, preciznost u celom opsegu je veća, naravno, ukoliko je na raspolaganju veći broj bitova za memorisanje mantise. Na veličini opsega dobija se povećavanjem broja bitova za reprezentaciju eksponenta. Postoji jedna okolina tačke 0 koju nije moguće predstaviti reprezentacijom broja u pokretnom zarezu, kao što je prikazno na slici 4.1.



Slika 4.1: Opseg realnih brojeva

Posmatrano u apsolutnim vrednostima, minimalna vrednost različita od nule je

$$\mid R_{min} \mid = \mid m_{min} \mid \cdot q^{e_{min}},$$

gde je m_{min} najmanja vrednost mantise različita od nule, a e_{min} najmanji ceo broj u datoj reprezentaciji. Maksimalna vrednost je

$$\mid R_{max} \mid = \mid m_{max} \mid \cdot q^{e_{max}},$$

gde su m_{max} i e_{max} najveće moguće vrednosti mantise i eksponenta, respektivno.

Realni brojevi se u memoriji pamte u normalizovanom obliku, tako da važi

$$1/q \leq m < 1, \quad (4.1)$$

ill, češće sretani oblik za reprezentaciju binarnih brojeva, za koji važi

$$1 \leq m < q. \quad (4.2)$$

Praktično gledano, u prvom slučaju (izraz 4.1) eksponent se bira tako da je celobrojni deo mantise uvek 0, a prvi broj iza decimalnog zareza različit od nule. Na primer, ako je $q = 10$, normalizovani oblik vrednosti $V = 12.375$ je $V = 0.12375 \cdot 10^2$, za koju je očigledno da važi izraz (4.1).

Za slučaj kada je oblik normalizacije binarnog broja dat sa (4.2) celobrojni deo mantise uvek je 1. Na primer, ako je $q = 2$, normalizovani oblik vrednosti $V = (12.375)_{10} = (12)_{10} + (0.375)_{10} = (1100)_2 + (0.011)_2 = (1100.011)_2$ je $V = (1.100011)_2 \cdot 2^3$.

U oba slučaja celobrojni deo se može podrazumevati, jer je isti za sve normalizovane brojeve konkretnog procesora, pa se u memoriju može upisati samo deo iza decimalne tačke.

Kod memorijske reprezentacije realnih brojeva u jednom delu memorijskog prostora pamti se mantisa, a u drugom delu eksponent. Broj bajtova za memorijsku

reprezentaciju je obično 2, 4, 8, itd. Programski jezik C ima tri tipa podataka za predstavljanje realnih brojeva. Realni tip u EBNF notaciji može se predstaviti na sledeći način:

$$\langle realni_tip \rangle \ ::= \ (\textbf{float} \mid [\textbf{long}]\ \textbf{double})$$

Standard ne definiše većinu osobina ovih tipova, sem minimalnih limita. Međutim, na većini sistema se za *float* podatke koristi IEEE 754 standard za jednostruku preciznost (eng. *IEEE 754 single-precision*), za *double* se koristi IEEE 754 standard za dvostruku preciznost (eng. *IEEE 754 double-precision*), i za *long double* se koristi IEEE 754 standard za četvorostruku preciznost (eng. *IEEE 754 quadruple-precision*). Za svaki od ovih tipova je u tabeli 4.2 prikazan ukupan broj bitova b, broj bitova znaka s, broj bitova eksponenta e, broj bitova mantise m, kao i minimalna i maksimalna vrednost (R_{min} i R_{max}) koje je moguće zapamtiti. Takođe je dat i broj bajtova koji je predviđen za ove tipove u okruženju *Microsoft Visual Studio 2010*.

S obzirom da se celobrojni deo kod normalizovanog zapisa može izostaviti, za mantisu od 24 bita potrebno je zapamtiti samo 23. Ove vrednosti su u koloni m tabele 4.2 dati kao **24 (23)**, odnosno, dat je broj bitova mantise, a u zagradi je dato koliko se zapravo bitova mantise pamti u memoriji.

Tip	b	s	e	m	R_{min}	R_{max}	Br. bajtova VS2010
float	32	1	8	24 (23)	$\approx 1.18 \cdot 10^{-38}$	$\approx 3.4 \cdot 10^{38}$	4
double	64	1	15	113 (112)	$\approx 3.36 \cdot 10^{-4932}$	$\approx 1.18 \cdot 10^{4932}$	8
long double	128	1	100	28	0	255	8

Tabela 4.2: Realni tipovi programskog jezika C

Razmotrimo *float* tip, uz napomenu da sve, sem dužina m i e, važi i za ostale tipove.

Bit koji predstavlja znak s je znak mantise. Eksponent je ceo broj u opsegu od -128 do 127 u dvojičnom komplementu ili kodiran sa pomerajem, zavisno od implementacije. Mantisa uključuje 23 binarne cifre desno od decimalne tačke. Raspored bitova u 4 bajta, koliko je predviđeno za tip *float* prikazan je na slici 4.2.

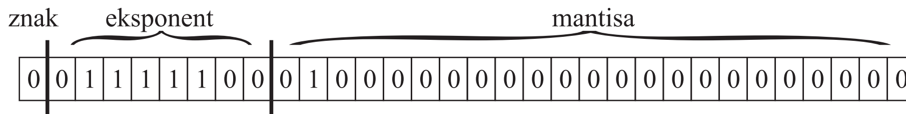
Na osnovu prethodnog, vrednost se može izračunati kao

$$V = (-1)^s \cdot (1.b_{22}b_{21}\ldots b_0) \cdot 2^{e-127},$$

pa je vrednost iz primera prikazanog na slici 4.2 jednaka

$$V = (-1)^0 \cdot (1.01000)_2 \cdot 2^3 = (0.15625)_{10}.$$

Prevedeno na dekadni brojni sistem, preciznost *float* podatka je takva da je moguće zapamtiti broj sa 6 do 9 decimala. Tip *double* ima preciznost na oko



Slika 4.2: Memorijska reprezentacija podataka *float* tipa po IEEE 754 formatu jednostruke tačnosti

15 dekadnih decimala, dok *long double* treba da ima preciznost na 36 decimala. Međutim, kako je prikazano u tabeli 4.2, veličina memorijskog prostora za tip *long double* u okruženju Visual Studio 2010 odstupa od standarda i jednaka je veličini predviđenoj za tip *double*.

Realnim promenljivama moguće je dodeljivati vrednosti realnih konstanti (vidi poglavlje 3.3.4 na strani 96). Interesantno je razmotriti šta se dešava sa promenljivom ukoliko joj se dodeli vrednost koja je van opsega. Ova situacija razmotrena je u sledećem primeru.

Primer 4.5 (Dodela vrednosti van opsega realnoj promenljivoj)**.** U sledećem programu realnoj promenljivoj sa jednostrukom tačnošću dodeljivane su vrednosti van opsega i prikazivani su rezultati.

Program 4.1

```
1  #include "stdio.h"
2  main()
3  {
4      float x;
5      // na granici opsega
6      x = 1e-38;
7      printf("%e\n",x);
8      // u okolini nule
9      x = 1e-50;
10     printf("%e\n",x);
11     // veliki broj
12     x = 1e+50;
13     printf("%e\n",x);
14 }
```

Izlaz programa 4.1

9.999999e-039
0.000000e+000
1.#INF00e+000
.

Ukoliko je broj veći od opsega, što je slučaj sa 12. linijom koda, u promenljivu se upisuje specijalna binarna kombinacija definisana IEEE 754 standardom $e =$

$(11111111)_2$ i $m = 0$, što predstavlja beskonačnost, odnosno na engleskom *infinity*, od čega je i skraćenica INF u 3. redu izlaza. △

4.1.2 Znakovni podaci

Često se u programiranju javlja potreba za obradom reči i rečenica govornog jezika, potreba za obradom teksta generalno, ili jednostavno prikazivanje tekstualnih poruka. Za kodiranje znakovnih podataka u programskim jezicima najčešće se koristi ASCII standard[3].

Osnovni ASCII standard specificira numeričke kodove za 128 simbola. Numerički kodovi su u opsegu od 0 do 127. Preslikavanje "simbol - kod" dato je tzv. ASCII tabelom. Simboli ASCII tabele su:

1. mala i velika slova engleske abecede,
2. decimalne cifre od 0 do 9,
3. znaci interpunkcije, i
4. specijalni simboli.

Svi simboli i odgovarajući kodovi ACSII tabele prikazani su na slici 4.3.

000	(nul)	016	► (dle)	032	sp	048	0	064	@	080	P	096	`	112	p
001	☺ (soh)	017	◄ (dc1)	033	!	049	1	065	A	081	Q	097	a	113	q
002	☻ (stx)	018	↕ (dc2)	034	"	050	2	066	B	082	R	098	b	114	r
003	♥ (etx)	019	‼ (dc3)	035	#	051	3	067	C	083	S	099	c	115	s
004	♦ (eot)	020	¶ (dc4)	036	$	052	4	068	D	084	T	100	d	116	t
005	♣ (enq)	021	§ (nak)	037	%	053	5	069	E	085	U	101	e	117	u
006	♠ (ack)	022	▬ (syn)	038	&	054	6	070	F	086	V	102	f	118	v
007	• (bel)	023	↨ (etb)	039	'	055	7	071	G	087	W	103	g	119	w
008	◘ (bs)	024	↑ (can)	040	(	056	8	072	H	088	X	104	h	120	x
009	(tab)	025	↓ (em)	041	)	057	9	073	I	089	Y	105	i	121	y
010	(lf)	026	(eof)	042	*	058	:	074	J	090	Z	106	j	122	z
011	♂ (vt)	027	← (esc)	043	+	059	;	075	K	091	[	107	k	123	{
012	♀ (np)	028	∟ (fs)	044	,	060	<	076	L	092	\	108	l	124	|
013	(cr)	029	↔ (gs)	045	-	061	=	077	M	093	]	109	m	125	}
014	♫ (so)	030	▲ (rs)	046	.	062	>	078	N	094	^	110	n	126	~
015	☼ (si)	031	▼ (us)	047	/	063	?	079	O	095	_	111	o	127	⌂

Slika 4.3: ASCII tabela

Definicija 4.5 (Karakter)**.** Karakterom u programiranju naziva se jedan simbol ASCII tabele.

Napomena: ASCII tabela nije jedini standard za kodiranje znakovnih podataka. Kao što se sa slike 4.3 može primetiti, u tabeli ne postoje slova č, ć, š, ž i đ. Takođe, u tabeli se nalazi samo latinično pismo. UNICODE je skup standarda, od kojih su

[3]ASCII - American Standard Code for Information Interchange

najpoznatiji UTF-8 i UTF-16. Bazična verzija UNICODE-a je dvobajtni format zapisa do $2^{16} = 65536$ karaktera. Sa 65536 karaktera rešen je problem zapisa skoro svih postojećih pisama.

Specijalni simboli su "nevidljivi" karakteri za koje se mogu naći tasteri na tastaturi. Na primer, ASCII kod za taster $\langle enter \rangle$ na tastaturi je 13, koji je predstavljen u ASCII tabeli na slici 4.3 kao *cr*, od engleskih reči *carriage return*. Takođe, u tabeli je i ASCII kod za taster $\langle escape \rangle$ - 27, itd. Neki specijalni simboli ne mogu se otkucati na tastaturi, ali imaju posebno značenje u programiranju, kao na primer *null* (kod 0), o čemu će biti reči u kasnijim poglavljima.

Potrebno je istaći da su ASCII oznake za karaktere koji označavaju dekadne cifre od '0' do '9' nalaze od koda 048 do koda 057, odnosno, posmatrano u heksadekadnom sistemu od koda 0x30 do 0x39.

Za pamćenje karaktera u memoriji, ili preciznije, za pamćenje numeričkih ASCII kodova željenih simbola, može se koristiti bilo koji celobrojni tip iz tabele 4.1. Međutim, zbog toga što je opseg ASCII kodova od 0 do 127, najčešće se koristi tip **char**. Tip *char* je zbog ove svoje upotrebe i dobio ime - *char*, skraćeno od engleske reči *character*.

Napomena: Jedna promenljiva tipa *char* može zapamtiti jedan simbol. Više uzastopnih simbola, odnosno reči i rečenice, pamte se u složenim strukturama, tzv. poljima, o kojima će više biti reči u narednim poglavljima.

Promenljivoj karakter tipa moguće je dodeliti numeričku vrednost željenog karaktera iz ASCII tabele, kao u sledećem primeru.

```
1  main()
2  {
3      char x=65;
4      printf("%d\n",x);
5      printf("%c\n",x);
6  }
```

Ako promenljivu x iz prethodnog primera prikažemo kao celobrojni podatak funkcijom *printf* sa konverzionim karakterom **%d**, prikazana vrednost će biti dekadni broj 65. Međutim, ako celobrojnu promenljivu x prikažemo konverzionim karakterom **%c**, kompajler prikazuje simbol koji odgovara ASCII kodu 65, što je veliko slovo **'A'**.

Da bi se izbeglo pamćenje simbola ASCII tabele, na raspolaganju u programskom jeziku C su znakovne konstante. Znakovna konstanta u ASCII kodu u C-u je simbol napisan između apostrofa ('). Na taj način, potpuno je ekvivalentno celobrojnoj promenljivoj dodeliti dekadnu konstantu 65, ili znakovnu konstantu 'A'. U oba slučaja u memorijskoj lokaciji će biti zapamćena vrednost 65, kao što je pokazano na primeru sledećeg programa.

Program 4.2

```
1  main()
```

```
2  {
3      char x='A';
4      printf("%d\n",x);
5      printf("%c\n",x);
6  }
```

Izlaz programa 4.2

65
A

Imajući u vidu da se u memoriji svakako pamti numerička vrednost, a da je stvar prikaza da li će vrednost biti prikazana kao dekadna ili će se prikazati odgovarajući simbol iz ASCII tabele, trivijalno je napisati program koji prikazuje celu ASCII tabelu. Program koji prikazuje ASCII tabelu dat je u sledećem primeru.

Primer 4.6 (Prikaz svih simbola i odgovarajućih kodova ASCII tabele)**.** Napisati program na programskom jeziku C koji prikazuje sve simbole i njima odgovarajuće kodove ASCII tabele. Numeričke kodove simbola predstaviti u dekadnom i heksadekadnom brojnom sistemu.

```
1  main()
2  {
3      for (char i = 0; i < 127; i++)
4          printf("%c␣%4d␣%4x\n",i,i,i);
5  }
```

Program će dati sledeći izlaz:

```
...
!   33   21
"   34   22
#   35   23
$   36   24
...
0   48   30
1   49   31
2   50   32
...
A   65   41
B   66   42
...
a   97   61
b   98   62
...
```

Izlaz je zbog veličine prikazan samo delimično. △

Interesantno je napomenuti da je nad karakter podacima, kako su u pitanju numerički kodovi, moguće izvoditi i aritmetičke operacije. Ovo je ilustrovano u sledećem primeru.

Primer 4.7 (Operacije nad ASCII kodovima)**.** Napisati program na programskom jeziku C koji prikazuje simbol koji se u ASCII tabeli nalazi iza simbola za slovo 'A'.

```
1 main()
2 {
3     char x = 'A';
4     char y;
5     y = x + 1; // u y ce biti upisan kod 66
6     printf("%c",y);
7 }
```

Izlaz iz prethodnog programa je slovo 'B'. △

4.1.3 Aritmetički operatori i operatori za rad sa bitovima

Aritmetički operatori i operatori za rad sa bitovima mogu biti unarni i binarni. Unarni operatori imaju jedan operand i mogu stajati sa leve ili desne strane promenljive, zavisno od operatora, dok binarni operatori imaju dva operanda, levi i desni.

Aritmetički operatori

Unarni aritmetički operatori se mogu primeniti nad promenljivama bilo kog numeričkog tipa, celobrojnog, realnog ili znakovnog. Unarni aritmetički operatori programskog jezika C su:

- operatori za promenu znaka vrednosti promenljive, i
- operatori inkrementiranja i dekrementiranja.

Operatori za promenu znaka vrednosti promenljive označavaju se simbolima + i -, koji se pišu sa leve strane promenljive. U EBNF notaciji ovo se može zapisati kao

$$\langle unarni_+i- \rangle \quad ::= \quad [+ \mid -]\ \langle promenljiva \rangle$$

Sledeći program ilustruje upotrebu unarnog operatora '-'.

```
1 main()
2 {
3     int x = 10, y;
4     y = -x;
5     printf("%d",y);
6 }
```

Ovaj program će na izlazu prikazati vrednost -10.

Definicija 4.6 (Inkrementiranje i dekrementiranje)**.** Inkrementiranje, od engleske reči *increment* je povećanje vrednosti. Dekrementiranje (eng. *decrement*) predstavlja smanjenje vrednosti.

Operatori inkrementiranja i dekrementiranja u C-u povećavaju, odnosno smanjuju vrednost promenljive uz koju stoje za 1. Čine ih po dva simbola `++` i `--`, i mogu se pisati i sa leve i sa desne strane promenljive. Ukoliko se pišu sa leve strane promenljive primenjuju se pre bilo koje druge operacije u izrazu, t.j. prvo se vrednost promenljive inkrementira, odnosno dekrementira, pa se tek onda izvršavaju ostale operacije u izrazu. Operator napisan sa leve strane naziva se *prefiksni* operator.

Ukoliko je operator u *postfiksu* (sa desne strane promenljive), tada se najpre izvršavaju sve operacije izraza, pa tek nakon dodele vrednosti i unarni operatori inkrementiranja, odnosno dekrementiranja.

Sledeći primer ilustruje prefiksnu, odnosno postfiksnu notaciju operatora inkrementiranja i dekrementiranja.

Program 4.3

```
1  main()
2  {
3      int x = 5, y = 5, z;
4      float a = 3, b = 3, c;
5
6      z = x + y++;
7      printf("x=%d, y=%d, z=%d\n", x,y,z);
8
9      c = --a + --b;
10     printf("a=%2.0f, b=%2.0f, c=%2.0f", a,b,c);
11
12 }
```

Izlaz programa 4.3

x=5, y=6, z=10
a=2, b=2, c=4

Promenljivoj y u 6. liniji koda vrednost je povećana tek nakon dodele vrednosti promenljivoj z. Promenljivama a i b je vrednost smanjena pre računanja vrednosti izraza i dodele vrednosti promenljivoj c. Ekvivalentan kod izrazu u 6. liniji programa 4.3 je

```
1  z = x + y;
2  y = y + 1;
```

Kod ekvivalentan izrazu iz linije 9. je

```
1 a = a - 1;
2 b = b - 1;
3 c = a + b;
```

Osnovne aritmetičke operacije koje je moguće izvoditi nad promenljivama numeričkih tipova su sabiranje, oduzimanje, množenje, deljenje i određivanje ostatka pri deljenju, odnosno modula broja. Simboli operatora ovih operacija su redom:

$$+, -, *, /, \%$$

Sve navedene operacije moguće je izvršiti nad promenljivama celobrojnog i znakovnog tipa, dok je nad realnim podacima moguće izvršiti sve sem određivanja ostatka pri deljenju.

Sledeći program ilustruje upotrebu binarnog operatora za određivanje ostatka pri deljenju.

```
1 main()
2 {
3     unsigned short x = 10, y = 4, z;
4     z = x % y;
5     printf("%d", z);
6 }
```

Ovaj program na izlazu prikazuje vrednost 2.

Napomena: Operator za određivanje ostataka pri deljenju često se koristi za određivanje parnosti broja. U tom slučaju se proverava da li je ostatak pri deljenju nekog broja brojem 2 jednak 0 ili ne: $if(x\%2 == 0)$.

Operatori za rad sa bitovima

Programski jezik C jedan je od retkih jezika koji ima ovu vrstu operatora. Ovi operatori omogućavaju manipulaciju nad podacima "na niskom nivou" i približavaju jezik C asemblerskim jezicima.

Operatore za rad sa bitovima moguće je primeniti jedino nad **celobrojnim** tipovima podataka. Operacije nad bitovima, za razliku od aritmetičkih operacija, direktno modifikuju bitove u zapisu broja, bez razmatranja aritmetičkih vrednosti brojeva.

Unarni operator za rad sa bitovima promenljive je:

- ~ – negacija vrednosti promenljive bit po bit,

i piše sa leve strane promenljive, dok su binarni operatori

- **&** – logičko "I" odgovarajućih bitova operanda (eng. *and*),
- | – logičko "ILI" odgovarajućih bitova operanda (eng. *or*),
- ^ – logičko ekskluzivno "ILI" odgovarajućih bitova operanda (eng. *xor*),

- `>>` – aritmetičko[4] pomeranje sadržaja u desno (eng. *shift right*),
- `<<` – aritmetičko pomeranje sadržaja u levo (eng. *shift left*.

Operator ∼ vrši negaciju svakog bita binarnog zapisa broja. Da bi se predvideo ishod, potrebno je poznavati način reprezentacije brojeva konkretnog C kompajlera. Sledeći primer ilustruje ovaj operator.

Primer 4.8 (Negacija bitova promenljivih)**.** Sledeći program vrši bitsku negaciju vrednosti dve promenljive, x i a, i negirane vrednosti upisuje u promenljive y i b, respektivno.

```
1 main()
2 {
3     char x = 3, y;
4     char a = -3, b;
5     y = ~x;
6     printf("%d\n",y);
7     b = ~a;
8     printf("%d",b);
9 }
```

Nakon prevođenja i izvršavanja u okruženju *Microsoft Visual Studio 2010*, program daje sledeći izlaz:

```
-4
2
```

Razlog zbog koga je negirana vrednost $x = 3$ jednaka -4 je sledeći. Naime, binarna reprezentacija broja 3 je $(3)_{10} = (0000\ 0011)_2$. Ukoliko se svaki bit negira (5. linija koda) vrednost postaje $y = (1111\ 1100)_2$. Kako je cifra najveće težine (krajnje levo) jednaka 1, to znači da se radi o negativnom broju. S obzirom da se kod ovog kompajlera negativni brojevi predstavljaju u dvojičnom komplementu, dekadni broj čija je binarna reprezentacija $y = (1111\ 1100)_2$ je $y = -(|3| + 1) = -4$, jer, kako je ranije rečeno, kod dvojičnog komplementa se negativnim brojevima u apsolutnom iznosu dodaje 1 da bi se izbeglo dupliranje kodova.

Sličan zaključak se može izvesti i iz činjenice da je negacija broja $a = -3$ kao rezultat dala broj $b = 2$.

△

Logičko "I" (&), logičko "ILI (|) i logičko ekskluzivno "ILI" (^) su binarne operacije nad operandima, koje se izvode tako što se odgovarajuća operacija primeni nad odgovarajućim bitovima operanada. Tako, za slučaj operacije logičko "I" vrši se logičko "I" nad bitom najmanje težine levog operanda i bitom najmanje težine desnog operanda i rezultat upisuje u bit najmanje težine rezultata, pa nad narednim bitom, itd.

[4]Kod aritmetičkog pomeranja sadržaja znak broja nije uključen u pomeranje, pa tako broj zadržava znak, a pomera se samo apsolutna vrednost broja.

Primer 4.9 (Bitsko "I")**.** Sledeći program ilustruje upotrebu binarnog operatora &.

```
1 main()
2 {
3     short a = 12, b = 7;
4     printf("%d", a & b);
5 }
```

Nakon prevođenja i izvršavanja, program će dati sledeći izlaz:

```
4
```

Kako je binarna reprezentacija broja $a = (12)_{10} = (0000\ 1100)_2$, a binarna reprezentacija broja $b = (7)_{10} = (0000\ 0111)_2$, rezultat logičke operacije "I" nad odgovarajućim bitovima je:

$$\begin{aligned} a\&b &= \\ &= (0000\ 1100)_2\ \& \\ &\& (0000\ 0111)_2 = \\ &\overline{} \\ &= (0000\ 0100)_2 \end{aligned}$$

što predstavlja binarnu reprezentaciju dekadnog broja 4. △

Što se operatora za pomeranje sadržaja tiče, u najjednostavnijoj formi, izraz sa operatorima za aritmetičko pomeranje sadržaja u levo ili desno može se predstaviti u EBNF notaciji kao:

$$\begin{aligned} \langle pomeranje \rangle &::= \\ &\langle \mathbf{levi_operand} \rangle \langle \mathbf{operator} \rangle \langle \mathbf{desni_operand} \rangle \\ \\ \langle operacija \rangle &::= \ << \mid >> \\ \langle levi_operand \rangle &::= \langle promenljiva \rangle \mid \langle konstanta \rangle \mid \langle izraz \rangle \\ \langle desni_operand \rangle &::= \langle promenljiva \rangle \mid \langle konstanta \rangle \mid \langle izraz \rangle \end{aligned}$$

Drugim rečima, u izrazu u kome figuriše binarni operator za pomeranje binarnog sadržaja lokacije, operator je jedan od dvokarakternih oznaka '<<' ili '>>', a operandi bilo promenljive, konstante ili izrazi.

Funkcija operatora je da binarnu reprezentaciju levog operanda pomeri za onoliko bitova ulevo (`<<`) ili udesno (`>>`) kolika je vrednost desnog operanda.

Primer 4.10 (Pomeranje bitova promenljive udesno)**.** Sledeći program ilustruje upotrebu binarnog operatora '`>>`'.

```
1  main()
2  {
3      short a = 12, b = 2;
4      printf("%d", a >> b);
5  }
```

Nakon prevođenja i izvršavanja, program će dati izlaz

```
3
```

Binarna reprezentacija broja $(12)_{10}$ je $(0000\ 1100)_2$. Nakon pomeranja sadržaja lokacije $a = 12$ za 2 mesta udesno dobija se $(0000\ 0011)_2 = (3)_{10}$. $\triangle$

4.1.4 Logički podaci

Skup vrednosti logičkih promenljivih je "*tačno*" i "*netačno*" (eng. TRUE i FALSE), odnosno matematičkim simbolima zapisano $\{\top, \bot\}$.

Bez obzira što logičke promenljive mogu imati samo dve vrednosti, u memoriji se za ove podatke kod velike većine kompajlera izdvaja minimalna adresibilna jedinica, što je u većini slučajeva jedan bajt. Zbog ovoga razlikujemo nekoliko tipova memorijskih reprezentacija promenljivih logičkog tipa. Na primer, za netačno se može uzeti vrednost $(0)_{10} = (0000\ 0000)_2$, a za tačno vrednost $(0xFF)_{16} = (1111\ 1111)_2$.

Većina programskih jezika nudi poseban tip podataka za deklaraciju logičkih promenljivih. C nije među njima. Programski jezik C nema poseban tip podataka za logičke promenljive, već se za pamćenje logičkih promenljivih koriste celobrojni tipovi C-a.

Definicija 4.7 (Logičke konstante u C-u)**.** Konstanta *false* u programskom jeziku C je celobrojna vrednost 0. Sve celobrojne vrednosti različite od 0 su logičke vrednosti *true*.

Primer 4.11 (Ilustracija logičkih promenljivih u upravljačkoj strukturi alternacije)**.** Kako je prethodno rečeno, promenljive bilo kog celobrojnog tipa se mogu koristiti za pamćenje logičkih podataka, što ilustruje sledeći progam.

```
1  main()
2  {
3      char x = 15;
4      int a = 0;
5      if (x)
6          printf("x je tacno\n");
7      else
8          printf("x je netacno\n");
9      if (a)
10         printf("a je tacno\n");
11     else
12         printf("a je netacno\n");
13 }
```

Nakon prevođenja i izvršavanja, program će dati izlaz

```
x je tacno
a je netacno
```

△

Interesantna je kombinacija logičkih podataka i operatora za rad sa bitovima. Razmotrićemo ovo na narednom primeru.

Primer 4.12 (Kombinacija operatora za rad sa bitovima i logičkih promenljivih)**.** Neka je na paralelni port računara povezano više mehaničkih prekidača postavljenih na nekoj mašini kako bi davali status položaja pojedinih delova mašine zbog upravljanja pomoću računara.

Postoje funkcije u C-u kojima je moguće očitati status porta i vrednost upisati u promenljivu. Nakon čitanja porta, u promenljivoj svaki bit odgovara po jednom pinu paralelnog porta. Pretpostavimo da je naponski nivo na pinu porta 5V, ako je prekidač pritisnut, i 0V ako nije[5], što nakon očitavanja porta odgovara logičkim vrednostima bitova 1 i 0, respektivno.

Pretpostavimo da imamo 8 prekidača i da im je trenutni položaj takav da na portu daju vrednost $(0011\ 1000)_2 = (0x38)_{16}$. Bez ulaženja u detalje kako izgledaju funkcije za očitavanje sadržaja paralelnog porta, sledeći program proverava da li su 3. i 4. taster zdesna pritisnuti.

```
1  main()
2  {
3      short port = 0x38; // pojednostavljeno, umesto ucitavanja vrednosti
4      short P3 = port & 0x04; // binarno je 0000 0100
5      short P4 = port & 0x08; // binarno je 0000 1000
6      if (P3)
7          printf("treci pritisnut");
8      if (P4)
9          printf("cetvrti pritisnut");
10 }
```

U 4. liniji izvršena je binarna opercija "I" nad odgovarajućim bitovima promenljive *port* i konstante $(0x04)_{16} = (0000\ 0100)_2$. Binarna reprezentacija konstante 0x04 sadrži samo jednu jedinicu i to na trećoj poziciji sa desne strane. Po logičkoj funkciji "I", rezultat će biti 1 samo ako su oba operanda 1, pa tako ova konstanta anulira vrednosti svih ostalih bitova, sem bita koji odgovara prekidaču na trećoj poziciji, bez obzira koju vrednost imaju. Ako je vrednost 3. bita promenljive *port* jednaka 1, rezultat će na trećoj binarnoj poziciji imati jedinicu (dekadna vrednost 4), a ako je vrednost 0, rezultat će na svim pozicijama imati vrednost 0.

[5]Logika može biti i obrnuta, što zavisi od toga kako su povezani port, prekidači, napajanje i masa.

U prvom slučaju, ako celobrojni rezultat $(0000\,0100)_2 = (4)_{10}$ posmatramo kao logičku vrednost u uslovu alternacije iz 6. linije, vrednost je tačna, a u drugom slučaju vrednost je netačna (definicija 4.7).

Slična je situacija i sa izrazom u 5. liniji, odnosno alternacijom u 8. liniji.

Konstante koje su korišćene u 4. i 5. linije uobičajeno se nazivaju **maske** za maskiranje bitova. △

Logički i relacioni operatori

Definicija 4.8 (Logički operatori)**.** Logički operatori su operatori koji izvršavaju logičke operacije "I", "ILI", "NE", i sl. nad operandima koji su logičkog tipa i kao rezultat daju vrednost iz skupa $\{\top, \bot\}$.

Po broju operanada logički operatori mogu biti unarni i binarni. Unarni logički operator u C-u je

- **!** – logičko "NE",

i piše se sa leve strane promenljive, dok su binarni operatori

- **&&** – logičko "I", i
- || – logičko "ILI".

Primer 4.13 (Ilustracija logičkih operatora)**.** Sledeći progam ilustruje upotrebu logičkih operatora.

```
1  main()
2  {
3      short x, y, z;
4      x = 0; y = 1; z = 5;       // x je false
5              // y je true
6              // z je true
7      short R = (!x || y) && z;
8      printf("%d\n",R);
9      if (R)
10         printf("tacno");
11     else
12         printf("netacno");
13 }
```

Nakon prevođenja i izvršavanja, program će dati izlaz

```
1
tacno
```

Vrednost logičkog izraza

$$(!x \;||\; y)\;\&\&\; z$$

u 7. liniji programa, matematički zapisanog kao

$$(\neg x \lor y) \land z,$$

određuje se na osnovu vrednosti promenljivih x, y i z. Za date vrednosti promenljivih vrednost ovog izraza je *true*.

U slučaju kada je vrednost izraza *true*, C kompajler za preračunatu vrednosti izraza uzima vrednost 1. Ukoliko je vrednost izraza *false*, kompajler će uzeti 0. Zbog ovoga program u 8. liniji ispisuje dekadnu vrednost 1, i poruku *tacno* nakon alternacije sa promenljivom R u uslovu. △

Definicija 4.9 (Relacioni operatori)**.** Relacioni operatori su binarni operatori koji za operande imaju numeričke, znakovne ili logičke promenljive i kao rezultat daju vrednost iz skupa $\{\top, \bot\}$, zavisno od relacije koje postoje između vrednosti promenljivih ("je manje", "je veće", "jednako", "nejednako", i sl.).

Relacioni operatori u C-u su

- `<` – "je manje",
- `>` – "je veće",
- `<=` – "manje ili jednako",
- `>=` – "veće ili jednako",
- `==` – "jednako", i
- `!=` – "različito".

Uzmimo primer sledećeg dela programa

```
1  ...
2  a = 10;
3  b = 9;
4  R = a != b;
5  if (R)
6    ...
```

Izraz `(a != b)` iz 4. linije sadrži relacioni operator != koji poredi da li je vrednost promenljivih a i b ista. Kako to nije slučaj, vrednost izraza je netačna, pa se promenljivoj R dodeljuje vrednost 0.

Umesto dodeljivanja vrednosti izraza promenljivoj, izraz je moguće direktno pisati u uslovu alternacije, što je ilustrovano sledećim primerom.

Primer 4.14 (Ilustracija relacionih operatora)**.** Na programskom jeziku C napisati program koji proverava da li je korisnik uneo sa tastature vrednost 10 i prikazuje odgovarajuću poruku.

```
1 main()
2 {
3     int x;
4     scanf("%d",&x);
5     if (x != 10)
6         printf("Uneta vrednost je razlicicta od 10.");
7     else
8         printf("Uneta je vrednost 10.");
9 }
```

△

4.1.5 Složeni operatori i operatori transformacije podataka

Programski jezik C ima bogat operatorski jezik. Za razliku od prethodno obrađenih operatora, među operatorima programskog jezika C nalaze se i operatori koji nisu tipični za druge programske jezike. Ti operatori su:

- složeni oblik operatora dodele vrednosti promenljivoj,
- operator grananja,
- *sizeof* operator,
- *comma* operator, i
- operator za konverziju tipa.

Operator dodele

Mada je u prethodnom tekstu više puta korišćen, operator dodele u programskom jeziku C je **binarni** operator =, koji sa svoje leve strane ima simboličko ime promenljive kojoj se dodeljuje vrednost, a sa desne strane promenljivu, konstantu, ili izraz čija se vrednost dodeljuje promenljivoj sa leve strane. Ovo je elementarni operator dodele vrednosti, koji u ovom ili sličnom obliku postoji u svim višim programskim jezicima.

Sintaksa elementarnog operatora dodele je:

$$\langle dodela_vrednosti \rangle \quad ::= \quad \langle prom. \rangle = \langle prom. \rangle | \langle izraz \rangle | \langle konstanta \rangle$$

a kompletna sintaksa operatora dodele u programskom jeziku C je:

$$\langle dodela_vrednosti \rangle \quad ::= \quad \langle \mathbf{promenljiva} \rangle\ [\langle \mathbf{op} \rangle] = \langle \mathbf{promenljiva} \rangle \mid \langle \mathbf{izraz} \rangle \mid \langle \mathbf{konstanta} \rangle$$

$$\langle op \rangle \quad ::= \quad + \mid - \mid * \mid / \mid \% \mid << \mid >> \mid \& \mid | \mid \hat{}$$

Kao što se iz prethodnog opisa sintakse može videti, dodela vrednosti u C-u se može izvršiti pomoću elementarnog operatora dodele vrednosti '=', ili pomoću jednog od dvokarakternih složenih operatora

$$+=, -=, *=, /=, \%=, <<=, >>=, \&=, |=, \hat{}=$$

s obzirom na to da se opciono sa leve strane simbola '=' može dopisati neki od navedenih operatora.

Dvokarakterni operator dodele je operator pomoću koga je izraz tipa

$$\langle promenljiva_1\rangle = \langle promenljiva_1\rangle\langle operacija\rangle\langle promenljiva_2\rangle$$

moguće kraće zapisati kao

$$\langle promenljiva_1\rangle\langle operacija\rangle = \langle promenljiva_2\rangle$$

Na primer, izraz

```
x = x + 1;
```

je u C-u moguće kraće zapisati kao

```
x += 1;
```

Isto važi i za ostale operatore navedene u prethodnoj definiciji sintakse složenog operatora dodele vrednosti.

Operator grananja

Programski jezik C jedan je od retkih jezika koji u svojoj sintaksi bogatoj operatorima ima i operator grananja. Uloga operatora grananja je identična ulozi upravljačke strukture alternacije *if-then-else*: zavisno od uslova izvršiće se jedan ili drugi deo koda. Razlika je u tome što je ovaj operator moguće napisati u okviru izraza, pa je sam programski kod kraći i kompaktniji.

Operator grananja je **ternarni operator**, koga čine dva simbola (`?` i `:`) i koji ima tri operanda.

Sintaksa ternarnog operatora grananja je:

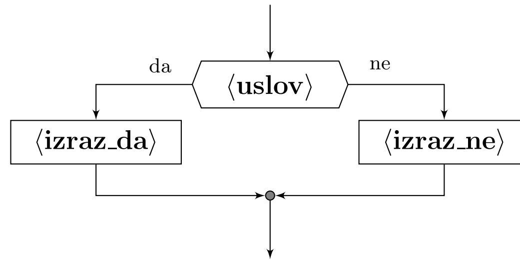
$$\langle operator_grananja\rangle ::= \langle \textbf{uslov}\rangle \; ? \; \langle \textbf{izraz_da}\rangle \; : \; \langle \textbf{izraz_ne}\rangle$$

$$\begin{aligned}
\langle uslov\rangle &::= \langle izraz\rangle|\langle logicka_prom,\rangle|\langle konstanta\rangle \\
\langle izraz_da\rangle &::= \langle izraz\rangle|\langle prom.\rangle|\langle konstanta\rangle \\
\langle izraz_ne\rangle &::= \langle izraz\rangle|\langle prom.\rangle|\langle konstanta\rangle
\end{aligned}$$

Ovaj operator radi tako što na početku ispita uslov $\langle uslov\rangle$, pa ukoliko je uslov tačan, računa vrednost izraza $\langle izraz_da\rangle$, zanemarujući u potpunosti $\langle izraz_ne\rangle$, i

obrnuto, ukoliko uslov nije tačan. Iza uslova nalazi se prvi simbol operatora `?`, i ovaj simbol razdvaja uslov od izraza. Izrazi ⟨*izraz_da*⟩ i ⟨*izraz_ne*⟩ su međusobno razdvojeni drugim simbolom ovog operatora `:`. Po sintaksi C-a, ⟨*uslov*⟩, kao i ⟨*izraz_da*⟩ i ⟨*izraz_ne*⟩, mogu biti izrazi, promenljive ili konstante.

Dijagramom toka algoritma koji implementira operator grananja može se predstaviti na način prikazan na slici 4.4.



Slika 4.4: Dijagramom toka algoritma operatora grananja

Sledeći primer ilustruje upotrebu operatora grananja.

Primer 4.15 (Određivanje maksimuma dva broja operatorom grananja)**. Zadatak**: Napisati program koji korišćenjem operatora grananja određuje veći od dva cela broja koje zadaje korisnik.

Rešenje:

```
1 main()
2 {
3     int R, a, b;
4     scanf("%d%d",&a, &b);
5     R = a>b?a:b;
6 }
```

Nakon unosa vrednosti u promenljive a i b, u 5. liniji koda nalazi se izraz sa operatorom grananja. Na početku operator grananja ispituje uslov `a>b`, pa ako je uslov zadovoljen ceo operatorski izraz postaje samo izraz u "da" grani operatora grananja, t.j. 5. linija koda postaje `R=a`. U suprotnom u 5. liniji koda imamo `R=b`.

Ekvivalentan kod bez upotrebe operatora granja je

```
1 main()
2 {
3     int R, a, b;
```

```
4     scanf("%d%d",&a, &b);
5     if (a > b)
6         R = a;
7     else
8         R = b;
9 }
```

△

Bilo koju algoritamsku strukturu moguće je implementirati upravljačkom strukturom alternacije *if-then* umesto operatorom grananja, tako da sem kraćeg i kompaktnijeg zapisa, ne postoji direktna potreba za ovim operatorom. Zbog toga mnogi drugi jezici u svojoj sintaksi i nemaju ovaj operator. Prednost ovog operatora je, s jedne strane, kraći i kompaktniji zapis, ali je s druge strane mana što je takav kod teže čitljiv i teže ga je analizirati.

Primer 4.16 (Ilustracija upotrebe operatora grananja u složenom izrazu)**.** Operator grananja u sledećem programu primenjen je u složenom izrazu. U okviru jednog izraza u jednoj liniji koda izračunat je zbir apsolutne vrednosti promenljive x, minimuma promenljivih a i b i maksimalne vrednosti od unetih vrednosti za m i n.

```
1  main()
2  {
3      int R, x, a, b, m, n;
4      printf("Unesite␣vrednosti␣za␣x,␣a,␣b,␣m␣i␣n:␣");
5      scanf("%d%d%d%d%d",&x, &a, &b, &m, &n);
6
7      R = ((x>0)?x:-x)+((a<b)?a:b)+((m>n)?m:n);
8
9      printf("Rezultat␣je␣%d",R);
10 }
```

Izgled konzole nakon prevođenja i izvršenja ovog programa, ako se redom za promenljive unesu vrednosti -5, 3, 2, 8 i 10 je

```
Unesite vrednosti za x, a, b, m i n: -5 3 2 8 10
Rezultat je 17
```

Upotrebom makroa ovaj program se može učiniti čitljivijim. Ako definišemo makroe kao:

```
1 #define abs(a) a>0?a:-a
2 #define min(x,y) x<y?x:y
3 #define max(x,y) x>y?x:y
```

gde prvi makro služi za određivanje apsolutne vrednost zadatog broja, a druga dva za određivanje minimalne i maksimalne vrednosti zadatih brojeva, tada se izraz iz 6. linije programa može napisati kao:

```
R = abs(a) + min(a,b) + max(m,n);
```

△

Operator *sizeof*

U programskom jeziku C *sizeof* je unarni operator koji određuje i daje veličinu memorijskog prostora u bajtovima koju zauzima operand. Bez obzira što je neuobičajeno koristiti cele reči za operatore jezika, operator *sizeof* je potpuno ravnopravan sa unarnim operatorima + ili -, piše se sa leve trane promenljive, i nalazi se među ključnim rečima C-a.

Sintaksa *sizeof* operatora je

$$\langle sizeof_operator \rangle \quad ::= \quad \textbf{sizeof}\ \langle promenljiva \rangle \mid (\langle tip \rangle)$$

Ukoliko se kao operand *sizeof* operatora nalazi naziv tipa, tada se naziv tipa piše u okviru malih zagrada.

Primer 4.17 (Određivanje veličine promenljive i tipa *sizeof* operatorom)**.** Sledeći program određuje i prikazuje veličinu memorijskog prostora u bajtovima koji zauzima promenljiva *uc* deklarisana kao karakter promenljiva, kao i veličinu koje u memoriji zauzimaju promenljive tipa *int*.

```
1  main()
2  {
3      unsigned char L, I;
4      char uc;
5
6      L = sizeof uc;
7      I = sizeof (int);
8
9      printf("Velicina promenljive uc je %dB.\n",L);
10     printf("Tip int zauzima %d bajta.\n",I);
11 }
```

Program na izlazu daje

```
Velicina promenljive uc je 1B.
Tip int zauzima 4 bajta.
```

△

Operator *comma*

Operator *comma* (prev. *zarez*) je binarni operator programskog jezika C, za koji se koristi simbol '`,`'. Ovaj operator služi za razdvajanje **dva nezavisna izraza** u okviru iste linije koda, t.j. u okviru jednog izraza. Levi operand operatora *comma* je izraz, kao i desni operand. Prvo se izvršava levi izraz, a nakon toga i desni. Sintaksa ovog operatora je:

⟨*comma_operator*⟩ ::=
⟨*levi_izraz*⟩ , ⟨*desni_izraz*⟩

Sledeći kod ilustruje upotrebu *comma* operatora:

```
1 main()
2 {
3     int x, y;
4     x = 3, y = 10;
5 }
```

U 4. liniji koda u istom izrazu navedena su dva nezavisna izraza: `x=3` i `y=10`, koji se nezavisno izvršavaju jedan za drugim. Generalno, ovo je bilo moguće izvesti i pomoću dve nezavisne naredbe na sledeći način:

```
1 main()
2 {
3     int x, y;
4     x = 3;
5     y = 10;
6 }
```

S obzirom da je u C-u oznaka za kraj naredbe simbol ';', a ne prelazak u novi red, izrazi iz 4. i 5. linije prethodnog programa mogli su biti napisani u okviru jedne linije kao `x = 3; y = 10;`.

Na osnovu prethodno izloženog postavlja se pitanje praktične primene operatora *comma*, kao i same upotrebne vrednosti ovog operatora. Upotrebnu vrednost ovaj operator dobija kad god je na mestu gde se sme pisati samo jedan izraz potrebno napisati više od jednog izraza.

Na primer, po sintaksi petlja tipa *for* sadrži 3 izraza (definicija sintakse na strani 128). Nakon ključne reči *for* izrazi su razdvojeni tačkom i zarezom: izraz za inicijalizaciju brojača, uslov za kraj, i izraz za promenu vrednosti brojača. Na primer, petlja:

```
1 int i;
2 for (i = 0 ; i < 5 ; i++)
3     ...
```

uvećava vrednost brojača *i* u svakoj iteraciji za 1, počev od 0 do 4. Ukoliko u *for* petlji iskoristimo *comma* operator, umesto svakog izraza možemo napisati više izraza i time dobiti veoma složene strukture petlji. Na primer, petlja u sledećem programu ima dva brojača, iako se radi o jednoj petlji:

```
1 int i,j;
2 for (i = 0, j = 9 ; i < 5 && j > 0 ; i++, j--)
3     printf("%d %d\n", i, j);
```

Brojač *i* ima početnu vrednost 0 i u svakoj iteraciji povećava se za 1. Brojač *j* ima početnu vrednost 9 i u svakoj iteraciji se smanjuje za 1. Petlja će se završiti kada jedan od uslova `i<5` ili `j>0` nije ispunjen (relacioni operator "I" je između ovih izraza). Prethodni program će dati sledeći izlaz:

```
0 9
1 8
2 7
3 6
4 5
```

Ovo je svakako moguće implementirati i na drugi način, pomoću dve petlje, ili pronalaženjem matematičke zavisnosti izmađu brojača. Međutim, postojanje operatora *comma* i mogućnost njegove upotrebe u strukturama za kontrolu toka daje veću fleksibilnost programskom jeziku C.

Operator za konverziju tipa

Operator za konverziju tipa, ili tzv. *cast* operator (čita se *kast*), je unarni operator čija je namena eksplicitna promena tipa promenljvih. Ovaj operator stoji sa leve strane operanda i eksplicitno vrši konverziju tipa, t.j. menja mu tip u željeni tip podataka. Sintaksa ovog operatora je:

$$\langle cast \rangle \; ::= \; (\langle \textbf{tip} \rangle)\ \langle \textbf{operand} \rangle$$

$$\langle tip \rangle \; ::= \; \textbf{int} \mid \textbf{float} \mid \textbf{char} \mid \ldots$$

$$\langle operand \rangle \; ::= \; \langle prom. \rangle \mid \langle konstanta \rangle \mid \langle izraz \rangle$$

Naziv željenog tipa u koji se prevodi promenljiva, konstanta ili izraz navodi se u malim zagradama. Sam naziv tipa i zagrade čine *cast* operator, pa tako imamo `(int)`, `(float)`, `(char)` operator, itd. Ovaj operator se piše sa leve strane operanda. Sledeći kod ilustruje upotrebu *cast* operatora.

```
1 main()
2 {
3     int x = 3;
```

```
4     float y;
5     y = (float) x;
6 }
```

U prethodnom primeru kompajler bi i bez eksplicitne upotrebe *cast* operatora implicitno preveo vrednost promenljive x u *float*, pa tek tada vrednost dodelio promenljivoj y. Zbog ove osobine C jezik se i naziva jezikom sa implicitnom konverzijom tipova. Međutim, postoje slučajevi kada je upotreba eksplicitne konverzije neophodna.

Na primer, binarni operator za deljenje brojeva '/' može biti celobrojni i realni, oznaka je ista, ali tip zavisi od tipa operanada. Ako su oba operanda celobrojna i deljenje će biti celobrojno. Celobrojno deljenje automatski odseca razlomljeni deo rezultata i vraća samo celobrojni deo. Ovo je ilustrovano sledećim programom:

```
1 main()
2 {
3     int x = 5;
4     float y;
5     y = x / 2;
6     printf("%f",y);
7 }
```

Iako se na prvi pogled može očekivati da je rezultat izvršenja ovog programa $y = 2.5$, zapravo će u y biti upisana vrednost 2. Naime, deljenje u 5. liniji je celobrojno deljenje, jer su oba operanda celobrojna. Celobrojno deljenje brojeva 5 i 2 daje rezultat 2, a ne 2.5, pa se vrednost 2 upisuje u promenljivu y, bez obzira što je ova promenljiva realnog tipa.

Rešenje problema je bilo koji od operanada eksplicitno prevesti u realni tip. Time i deljenje postaje realno deljenje, koje za rezultat ima realni broj. S obzirom na to da je u ovom konkretnom slučaju jedan od operanada konstanta, jedan način je umesto celobrojne konstante pisati realnu:

```
y = x / 2.;
```

Drugi način je iskoristiti operator za ekplicitno prevođenje tipa i promenljivu x prevesti u tip *float* na sledeći način:

```
y = (float)x / 2;
```

Greške koje se mogu desiti prilikom gubitka razlomljenog dela kod celobrojnog deljenja ne spadaju u sintaksne greške, kao što je pokazano na prethodnom primeru. Nekada je i korisno imati celobrojno deljenje da bi se odredio samo celobrojni deo rezultata. Međutim, zbog moguće pojave greške većina kompajlera prilikom prevođenja programa na ovakve sutuacije ukazuje upozorenjem (eng. *warning*). Program koji sadrži upozorenja se može prevesti i izvršiti, ali kompajler skreće pažnju

programeru navođenjem linije koda na koju upozorava programera. Okruženje *Microsoft Visual Studio 2010* prilikom prevođenja gore navedenog programa, u slučaj kada su oba operanda celobrojna, javlja sledeće upozorenje za 5. liniju koda:

"Warning 1 warning C4244: '=' : conversion from 'int' to 'float', possible loss of data c:\users\vciric\documents\visual studio 2010\projects\ch4\ch4\primer.c 5"

Operatori referenciranja i dereferenciranja

Definicija 4.10 (Operator referenciranja)**.** Operator referenciranja je unarni operator koji stoji sa leve stane promenljive i vraća adresu u memoriji na kojoj se promenljiva nalazi. Simbol kojim se predstavlja operator referenciranja je '`&`'.

Sintaksa izraza za operatorom referenciranja je:

$$\langle referenciranje \rangle \quad ::= \quad \& \; \langle \textbf{promenljiva} \rangle$$

Napomena: Bez obzira što se za operaciju referenciranja i operaciju "I" nad bitovima promenljivih koristi isti simbol, o kom se od ova dva operatora radi kompajler zaključuje na osnovu toga da li su operatori primenjeni kao binarni ili kao unarni. Operator "I" je binarni operator, a operator referenciranja je unarni operator.

Primer 4.18 (Operator referenciranja)**.** Sledeći program prikazuje adrese na kojima se nalaze promenljive A i B. Kao što je i običaj za predstavljanje adresa u memoriji, adrese su prikazane kao brojevi u heksadekadnom brojnom sistemu pomoću konverzionog karaktera `%x`.

```
1 main()
2 {
3     int A = 3, B = 2;
4     printf("A je na adresi %x, a B je na lokaciji sa adresom %x u memoriji.",
      &A, &B);
5 }
```

Nakon kompajliranja i pokretanja program daje sledeći izlaz:

```
A je na adresi 30faf0, a B je na lokaciji sa adresom 30fae4
u memoriji.
```

Neophodno je napomenuti da su adrese koje prikazuje program adrese u memoriji koje su alocirane za promenljive na zahtev programa u trenutku deklaracije. Kako je memorija veoma dinamičan sistem čiji se sadržaj, zauzeti i oslobođeni prostor menja iz časa u čas, prilikom sledećeg pokretanja programa promenljivama vrlo verovatno neće biti dodeljene iste adrese, pa će program prikazati neke druge adrese na kojima se promenljive trenutno nalaze. △

Definicija 4.11 (Operator dereferenciranja)**.** Operator dereferenciranja je unarni operator koji stoji sa leve stane promenljive koja sadrži adresu i vraća sadržaj memorijske lokacije sa te adrese. Simbol kojim se predstavlja operator dereferenciranja je '*'.

Ista napomena, kao i za unarni operator referenciranja & i binarni bitski operator "I", važi i za unarni operator dereferenciranja * i binarni operator za množenje. O kom se operatoru radi, s obzirom na to da se koristi isti simbol, kompajler određuje iz konteksta izraza.

Sintaksa izraza sa operatorom referenciranja je:

$$\langle dereferenciranje \rangle \quad ::= \quad * \langle \textbf{adresa} \rangle$$

Primer 4.19 (Operator dereferenciranja)**.** Sledeći program ilustruje upotrebu operatora dereferenciranja:

```
1 main()
2 {
3     int A = 3, B;
4     B = * (&A);
5     printf("%d",B);
6 }
```

Vrednost promenljive B nakon izvršenja ovog programa je 3. △

Operatori referenciranja i dereferenciranja nalaze veliku primenu kod pokazivačkih tipova podataka, koji omogućavaju indirektno adresiranje memorije, čime se programski jezik C po karakteristikama i brzini izvršavanja programa takođe može približiti asemblerskim jezicima. Pokazivači će biti obrađeni u narednim poglavljima.

4.1.6 Prioritet operatora i prioritet tipova podataka

U složenim izrazima koji obuhvataju više operatora, s jedne strane neophodno je znati po kom prioritetu će se operatori izvršavati, ukoliko to zagradama nije eksplicitno naglašeno.

S druge strane, kao jedna od karakteristika programskog jezika C navedena je slaba tipizacija, što znači da se u okviru istog izraza mogu naći promenljive različitog tipa. Kada je ovo slučaj kompajler automatski vrši transformaciju tipova podataka, a koji će tip automatski biti preveden u koji određuje prioritet tipova podataka.

Prioritet operatora

Zbog preglednosti, svi prethodno opisani operatori navedeni su u tabeli 4.3.

Za svaki operator naveden je tip operatora, simbol koji se koristi za operator, kao i **asocijativnost**, što označava redosled izvršavanja operatora ako je više istih operatora u istom izrazu. Na primer, izraz

$$R = x + y + z + 10;$$

kompajler će izvršiti u 3 koraka (ne računajući dodelu vrednosti). S obzirom na to da je asocijativnost binarnog operatora + (tabela 4.3) s leva u desno, prvo će se izvršiti sabiranje

$$x + y.$$

Nakon toga, međurezultatu će biti dodata vrednost promenljive z, t.j.

$$(x + y) + z,$$

a na kraju će biti dodata i konstanta 10 kao:

$$((x + y) + z) + 10.$$

Prethodno opisani postupak naziva se leva asocijativnost, ili asocijativnost sleva udesno.

<table>
<tr><th>Pr.</th><th>Tip operatora</th><th>Operator</th><th>Asocijativnost</th></tr>
<tr><td>1</td><td>Primarni operatori izraza</td><td>( ) [ ] . desni++ desni--</td><td>sleva udesno</td></tr>
<tr><td>2</td><td>Unarni operatori</td><td>* & + - ! ^ ++levi --levi (cast_operator) sizeof</td><td>zdesna ulevo</td></tr>
<tr><td>3</td><td rowspan="10">Binarni operatori</td><td>* / %</td><td rowspan="10">sleva udesno</td></tr>
<tr><td>4</td><td>+ -</td></tr>
<tr><td>5</td><td>>> <<</td></tr>
<tr><td>6</td><td>< > <= >=</td></tr>
<tr><td>7</td><td>== =!</td></tr>
<tr><td>8</td><td>&</td></tr>
<tr><td>9</td><td>^</td></tr>
<tr><td>10</td><td>|</td></tr>
<tr><td>11</td><td>&&</td></tr>
<tr><td>12</td><td>||</td></tr>
<tr><td>13</td><td>Ternarni operator</td><td>? :</td><td>zdesna ulevo</td></tr>
<tr><td>14</td><td>Operatori dodele</td><td>= += -= *= ...</td><td>zdesna ulevo</td></tr>
<tr><td>15</td><td>comma operator</td><td>,</td><td>sleva udesno</td></tr>
</table>

Tabela 4.3: Skup karaktera programskog jezika C

Pored asocijativnosti u tabeli je naveden je i prioritet izvršavanja operatora. Operatori su u tabeli poređani po prioritetu počev od operatora sa najvećim prioritetom (1), do operatora sa najmanjim prioritetom (15).

Prioritet operatora određuje redosled izvršavanja operatora u izrazu, ukoliko to zagradama nije drugačije rečeno. Na primer, izraz:

$$3 * 2 + 2 * 2$$

imaće vrednost 10, zato što binarni operator za množenje ima veći prioritet u odnosu na sabiranje (tabela 4.3), pa je prethodno napisani izraz ekvivalentan izrazu napisanom sa zagradama:

$$(3 * 2) + (2 * 2).$$

Detaljnije, s obzirom na to da je množenje `*` levo asocijativna operacija, prvo će se izvršiti množenje `3*2`, pa zatim množenje `2*2`, a na kraju će se ova dva međurezultata sabrati.

Napomena: Ne računajući operator *comma*, operatori dodele vrednosti imaju najmanji prioritet, zato se dodela vrednosti obavlja na kraju, tek nakon što se obave sve druge operacije u izrazu. Na primer, u izrazu `R = x + y * z;` će se prvo po prioritetu izvršiti množenje, zatim sabiranje, a na kraju će se izračunata vrednost dodeliti promenljivoj R.

Interesantno je pomenuti da se u izrazu u C-u može naći više operatora dodele. S obzirom na desnu asocijativnost, sledeći izraz je sintaksno ispravan:

```
x = y = z = 10;
```

U ovom izrazu će se prvo izvršiti prva dodela vrednosti zdesna, t.j. $z = 10$. Nakon toga će vrednost promenljive z, koja je sada jednaka 10, biti dodeljena promenljivoj y, itd. Sledeći izraz, međutim, nije sintaksno ispravan, jer je druga dodela po redu dodela vrednosti izrazu, a ne promenljivoj, što je neispravno.

```
x = y + 1 = z = 10;
```

Prioritet tipova podataka

Prioritet tipova podataka razmotrićemo na primeru operacije deljenja (`/`), jer se kod ove operacije rezultat može razlikovati, ukoliko se radi o celobrojnom ili realnom deljenju. Uzmimo sledeći primer u razmatranje:

```
int a=5, b=2;
float c=3, R;
R = a / b * c;
```

U izrazu `R = a / b * c;` imamo tri binarna operatora: operator deljenja, množenja i dodelu vrednosti. Pošto je prioritet deljenja i množenja veći od prioriteta dodele vrednosti (tabela 4.3), a deljenje i množenje imaju isti prioritet i levu asocijativnost, prvo se izvršava operacija deljenja, pa zatim operacija množenja, i na kraju dodela vrednosti.

Oba operanda operatora deljenja su celobrojna, pa kompajler uključuje celobrojnu aritmetiku za izvršavanje ove operacije[6]. Celobrojno deljenje daje celobrojni rezultat. U ovom slučaju može doći do gubitka razlomljenog dela, o čemu je bilo reči ranije.

Nakon deljenja izvršiće se množenje. Operacija množenja ima jedan celobrojni operand sa leve strane, koji je dobijen kao rezultat deljenja, i jedan realni operand, promenljivu c, sa desne strane.

Napomena: Kada se operandi neke operacije razlikuju po tipu, kompajler automatski prevodi operande svodeći ih na isti tip, po prioritetu tipova. Prevođenje tipova se vrši za svaki operator pojedinačno, neposredno pre izvršenja operatora. Kompajler uzima operand sa tipom koji ima veći prioritet i drugi operand prevodi u taj tip.

Tipovi poređani po prioritetu, počev od tipa sa najnižim prioritetom, su:

char < **short** < **int** < **long** < **float** < **double.**

Po ovome, kako operacija množenja iz primera ima jedan operand tipa *float*, a drugi operand tipa *int*, operand tipa *int* će biti automatski transformisan u *float*, pa će se izvršiti množenje dva realna broja. Potrebno je napomenuti da je prilikom celobrojnog deljenja već izgubljen razlomljeni deo rezultata deljenja i da se on ovom transformacijom neće vratiti. Rezultat deljenja je $5/2 = 2$, a konverzijom će celobrojna vrednost 2 biti prevedena u realnu vrednost 2.000, tako da će rezultat množenja, odnosno rezultat izraza biti 6.000.

Sledeća operacija koju je potrebno izvršiti je dodela vrednosti. Operator dodele vrednosti ima dva operanda, ali je ovaj operator izuzetak iz prioriteta tipova, jer u slučaju da se razlikuju tipovi uvek se tip desnog operanda svodi na tip levog operanda, kome se i dodeljuje vrednost.

4.2 Izvedeni tipovi podataka

Bogatstvo jezika po pogledu tipova podataka se može proširiti izvođenjem novih tipova podataka iz osnovnih tipova. U C-u postoje sledeći izvedeni tipovi:

1. pokazivači,
2. strukture, i
3. unije.

Pokazivači su promenljive čiji se tip izvodi iz osnovnih tipova. Koriste se za pamćenje memorijskih adresa i služe za indirektno adresiranje memorije, odnosno pristup podacima kada je poznata memorijska adresa na kojoj se podaci nalaze.

Strukture podataka u C-u su korisnički definisani tipovi, koji se koriste kada skup osnovnih tipova nije dovoljan, već postoji potreba za nekim dodatnim, složenim

[6] Procesori obično imaju nezavisne hardverske jedinice za celobrojnu i realnu aritmetiku. Celobrojna aritmetika se uglavnom izvršava daleko brže od realne aritmetike.

tipom podataka. Na primer, za pamćenje podataka o celim brojevima koristi se osnovni tip *int*, a ukoliko je potrebno zapamtiti kompleksni broj moguće je definisati novi tip *kompleksni*, koji će biti u mogućnosti da prihvati sve potrebne podatke.

Unije su takođe korisnički definisani tipovi kao i strukture, koji za razliku od struktura imaju konceptualnu razliku u načunu skladištenja podataka u memoriji.

4.2.1 Pokazivači

U prethodnim poglavljima obrađeni su unarni operator referenciranja & i operator dereferenciranja *. Primer koji je tom prilikom naveden je:

```
1 main()
2 {
3     int A = 3, B;
4     B = * (&A);
5     printf("%d",B);
6 }
```

U ovom primeru su oba operanda namerno navedena u istom izrazu kako bi se izbeglo pamćenje adrese promenljive A. Sledeći program je sintaksno neispravan, jer C ne dozvoljava upis adrese u osnovne celobrojne promenljive. Ovo je ujedno i izuzerak u implicitnoj konverziji tipova.

```
1 main()
2 {
3     int A = 3, B;
4     long adresa;
5     adresa = &A;
6     B = * adresa;
7     printf("%d",B);
8 }
```

Za pamćenje memorijskih adresa koriste se posebni tipovi podataka koji se nazivaju pokazivači (eng. *pointeri*). Ovakav naziv su dobili zato što promenljive pokazivačkog tipa, umesto da sadrže potreban podatak, sadrže adresu na kojoj se potrebni podatak nalazi.

S obzirom na to da je neophodno znati i koja je memorijska reprezentacija korišćena da bi se podatak u nekoj memorijskoj lokaciji mogao iskoristiti, kod pokazivačkog tipa potrebno je znati na kakav podatak pokazivač pokazuje. Zbog toga razlikujemo pokazivač na tip *char*, pokazivač na tip *int*, itd.

Naziv pokazivačkog tipa formira se tako što se nazivu osnovnog tipa doda simbol * sa desne strane. Osnovni tip zapravo određuje tip vrednosti na koju pokazivač

pokazuje. U EBNF notaciji naziv pokazivačkih tipova može se predstaviti kao:

$$\langle pokazivacki_tip \rangle ::= \langle osnovni_tip \rangle *$$

$$\langle osnovni_tip \rangle ::= \textbf{char} \mid \textbf{short} \mid \textbf{int} \ldots$$

Deklaracija pokazivačke promenljive u EBNF notaciji može se predstaviti kao:

$$\langle pokazivacka_promenljiva \rangle ::= \langle osnovni_tip \rangle * \langle identifikator \rangle$$

Na primer, deklaracija pokazivača čiji je identifikato p i koji pokazuje na vrednost tipa *int* po prethodno opisanoj sintaksi bila bi:

```
int* p;
```

Umesto dodavanja simbola `*` sa desne strane tipa, ukoliko je u naredbi deklaracije potrebno deklarisati više pokazivača, svakom identifikatoru sa leve strane se može dodati simbol `*`. Na primer:

```
int *p, *q;
```

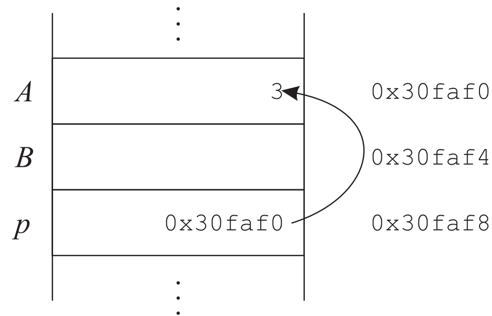
Primer 4.20 (Osnovna upotreba pokazivača)**.** Prethodno navedeni primer napisan sintaksno ispravno dat je u nastavku:

```
1 main()
2 {
3     int A = 3, B;
4     int* p;
5     p = &A;
6     B = * p;
7     printf("%d",B);
8 }
```

Promenljiva p je pokazivačkog tipa (`int*`), koja pokazuje na podatak tipa *int*. U 5. liniji koda je operatorom referenciranja uzeta adresa na kojoj se nalazi promenljiva A i upisana u pokazivač p. U 6. liniji je operatorom dereferenciranja sa adrese koja je upisana u p pročitan podatak i upisan u B, tako da je vrednost u promenljivoj B=3.

Zauzeće i sadržaj memorije za ovaj primer ilustrovani su na slici 4.5. Na slici je strelicom naglašeno na koju memorijsku lokaciju pokazuje pokazivač p. Promenljive na slici 4.5 predstavljene su redom jedna ispod druge po redosledu deklaracije u programu, pošto je ovo najverovatniji redosled prilikom zauzimanja memorije zbog redosleda deklaracija u programu. Adrese napisane sa desne strane u heksadekadnom obliku (0x30faf0, itd.) su, ilustracije radi, nasumično odabrani brojevi, ali se, svakako, međusobno razlikuju za 4 zato što su promenljive A i B tipa *int*.

Da se umesto 7. linije koda, u kojoj se prikazuje sadržaj promenljive B, nalazi



Slika 4.5: Ilustracija pokazivača

```
printf("%x %d", p, *p);
```

bila bi prikazana adresa koja se nalazi u pokazivaču p, t.j. adresa na kojoj je promenljiva A, i vrednost koja se nalazi na adresi na koju ukazuje pokazivač ($*p$), odnosno vrednost promenljive A.

△

Dereferencirani pokazivači mogu figurisati u izrazima, ravnopravno sa ostalim promenljivama, kao što je pokazano u sledećem primeru.

Primer 4.21 (Upotreba pokazivača u izrazima)**.** U sledećem primeru pokazano je kako je moguće sabrati vrednosti promenljivih A i B korišćenjem pokazivača, bez korišćenja identifikatora samih promenljivih A i B u izrazu.

```
1 main()
2 {
3     int A = 3, B = 7, R;
4     int *p,*q;
5     p = &A;
6     q = &B;
7     R = *p + *q;
8     printf("%d",R);
9 }
```

△

Prednost kod upotrebe pokazivača je mogućnost promene adrese na koju pokazuje pokazivač u toku izvršenja programa, čime je pomoću jedne promenljive moguće pristupati nizu sukcesivno smeštenih podataka u memoriji.

4.2.2 Pokazivačka algebra

Prethodno je rečeno je da dereferencirani pokazivači mogu figurisati u izrazima ravnopravno sa ostalim promenljivama. Međutim, u izrazima mogu figurisati i

nedereferencirani pokazivači, odnosno pokazivači u izvornom obliku. Suštinska razlika je da se u prvom slučaju radi sa vrednostima i kao rezultat dobijaju vrednosti, a u drugom slučaju se radi sa adresama, a kao rezultat se dobijaju nove adrese.

Moguće aritmetičke operacije nad pokazivačima su:

```
+, -, ++, --
```

kao i sve relacione operacije za poređenje sadržaja dve promenljive:

```
<, >, <= >=, ==, !=
```

Napomena: Ukoliko se vrednost pokazivača poveća za 1, pokazivaču će vrednost biti uvećana ne za 1, nego za onu vrednost koliko u bajtovima zauzima tip na koji pokazivač ukazuje. To znači da će nad pointerom `int* p;` operacija inkrementiranja `p=p+1;` ili `p++;` sadržaj lokacije `p` povećati za `sizeof(int)`.

Praktično, uvećavanjem pointera za 1 prelazi se na sledeći podatak, a za tu svrhu se pointeru dodaje onoliko bajtova koliko zauzima podatak.

Primer 4.22 (Aritmetičke operacije nad pointerima)**.** Sledeći program prikazuje vrednosti pokazivača prilikom inkrementiranja za 1.

```
1  main()
2  {
3      int A, B, C, D;
4      int* p;
5      p = &A;
6      for (int i = 1; i <= 4; i++)
7      {
8          printf("%d\n",p);
9          p = p + 1;  // ili p++, svejedno
10     }
11 }
```

Pokazivač za inicijalnu vrednost ima adresu na kojoj se nalazi promenljiva A. U *for* petlji se ova vrednost povećava za 1, što za posledicu ima uvećanje adrese za 4, ukupno 5 puta, koliko petlja ima iteracija[7]. Program će na izlazu dati vrednosti:

```
3405800
3405804
3405808
3405812
```

Naravno, da se program izvrši još jednom prikazao bi druge vrednosti, u zavisnosti koji deo operativne memorije mu je u trenutku poziva dodeljen.

[7]Program je prevođen u okruženju *Micrrosoft Visual Studio* 2010, gde tip *int* zauzima 4 bajta.

Prvoispisana adresa, 3405800, je adresa promenljive A. Neophodno je napomenuti da ostale adrese koje su prikazane mogu, ali i ne moraju odgovarati promenljivama B, C i D.

U C-u postoji način za alokaciju sukcesivnih lokacija pomoću strukturnih tipova koji se nazivaju **nizovi**, odnosno **linearna polja**, o kojima će biti reči u kasnijim poglavljima.
△

Prilikom modifikacije pokazivača aritmetičkim operacijama *mora se* voditi računa da pointer pokazuje na memorijski prostor koji je u "vlasništvu" programa. Kompajler u toku prevođenja programa ovo ne proverava, pa je moguće da prilikom izvršenja programa program dođe u situaciju da pokuša pristup memorijskom prostoru koji nije njegov. U tom slučaju operativni sistem nasilno prekida izvršenje programa i u slučaju *Windows*-a 7 prikazuje dijalog sa porukom:

"ime_programa.exe *has stopped working.*
A problem caused the program to stop working correctly. Windows will close the program and notify you if a solution is available."

Ovo je bezbednosni mehanizam *muliti-tasking*[8] operativnih sistema da zaštiti programe prilikom izvršenja od međusobnog pristupanja memorijskim lokacijama. Jedan program ne može čitati iz memorijskog prostora, niti može pisati u memorijski prostor drugog programa.

Primer 4.23 (Primer pogrešnog adresiranja memorije)**.** Sledeći program pokušava pristup memorijskoj lokaciji sa adresom 0 (`p=0`), radi upisa konstante u nju (`*p=100`). Kako je izvesno da ova lokacija nije u vlasništvu programa[9], program će prouzrokovati gore navedenu grešku i operativni sistem će nasilno prekinuti izvršenje programa.

```
1 main()
2 {
3     int A, B, C, D;
4     int* p;
5     p = 0;
6     *p = 100;
7 }
```

△

Sledeći primer ilustruje relacione operacije nad pointerima.

[8] *Muliti-tasking* je mogućnost operativnog sistema da izvršava istovremeno više od jednog programa tako što deo vremena daje procesor jednom programu, pa deo vremena sledećem, i tako redom.

[9] Početak memorijskog prostora operativni sistem koristi za sistemske podatke od vitalnog značaja za rad računara. Kod nekih sistema podaci iz BIOS-a računara se mapiraju na početak memorijskog prostora.

Primer 4.24 (Relacione operacije nad pointerima)**.** Sledeći program korišćenjem relacionih operatora nad pokazivačima proverava da li su promenljive A i B na susednim lokacijama u memoriji.

```
1  main()
2  {
3      int A, B;
4      int *p,*q;
5      p = &A; q = &B;
6      if (p+1 == q)
7          printf("jesu susedi");
8      else
9          printf("nisu");
10 }
```

△

4.2.3 Strukture podataka

Struktura u programskom jeziku C je mehanizam jezika pomoću koga korisnik može kreirati novi, kompleksni tip podataka, koji se naziva **strukturni tip**[10].

Strukturni tip je izvedeni tip. Njime se definišu složene promenljive, koje same mogu da sadrže više od jednog podatka. Obično se strukture definišu nad podacima koji su na neki način u semantičkoj vezi.

Strukturama se grupišu semantički povezani podaci. Na primer, realni tip *float* se koristi za pamćenje jednog realnog broja. Međutim, ukoliko je potrebno zapamtiti koordinate (x, y) neke tačke u 2D prostoru, potrebno je zapamtiti 2 realna broja. Ovo je, naravno, moguće uraditi tako što će se deklarisati dve potpuno nezavisne promenljive, npr. x i y. Drugi način za pamćenje koordinata tačaka je definisanje novog, strukturnog tipa podataka, npr. *tacka*, koji će se sastojati od dva realna broja, x i y. Još jedan primer mogu biti kompleksni brojevi sa svojim realnim i imaginarnim delom, itd.

Uvođenjem struktura umnogome se olakšava manipulacija sa kompleksnim podacima, a sam kod postaje daleko čitljiviji.

Definicija 4.12 (Definicija strukture)**.** Definicija strukture je deo koda programa koji opisuje strukturu.

Definicijom strukture se ne dobijaju konkretne promenljive, već se definiše sam izgled i sadržaj strukture.

Definicija 4.13 (Deklaracija strukture)**.** Deklaracija strukture, odnosno deklaracija promenljivih strukturnog tipa je kreiranje promenljivih strukturnog tipa i alokacija prostora u memoriji za njih.

[10] U programskom jeziku Pascal struktura se naziva *record*.

Sintaksa definicije strukture je sledeća:

$$\langle def_strukt._tipa \rangle ::= $$

$$\textbf{struct } [\langle naziv_strukture \rangle]$$

$$\{$$

$$\langle tip \rangle \ \langle promenljiva_1 \rangle;$$

$$\langle tip \rangle \ \langle promenljiva_2 \rangle;$$

$$\ldots$$

$$\langle tip \rangle \ \langle promenljiva_N \rangle;$$

$$\} \ [\langle identifikator_1 \rangle, \langle identifikator_2 \rangle, \ldots] \ ;$$

$$\langle naziv_strukture \rangle ::= \langle identifikator \rangle$$

$$\langle tip \rangle ::= \textbf{char} \mid \textbf{int} \mid \ldots \mid \langle def_strukt._tipa \rangle$$

Drugim rečima, nakon ključne reči *struct*, navodi se identifikator koji označava ime novog tipa, t.j. ime strukture, a za njim se u okviru vitičastih zagrada navode svi podaci koji su deo strukturnog tipa. Za svaki od ovih podataka navodi se tip i identifikator, odnosno naziv podatka, t.j. promenljive. Ovi podaci se nazivaju i **elementi strukture**.

Na kraju, nakon zatvorene vitičaste zagrade, opciono se može napisati proizvoljan broj identifikatora promenljivih ($\langle identifikator_1 \rangle, \langle identifikator_2 \rangle, \ldots$) i na ovaj način odmah prilikom definicije strukturnog tipa alocirati potreban broj promenljivih. Ukoliko se promenljive alociraju prilikom definicije strukturnog tipa, naziv strukture je moguće izostaviti, pa je zbog toga i $\langle naziv_strukture \rangle$ u sintaksi naveden kao opcioni argument, u okviru uglastih zagrada.

Primer 4.25 (Primer definicije strukturnog tipa *tacka*)**.** Na osnovu EBNF definicije strukture, novi, strukturni tip podataka koji može pamtiti podatke o tački u dvodimenzionalnom prostoru može se definisati na sledeći način:

```
1 struct tacka
2 {
3     float x;
4     float y;
5 };
```

Elementi ove strukture su realne promenljive x i y.
△

Definicija strukture se u progamu može naći ili između preprocesorskih direktiva (`#include...`) i početka programa (`main()`), ili u okviru glavnog programa. U zavisnosti od položaja u programu gde je definicija strukture napisana, razlikuje se mogućnost korišćenja strukture. Ako se definicija strukture nalazi na samom početku programa, iza preprocesorskih direktiva i ispred programa, definiciju strukture je moguće koristiti iz glavnog programa, kao i iz svih drugih delova programa

(kao što su korisničke funkcije, o kojima će biti reči kasnije). Ukoliko se definicija strukture nalazi u glavnom programu, ili u nekoj funkciji, definiciju je moguće koristiti samo iz glavnog programa, odnosno funkcije u kojoj se definicija strukture nalazi.

Deklaracija promenljivih strukturnog tipa u programskom jeiku C razlikuje se kod ANSI C kompajlera, i kod C99 i novijih kompajlera. EBNF notacija deklaracije promenljive strukturnog tipa u ANSI C-u je:

$$\langle dekl._struct_prom. \rangle ::= \\ \textbf{struct} \quad \langle naziv_strukture \rangle \quad \langle prom. \rangle \, \{ , \langle prom. \rangle \} \, ;$$

EBNF notacija deklaracije promenljive strukturnog tipa kod C99 i novijih kompajlera je:

$$\langle dekl._struct_prom. \rangle ::= \\ [\textbf{struct}] \quad \langle naziv_strukture \rangle \quad \langle prom. \rangle \, \{ , \langle prom. \rangle \} \, ;$$

Kao što se iz prethodnog izlaganja može videti, razlika u deklaraciji promenljive strukturnog tipa kod ANSI C kompajlera i novijih kompajlera je u tome što se kod ANSI C kompajlera ključna reč *struct* mora navesti pre naziva strukture, a kod C99 i novijih kompajlera je ova ključna reč opciona i može se izostaviti. U oba slučaja, nakon naziva strukture, navodi se identifikator promenljive koja se deklariše, a za tim identifikatorom može se navesti proizvoljan broj dodatnih identifikatora odvojenih zarezom (vitičaste zagrade u EBNF notaciji). Ovo je ekvivalentno deklaraciji bilo koje promenljive osnovnog tipa u C-u.

Primer 4.26 (Deklaracija promenljivih strukturnog tipa)**.** Sledeći primer ilustruje način deklaracije promenljivih strukturnog tipa *tacka*, čija je definicija data u primeru 4.25.

```
1  #include "stdio.h"
2  struct tacka
3  {
4      float x;
5      float y;
6  };
7  main()
8  {
9      struct tacka T1,T2,T3,T4;
10
11     unsigned char D;
12     D = sizeof T1;
13
14     printf("Zauzece memorije u bajtovima je %d",D);
15 }
```

Definicija strukture napisana je između preprocesorske direktive *#include"stdio.h"* i početka glavnog programa `main()`. Deklaracija 4 promenljive tipa *tacka*, T1, T2, T3 i T4, nalazi se u 9. liniji. Da svaka od ovih promenljivih sadrži po 2 *float* podatka, koji u okruženju *Microsoft Visual Studio* zauzimaju po 4 bajta, može se dokazati upotrebom operatora *sizeof*. Vrednost koju ovaj program prikazuje na izlazu je 8.
△

Jednom kada je promenljiva strukturnog tipa deklarisana, može se koristiti. Struktura se koristi tako što se preko identifikatora imena promenljive strukturnog tipa pristupa željenim elementima strukture.

Operator za pristup elementima strukture je binarni operator `'.'` (tačka). Levi operand ovog operatora je identifikator promenljive strukturnog tipa, a desni operand je identifikator elementa strukture kome se pristupa. U EBNF notaciji ovo se može izraziti na sledeći način:

$$\langle pristup_elementima \rangle \quad ::= \quad \langle naziv_promenljive \rangle . \langle element \rangle$$

Primer 4.27 (Pristup elementima strukture)**.** Sledeći primer ilustruje način pristupa elementima promenljive strukturnog tipa *tacka*, deklarisanih u primeru 4.26. Definicija strukture je izostavljena iz primera.

```
1     ...
2     struct tacka T1,T2,T3,T4;
3
4     T1.x = 2.3;
5     T1.y = 4.0;
6
7     printf("Koordinate tacke T1 su (%4.1f,%4.1f).",T1.x, T1.y);
8     ...
```

U ovom primeru elementu x strukturne promenljive $T1$ dodeljuje se realna konstanta 2.3, a elementu y konstanta 4.0. Program daje sledeći izlaz:

```
Koordinate tacke T1 su ( 2.3, 4.0).
```

△

Ukoliko se elementima strukture pristupa preko pokazivača na strukturu, pokazivač je potrebno dereferencirati, pa zatim operatorom `.` pristupiti željenom elementu. Ovo je ilustrovano sledećim delom koda.

```
1  main()
2  {
3     struct tacka T1, *p;
4
5     p = &T1;
```

```
6
7     (*p).x = 2.3;
8     (*p).y = 4.0;
9 }
```

Kako je pristup strukturama dosta čest[11], da bi se izbeglo pisanje operatora dereferenciranja, operatora *tačka* i malih zagrada, uveden je novi binarni operator `->`. Sintaksa pristupa elementima strukture operatorom `->` je:

$$\langle pristup_elementima\rangle \quad ::= \quad \langle pokazivac_na_prom.\rangle \; - > \; \langle element\rangle$$

Prethodni primer se upotrebom operatora '$->$' elegantnije može napisati na sledeći način:

```
1 main()
2 {
3     struct tacka T1, *p;
4
5     p = &T1;
6
7     p->x = 2.3;
8     p->y = 4.0;
9 }
```

4.2.4 Koncept objektno-orjentisanog programiranja

Objektno-orjentisani programski jezik C++ (pa i C#, i dr.) je nadogradnja proceduralnog programskog jezika C. Nakon detaljno izloženih struktura u C-u, poželjno je istaći vezu između C-a i C++-a, koja se, najjednostavnije rečeno, tehnički svodi na razliku između struktura podataka i klasa.

Naime, **klasa** u C++ je sintaksno veoma slična strukturi u C-u, uz dve bitne sintaksne razlike:

1. klasa umesto ključne reči *struct* ima ključnu reč **class**, i
2. klasa za elemente, pored promenljivih, može imati i funkcije, tzv. metode klase, koje mogu izvršavati operacije i modifikovati vrednosti promenljivih.

U objektnom programiranju, može se reći da klasa predstavlja tip, kao i struktura u C-u, a **objekat** "promenljivu" deklarisana iz klase.

Pravi značaj objektnom programiranju daju gotovi skupovi klasa koji se distribuiraju uglavnom uz razvojna okruženja i nazivaju se **framework**. U literaturi se *framework* često prevodi kao "okvir".

[11] Koncept struktura se u objektno-orjentisanom programiranju nadgrađuje i strukture se proširuju u klase, koje čine osnovu objektnog programiranja.

Trenutno najpoznatiji *framework* je *Microsoft*-ov .NET framework (čita se "dot NET"). U ovom okviru postoji skup različitih vrsta klasa za gotovo sve funkcije računara. Jedna od klasa koje dolaze uz ovaj okvir je i klasa za prozor u *Windows* operativnom sistemu. Na programeru je u ovom slučaju da kreira promenljivu klase, t.j. objekat, pristupi elementima (npr. pozicija prozora na ekranu, veličina, tip prozora i sl.) i pozove metode objekta za iscrtavanje prozora. Time će se na ekranu prikazati prozor sa željenim karakteristikama.

Ovde ćemo se zadržati na prethodnom konceptualnom objašnjenu i kratkom primeru klase koja opisuje tačku sa koordinatama x i y, sa metodom za prikaz elemenata.

```
1  #include "stdio.h"
2  class tacka
3  {
4  public:
5      float x;
6      float y;
7      void prikazi() // metod (funkcija) za prikaz
8      {
9          printf("(%4.1f,%4.1f)",x,y);
10     }
11 };
12 main()
13 {
14     class tacka T1, T2;
15     T1.x = 2.3; T1.y = 4.0;
16     T2.x = 1.2; T2.y = 5.0;
17
18     T1.prikazi();
19     T2.prikazi();
20 }
```

Ovaj program na izlazu daje:

```
( 2.3, 4.0)( 1.2, 5.0)
```

4.2.5 Ugnježdene i samoreferencirajuće strukture podataka

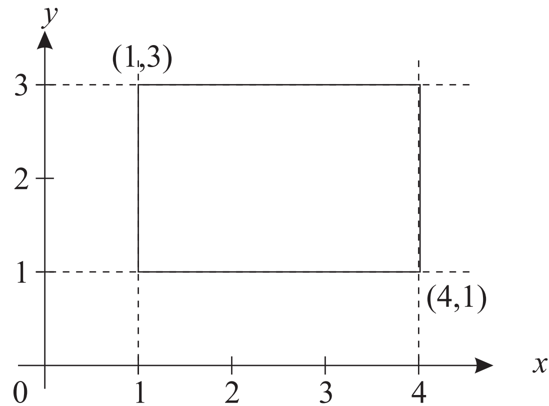
Kako je navedeno u definiciji sintakse strukture (strana 205), tip promenljivih koje čine strukturu može biti bilo koji osnovni tip podataka, uključujući i izvedene tipove pokazivače, ili drugi strukturni tip.

Definicija 4.14. Za strukturu koja se nalazi među elementima druge strukture kaže se da je ugnježdena struktura.

Ugnježdene strukture razmotrićemo na primeru strukture za opis pravougaonika.

Primer 4.28 (Ugnježdene strukture za opis pravougaonika)**.** Za pamćenje pozicije i veličine geometrijskog oblika pravougaonika, ukoliko su stranice pravougaonika

paralelne koordinatnim osama, dovoljno je i uobičajeno pamtiti samo koordinate gornje leve i donje desne tačke. Ovo je ilustrovano na slici 4.6. Jedna mogućnost



Slika 4.6: Podaci dovoljni za pamćenje pozicije i veličine pravougaonika u Dekartovom koordinatnom sistemu

za opis ove strukture je definisanje strukture sa 4 promenljive, na primer: *x_goreL*, *y_goreL*, *x_deleD*, *y_doleD*. Druga mogućnost je iskoristiti strukturu *tacka*, koju smo prethodno opisali, i definisati novu strukturu *pravougaonik*, koja će imati dva elementa tipa *tacka*. Ovaj način ilustrovan je sledećim kodom.

```
1  #include "stdio.h"
2  struct tacka
3  {
4      float x;
5      float y;
6  };
7  struct pravougaonik
8  {
9      struct tacka gl;
10     struct tacka dd;
11 };
12 main()
13 {
14     pravougaonik P1, P2;
15     P1.gl.x = 1;
16     P1.gl.y = 3;
17     P1.dd.x = 4;
18     P1.dd.y = 1;
19 }
```

Struktura pravougaonik definisana je od 7. do 11. linije koda, a sastoji se od dva elementa strukturnog tipa *tacka*: gl (9. linija koda) za gore levo, i dd (10. linija koda) za dole desno.

Deklarisane su dve promenljive tipa *pravougaonik*, P1 i P2, i odgovarajućim elementima dodeljene su vrednosti sa slike 4.6. △

Za pristup elementima ugnježdenih struktura, kao što se iz primera 4.28 vidi, takođe se koristi binarni operator '`.`', dok se operator `->` koristi kada se radi sa pokazivačima na strukture. Najpre je potrebno doći do elementa na prvom nivou strukturne promenljive tipa *pravougaonik*, npr. P1 iz primera 4.28, na sledeći način:

```
P1.gl
```

Ovako naveden operator '`.`' daje element promenljive P1 koji je takođe strukturnog tipa, pa je njegovim elementima dalje moguće pristupiti istim operatorom:

```
(P1.gl).x
```

Kako je operator '`.`' levo asocijativan (tabela 4.3, prva vrsta), male zagrade nisu neophodne, jer će se zbog leve asocijativnosti svakako prvo izvršiti levi operator (`P1.gl`), pa za njim i `(P1.gl).x`. Dovoljno je navesti:

```
P1.gl.x
```

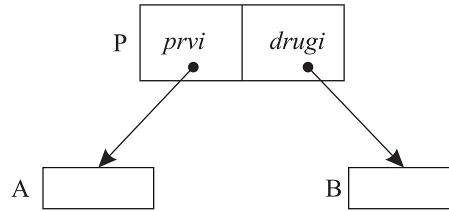
kao što je to urađeno u primeru 4.28.

Pored osnovnih i strukturnih tipova podataka, elementi strukture mogu biti i pokazivači. Sledeći primer ilustruje strukturu koja za elemente ima dva pokazivača na cele brojeve.

```
1  #include "stdio.h"
2  struct pokazivaci
3  {
4      int* prvi;
5      int* drugi;
6  };
7  main()
8  {
9      int A, B;
10     struct pokazivaci P;
11     P.prvi = &A;
12     P.drugi= &B;
13 }
```

Promenljiva P je strukturnog tipa, sa dva elementa kojima su dodeljene adrese nezavisnih promenljivih A i B, tako da je u memoriji formirana situacija prikazana na slici 4.7. Potrebno je napomenuti da se u ovom slučaju sami celobrojni podaci ne pamte u strukturi, već u posebnim promenljivama, a da im se preko strukture može pristupiti dereferenciranjem pokazivackog elementa `*(P.prvi)` i `*(P.drugi)`.

Interesantna mogućnost je da strukturni tip može biti definisan tako da se među elementima strukture nalazi pokazivač na elemente iste te strukture.

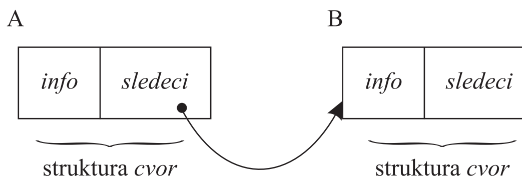


Slika 4.7: Struktura sa dva pokazivača na cele brojeve

Definicija 4.15 (Samoreferencirajuća struktura)**.** Samoreferencirajuća struktura je struktura koja među elementima ima pokazivače na istu tu strukturu.

Karakteristika samoreferencirajućih struktura je ta da se one mogu ulančavati, t.j. povezivati tako da svaka strukturna promenljiva ima informaciju o tome koja je promenljiva sledeća u listi. Sledeći primer ilustruje samoreferencirajuće strukture.

Primer 4.29 (Ulančavanje samoreferencirajućih struktura)**.** Ukoliko želimo kreirati skup od 2 podatka, ali tako da prvi podatak ima informaciju o lokaciji drugog, t.j. narednog podatka[12], dobićemo strukturu podataka prikazanu na slici 4.8. Ovu



Slika 4.8: Ulančane samoreferencirajuće strukture

strukturu možemo implementirati na način prikazan u sledećem programu.

```
1  #include "stdio.h"
2  struct cvor
3  {
4      int info;
5      struct cvor* sledeci;
6  };
7  main()
8  {
9      struct cvor A, B;
10     // pokazivacki deo cvora A pokazuje na naredni cvor B:
11     A.sledeci = &B;
12 }
```

[12]Da je više od dva podatka u listi, tada bi drugi imao informaciju o trećem, treći o četvrtom, itd.

Struktura *cvor* je samoreferencirajuća struktura, koja za elemente ima:

1. "deo" za pamćenje informacija tipa *int* (element *info*, 4. linija), i
2. pokazivač na sledeći čvor liste (*sledeci*, 5. linija).

Nakon deklaracije dve strukturne promenljive A i B u 9. liniji programa, jednom dodelom u 10. liniji adresa promenljive B je zapamćena u pokazivačkom elementu *sledeci* promenljive A, čime je formirana ulančana struktura sa slike 4.8. △

4.2.6 Unije

Definicija 4.16 (Unija)**.** Unija u programskom jeziku C je izvedeni tip podataka, koji omogućava da se u istoj memorijskoj lokaciji pamte podaci različitog tipa.

Podaci u uniji mogu se pamtiti po jedan u jednom trenutku, ali ne više podataka istovremeno kao kod struktura.

Kako i strukture, i unije je potrebno definisati, nakon čega se mogu koristiti u deklaracijama. Sintaksa definicije i deklaracije promenljvih unije ista je kao i sintaksa struktura, uz razliku da se umesto ključne reči *struct* koristi ključna reč **unoin**. Sintaksa definicije unije je sledeća:

$$
\begin{aligned}
\langle def_unije \rangle \;::=& \\
& \textbf{union}\ [\langle naziv_unije \rangle] \\
& \{ \\
& \quad \langle tip_1 \rangle\ \langle promenljiva_1 \rangle; \\
& \quad \langle tip_2 \rangle\ \langle promenljiva_2 \rangle; \\
& \quad \ldots \\
& \quad \langle tip_N \rangle\ \langle promenljiva_N \rangle; \\
& \}\ [\langle identifikator_1 \rangle, \langle identifikator_2 \rangle, \ldots]\ ;
\end{aligned}
$$

Sintaksa deklaracije promenljivih tipa unije je:

$$
\begin{aligned}
\langle dekl._prom. \rangle \;::=& \\
& [\textbf{union}]\ \langle naziv_unije \rangle \langle prom. \rangle\ \{, \langle prom. \rangle\}\ ;
\end{aligned}
$$

Kao i kod struktura, u ANSI C-u je neophodno prilikom deklaracije promenljive navesti ključnu reč *union*, dok se ova ključna reč može izostaviti kod C99 i novijih kompajlera.

Za pristup elementima unije takođe se koriste operatori `.` i `->`.

Suštinsku razliku između struktura i unija ilustrovaćemo na primeru strukture:

```
1 struct primer
2 {
3     int x;
4     float y;
5 };
```

i unije:

```
1 union primer
2 {
3     int x;
4     float y;
5 };
```

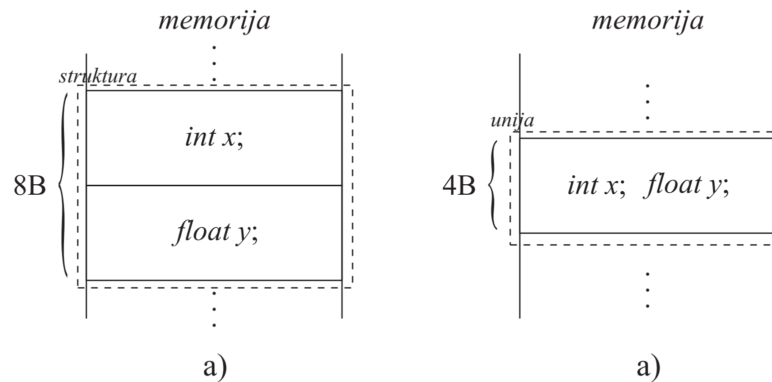
Razlika je u zauzeću memorije. Struktura `primer` zauzima

`sizeof(int) + sizeof(float)`

bajtova, dok unija `primer` zauzima

`max(sizeof(int), sizeof(float))`

U okruženju Microsoft Visual Studio 2010 gore navedena struktura zauzima 8 bajtova, a unija 4 bajta. Ovo je prikazano na slici 4.9.



Slika 4.9: Memorijski prostor koji zauzima: a) struktura *primer*, b) unija *primer*

Kao što je u definiciji 4.16 navedeno, unije omogućavaju da se u istoj memorijskoj lokaciji pamte podaci različitog tipa.

Neka je P promenljiva tipa `union primer`. Tada, pristup elementu unije x, u oznaci `P.x` i pristup elementu unije y, u oznaci `P.y` predstavljaju pristup istoj memorijskoj lokaciji (slika 4.9). Ako se pristupa preko `P.x`, tada se sadržaj lokacije posmatra kao celobrojni *int*, a ako se pristupa preko `P.y` sadržaj se posmatra kao realni *float*.

Primer 4.30 (Pristup elementima unije)**.** U sledećem programu deklarisana je promenljiva P kao unija tipa *primer* i realnom elementu unije `P.y` dodeljena je vrednost 13.0. Ova vrednost se u memorijsku lokacuju upisuje kao *float*, odnosno po IEEE 754 standardu za jednostruku tačnost (10. linija programa).

```
1  #include "stdio.h"
2  union primer
3  {
4      int x;
5      float y;
6  };
7  main()
8  {
9      union primer P;
10     P.y = 13.0;
11     printf("%d",P.x);
12 }
```

U 11. liniji prikazan je element `P.x`, koji je definisan kao celobrojni element unije. Kako se radi o istoj memorijskoj lokaciji, u 10. liniji je u tu lokaciju upisana konstanta 13. u formatu za realne brojeve, a u liniji 11. se taj isti binarni zapis čita po celobrojnom formatu u dvojičnom komplementu. S obzirom na to da je eksponent pri normalizaciji realnog broja 13.0 različit od nule, a on se nalazi u prvom od 4 bajta, celobrojno tumačenje zapisa je prilično veliki broj ceo broj. Konkretno, ovaj program na izlazu daje vrednost

```
1095761920
```

△

4.2.7 Definisanje novih tipova

Za definisanje novog identifikatora, odnosno naziva tipa podataka u C-u koristi se ključna reč **typedef**. Sintaksa je sledeća:

$$\langle novi_tip\rangle ::= \mathbf{typedef}\langle poznati_tip\rangle\langle novi_tip\rangle$$

$$\langle novi_tip\rangle ::= \langle identifikator\rangle$$

$$\langle poznati_tip\rangle ::= [\mathbf{struct}|\mathbf{union}]\ \langle identifikator\rangle$$

Primer 4.31 (Redefinicija naziva tipa promenljivih *int*)**.** Sledeći program korišćenjem ključne reči *typedef* redefiniše naziv tipa *unsigned int* u novi i kraći naziv za ovaj tip: *"celi"*.

```
1  #include "stdio.h"
2  typedef unsigned int celi;
3  main()
4  {
5      celi A;
```

```
6     A = 3;
7     printf("%d",A);
8 }
```

△

Definicija novog tipa se može naći van glavnog programa, kao što je dato u primeru 4.31, ili u glavnom programu. Od pozicije gde se nalazi definicija zavisi vidljivost novog tipa u programu. Ako je definicija data van glavnog programa, vidljiva je u glavnom programu, u svim funkcijama, strukturama i unijama. Ukoliko se definicija nalazi u glavnom programu, samo je u njemu i vidljiva.

Pored skraćivanja naziva postojećih tipova, definicija novih tipova je korisna kako bi se izbeglo pisanje ključnih reci *struct* i *union* kod deklaracije promenljivih.

Primer 4.32 (Definisanje novog tipa za strukturu)**.** U sledećem primeru pokazano je kako je moguće nakon definisanja strukturnog tipa definisati novi naziv tipa i time izbeći prisanje ključne reči *struct* prilikom deklaracije promenljive.

```
 1 #include "stdio.h"
 2 struct tacka
 3 {
 4     int x;
 5     int y;
 6 };
 7 typedef struct tacka tacka;
 8 main()
 9 {
10     tacka T;
11     printf("%d",T.x);
12 }
```

Kao sto se iz primera vidi, novi tip može imati isto ime kao i struktura. △

Prilikom deklaracije strukture kod C99 i novijih kompajlera, kompajler automatski dodaje definiciju novog tipa koji ima identifikator kao i definisana struktura, pa je zbog ovoga moguće izostaviti ključnu reč *struct* prilikom deklaracije promenljivih.

Kontrolna pitanja

1. Šta je određeno tipom promenljive?
2. Koji su osnovni tipovi podataka u C-u?
3. Koja je minimalna, a koja maksimalna vrednost broja koji se može zapamtiti u 2-bajtnoj (16b) promenljivoj u dvojičnom komplementu?
4. Na osnovu podataka iz tabele 4.2 (strana 164) odrediti okolinu broja 0 iz koje nije moguće predstaviti brojeve *float* formatom.
5. Na osnovu podataka iz tabele 4.2 (strana 164) odrediti karakteristične brojeve R_{min} i R_{max} (slika 4.1) formata *float*.
6. Napisati program koji prikazuje ASCII tabelu tako što za svaki karakter prikazuje ASCII kod u dekadnom i heksadekadnom brojevnom sistemu, kao i sam simbol. Tabeli prilikom prikaza pridodati odgovarajuće zaglavlje, a pravilnim izborom konverzionih karaktera u formatu funkcije *printf* obezbediti da se kodovi ispisuju jedan ispod drugog, poravnati uz desnu ivicu kolone u kojoj se nalaze. Kolone prilikom prikaza razdvojiti simbolom '|'.
7. Napisati program koji korišćenjem binarnog operatora '&' proverava da li je četvrti bit unetog celog broja jednak 0, ili 1. Prikazati odgovarajuću poruku.
8. Upotrebom ternarnog operatora grananja napisati makro za određivanje većeg od dva broja.
9. Navesti redosled primene operatora u sledećem izrazu ispisivanjem izraza uz korišćenje zagrada.

```
a*=b+++c!=d>>e+~f+--a
```

10. Šta se ispisuje naredbom *printf* iz poslednje linije sekvence datog koda?

```
1    int a,b,c,d;
2    int *p;
3    a=2; b=6; p=&a;
4    c=a*(*p);
5    d-=a*++b;
6    printf("c␣=␣%d␣d␣=␣%d", c, d);
7
```

11. Šta se ispisuje naredbom *printf* iz poslednje linije sekvence datog koda?

```
1    int a=1,b=2,c=3,d=0;
2    int *p;
3    p=&a;
4    d+=a+(*p);
5    p=&b;
```

```
6       d+=a+(*p);
7       p=&c;
8       d+=a+(*p);
9       printf("d␣=␣%d", d);
10
```

12. Deklarisati četiri promenljive A, B, C i D, i korišćenjem pokazivača i *sizeof* operatora proveriti da li se promenljive nalaze na sukcesivnim memorijskim lokacijama. Tip promenljivih izabrati proizvoljno.

13. Nacrtati strukturni dijagram toka algoritma i na programskom jeziku C napisati strukturni program koji određuje i prikazuje zbir cifara zadatog trocifrenog broja.

14. Nacrtati strukturni dijagram toka algoritma i na programskom jeziku C napisati program koji prikazuje sve trocifrene brojeve čiji je zbir cifara deljiv zadatim brojem b. Broj b zadaje korisnik.

15. Nacrtati strukturni dijagram toka algoritma i na programskom jeziku C napisati strukturni program koji određuje i prikazuje zbir cifara unetog N-tocifrenog pozitivnog celog broja. Prikazati i ukupan broj cifara zadatog broja.

16. Napisati definiciju strukturnog tipa za pamćenje podataka o kružnici. Da bi se zapamtili podaci o kružnici potrebno i dovoljno je zapamtiti koordinate centra i dužinu poluprečnika. Deklarisati dve promenljive strukturnog tipa K1 i K2.

17. Napisati definiciju strukture za pamćenje podataka o kompleksnim brojevima. Kompleksni broj čini jedan celobrojni podatak za realni deo i jedan celobrojni podatak za imaginarni deo. U glavnom programu sa tastature uneti dva kompleksna broja i prikazati njihov zbir. Pravilnim izborom formata funkcija *scanf* i *printf* omogućiti da se kompleksni brojevi unose i prikazuju u formatu $x + iy$.

18. Definisati strukturu za pamćenje koordinate tačke (x, y). Korišćenjem definisane strukture definisati strukturu za pamćenje veličine, pozicije i ugla pod kojim se nalaze stranice kvadrata u koordinatnom sistemu. Za elemente strukture uzeti minimalni skup podataka koji je dovoljan da bi se zapamtile tražene karakteristike.

19. Šta su unije i koja je ključna razlika između unija i struktura?

20. Šta prikazuje sledeći program?

```
1 #include "stdio.h"
2 union primer
3 {
4     int x;
```

```
 5     int y;
 6 };
 7 void main()
 8 {
 9     primer A;
10     A.x = 10;
11     A.y = 15;
12     printf("%d",A.x);
13 }
14
```

5

Osnovne strukture podataka

Struktura podataka u opštem smislu je skup od više podataka, koji su na neki način povezani. Veze koje postoje između podataka uglavnom modeluju veze koje postoje među podacima u stvarnom svetu za problem koji se rešava na računaru.

Potrebno je naglasiti da je struktura podataka termin koji postoji u programiranju uopšte, i generalniji je od strukturnog tipa podataka (ključna reč *struct* u C-u). Za kreiranje složenijih struktura podataka koriste se osnovni i izvedeni tipovi podataka. Strukturom podataka opisuju se podaci iz domena problema koji se rešava na računaru.

5.1 Linearne i nelinearne strukture podataka

Generalno, sve strukture mogu se podeliti u dve grupe:

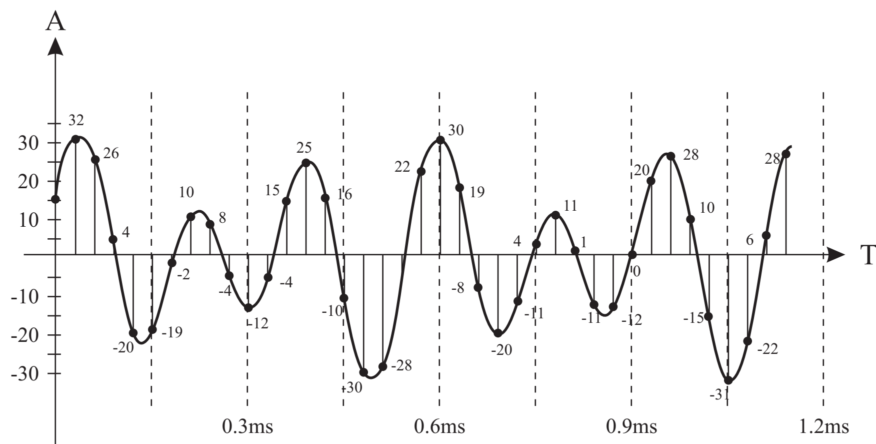
1. linearne, i
2. nelinearne strukture.

Definicija 5.1 (Linearne strukture)**.** Linearna struktura je skup podataka gde svaki podatak ima tačno dva susedna elementa, sem dva elementa na krajevima strukture, koji imaju po jednog suseda.

Kaže se da svaki element ima levog i desnog suseda.

Definicija 5.2 (Nelinearne strukture)**.** Nelinearna struktura je skup podataka u kome ima podataka koji imaju više od dva susedna elementa.

Na primer, linearnom strukturom se može opisati muzička numera na CD-u. Naime, digitalni zapis zvuka je niz sukcesivnih celih brojeva, poređanih jedan za drugim, koji redom predstavljaju vrednosti amplitude zvuka u datim trenucima. Ovo je ilustrovano na slici 5.1. Kod zapisa u CD kvalitetu, zvuk se zapisuje sa 44.100 16-bitnih brojeva za jednu sekundu zvuka. Sve manje od toga je lošiji kvalitet zvuka od CD kvaliteta.



A={32,26,4,-20,-19,-2,10,8,12,-4,15,25,16,-10,-30,-28,-22,30,
19,-8,-20,-11,4,11,1,-11,-12,0,20,28,10,-15,-31,-22,6,28,...}

Slika 5.1: Primer predstavljanja amplitude zvuka nizom celih brojeva

S druge strane, kao primer nelinearnih struktura može se navesti primer evolutivnih veza među životinjama. Ukoliko se jedna životinjska vrsta predstavi novim strukturalnim tipom, vezama između njih moguće je modelovati evolutivni tok, itd.

U linearne strukture ubrajamo:

1. polja,
2. magacine,
3. redove, i
4. lančane liste.

U nelinearne strukture ubrajamo:

1. stabla, i
2. grafove.

Zbog celovitosti slike, u ovom poglavlju ćemo konceptualno predstaviti najzastupljenije strukture, ali ćemo detaljnije obraditi jedino strukture linearnih polja.

5.2 Polja

Polja su najjednostavnije linearne strukture podataka, gde su svi elementi jednog polja istog tipa.

Definicija 5.3 (Polja)**.** Polje je homogena linearna struktura podataka, kod koje se podaci u memoriji nalaze u sukcesivnim memorijskim lokacijama.

Svi podaci u jednom polju dele zajedničko ime, a svaki podatak u polju jednoznačno je određen imenom polja i svojim indeksom, odnosno indeksima. U zavisnosti od broja indeksa koji jednoznačno određuju pojedinačni element polja razlikujemo:

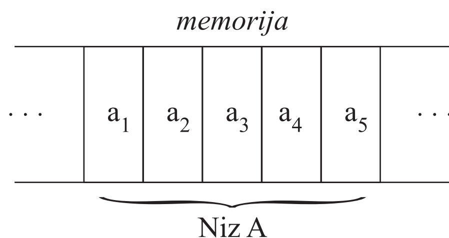
1. jednodimenzionalna polja,
2. dvodimenzionalna polja, i
3. višedimenzionalna polja.

Jednodimenzionalna polja se drugačije nazivaju i **nizovi**, a dvodimenzionalna polja **matrice**.

Uobičajeno je da se pojedinačni elementi u algoritmima navode u matematičkoj notaciji, gde se indeks elementa piše dole desno uz ime niza (u *subskriptu*), ili navođenjem indeksa u malim zagradama. U opštem slučaju jedan element N-dimenzionalnog polja A, sa indeksima $i_1, i_2, \ldots, i_N$, je

$$A(i_1, i_2, \ldots, i_N).$$

Na slici 5.2 prikazan je niz A sa 5 elemenata. Matematička notacija elemenata niza sa slike 5.2 je a_1, a_2, a_3, a_4 i a_5.[1]



Slika 5.2: Jednodimenzionalno polje podataka (niz)

Polja se koriste kada je potrebno memorisati veću količinu podataka. Dodeljivanjem jedinstvenog identifikatora svim podacima eliminiše se problem davanja posebnih imena svakom podatku, a uvođenje indeksa značajno pojednostavljuje obradu elemenata u petlji. Kod indeksiranih promenljivih jednostavno je implementirati obradu koju je potrebno izvršiti nad svakim elementom, indeksiranjem elemenata redom u petlji.

[1]U matematici se obično sam niz, odnosno polja označavaju velikim slovom, a elementi niza istim tim, ali malim slovom (niz A, elementi $a_1, a_2, \ldots$). S obzirom na to da se kod nekih programskih jezika razlikuju promenljiva definisana malim i promenljiva definisana velikim slovom (*case-sensitive*), u programiranju se koristi ista oznaka i za niz i za elemente niza.

5.2.1 Statička deklaracija polja

Polje je pre upotrebe neophodno deklarisati, kao i svaku drugu promenljivu, čime se rezerviše prostor za polje u memoriji i zadaje simboličko ime. Sintaksa statičke deklaracije polja u EBNF notaciji je:

$$\begin{aligned} \langle deklar._polja \rangle \; &::= \\ & \langle tip \rangle \; \langle identifikator \rangle \; '[' \; \langle broj_elemenata \rangle \; ']' \\ & \left\{ '\,[' \langle broj_elemenata \rangle ']' \right\} \\ \langle tip \rangle \; &::= \; \mathbf{char} | \mathbf{short} | \mathbf{int} | \ldots | \langle strukturni_tip \rangle \\ \langle broj_elemenata \rangle \; &::= \; \langle celobrojna_konstanta \rangle \end{aligned}$$

Drugim rečima, kao i kod promenljive bilo kog drugog tipa, deklaracija polja počinje navođenjem tipa elemenata ($\langle tip \rangle$). Svi elementi polja će biti ovog tipa.

Nakon navođenja tipa, navodi se identifikator koji označava ime niza, a odmah iza imena navodi se broj elemenata niza u okviru srednjih zagrada '[' i ']'.[2] Ovo je ukupan broj elemenata u prvoj dimenziji polja. Ukoliko se ovim jednim brojem elemenata završi deklaracija, biće deklarisano jednodimenzionalno polje, odnosno niz. Primer deklaracije niza sa 100 celobrojnih elemenata tipa *int* je:

```
int A[100];
```

Napomena: Dimenzije koje se navode u uglastim zagradama **moraju** biti celobrojne konstante. Statičkom deklaracijom nije moguće omogućiti korisniku da zada dimenzije polja, t.j. u uglastim zagradama kod statičke deklaracije nije dozvoljeno navođenje promenljivih. Programer treba da zada veličinu polja tako da zadovolji potrebe algoritma u svakom slučaju.

Kako je u EBNF opisu sintakse navedeno, nakon prve dimenzije, moguće je navesti 0 ili više dodatnih dimenzija (zagrade { i }). Broj elemenata svake dimenzije navodi se u okviru srednjih zagrada. Primer deklaracije dvodimenzionalnog polja, odnosno matrice je:

```
int A[50][50];
```

Ova matrica ima ukupno 50x50=2.500 celobrojnih elemenata. Ukoliko je veličina podatka tipa *int* 4 bajta, prethodno deklarisana matrica zauzeće 10.000 bajtova ($\approx 10kB$).

Elementi polja se prilikom deklaracije mogu inicijalizovati navođenjem vrednosti svih elemata, razdvojenih zarezom, u okviru zagrada { i }. Na primer, polje `A` sa 5 elemenata može se inicijalizovati prilikom deklaracije na sledeći način:

```
int A[5] = {4,7,18,2,12};
```

[2]U EBNF oznaka za srednje zagrade, kao i za bilo koje druge simbole navodi se pod apostrofima, da bi se razlikovalo od EBNF oznaka [i] za opcione elemente.

Broj elemenata polja se ne mora navesti ukoliko se polje inicijalizuje prilikom deklaracije. Ovu informaciju kompajler može dobiti automatski prebrojavanjem navedenih vrednosti, pa se prethodna deklaracija može napisati i kao:

```
int A[] = {4,7,18,2,12};
```

Inicijalizacija matrice prilikom deklaracije može se uraditi kao što je pokazano na sledećem primeru.

```
int A[3][3] = {
                 {1,2,3},
                 {4,5,6},
                 {7,8,9}
               }
```

Iz deklaracije matrice, ukoliko se vrednosti inicijalizuju prilikom deklaracije, može se izostaviti samo prva dimenzija. Za prethodni primer deklaracija bi bila:

```
int A[][3] = {
                 ...
             }
```

Generalno, kod inicijalizacije prilikom deklaracije N-dimenzionalnog polja može se izostaviti samo jedna dimenzija, jer kompajler na osnovu ukupnog broja navedenih inicijalizacionih vrednosti i ostalih $N-1$ dimenzija može odrediti tu jednu nedostajajuću dimenziju. Kod inicijalizacije prilikom deklaracije jednodimenzionalnog polja ne mora se navesti nijedna dimenzija, kod dvodimenzionalnog mora bar jedna, itd.

5.2.2 Pristup elementima polja

Svaki element polja određen je identifikatorom niza i svojim indeksom (indeksima) u polju. Pojedinačnom elementu polja može se pristupiti navođenjem imena, odnosno identifikatora polja i konkretnih indeksa elementa. Indeksi elementa navode se sa desne strane identifikatora polja u okviru srednjih zagrada, i to svaki indeks posebno na sledeći način:

$$\langle pristup_elem._polja\rangle \quad ::= \quad \langle identifikator\rangle'\,['\langle indeks_1\rangle']' \left[\,'\,['\,\langle indeks_i\rangle']'\ \ldots\right]$$

$$\langle indeks\rangle \quad ::= \quad \langle promenljiva\rangle \mid \langle konstanta\rangle \mid \langle izraz\rangle$$

Na primer, u sledećem programu izvršena je deklaracija jednodimenzionalnog niza A sa 5 elemata i u elemenat sa indeksom 2 upisana je vrednost 10.

```
1 main()
2 {
3     int A[5];   // deklaracija
4     A[2] = 10;  // pristup elementu
5 }
```

Napomena: Prilikom pristupa elementima polja u uglastim zagradama za indekse elementa moguće je navesti konstante, promenljive, ili celobrojne izraze.

U programskom jeziku C prvi element polja ima sve indekse jednake 0, t.j.

$$i_1 = 0, i_2 = 0, \ldots, i_N = 0,$$

a poslednji element ima indekse:

$$i_1 = e_1 - 1, i_2 = e_2 - 1, \ldots, i_N = e_N - 1,$$

gde su $e_1, e_2, \ldots, e_N$ brojevi elemenata u prvoj, drugoj, ..., N-toj dimenziji.[3]

Ukoliko je niz sa 10 elemenata deklarisan kao `int A[10]`, prvi element niza ima indeks 0 (`A[0]`), a poslednji element ima indeks 9 (`A[9]`). Za slučaj kvadratne matrice dimenzija 5x5 deklarisane kao `A[5][5]` prvi element je `A[0][0]`, a poslednji `A[4][4]`.

Da su svi elementi polja smešteni u sukcesivnim memorijskim lokacijama, kako se tvrdi u definiciji 5.3 (strana 214), može se dokazati izvršavanjem programa iz sledećeg primera.

Primer 5.1 (Prikaz lokacija elemenata niza)**. Zadatak**: Napisati program na C-u koji prikazuje adrese na kojima se nalaze elementi niza.

Rešenje: Za prikaz adresa na kojima se nalaze elementi niza može se koristiti unarni operator referenciranja uz svaki od elemenata pojedinačno, kao $\&A[i]$. Ovo je pokazano u sledećem programu.

```
1 #include "stdio.h"
2 main()
3 {
4     int A[5], i;
5     for (i = 0; i< 5; i++)
6         printf("%d\n",&A[i]);
7 }
```

Izlaz koji je prgram dao prilikom jednog izvršenja je:

[3] U nekim programskim jezicima indeksi niza nakon deklaracije su od 1 do N, a kod nekih jezika je proizvoljno moguće zadati opseg indeksa. Kod programskog jezika C indeksi su od 0 do N-1.

```
3866372
3866376
3866380
3866384
3866388
```

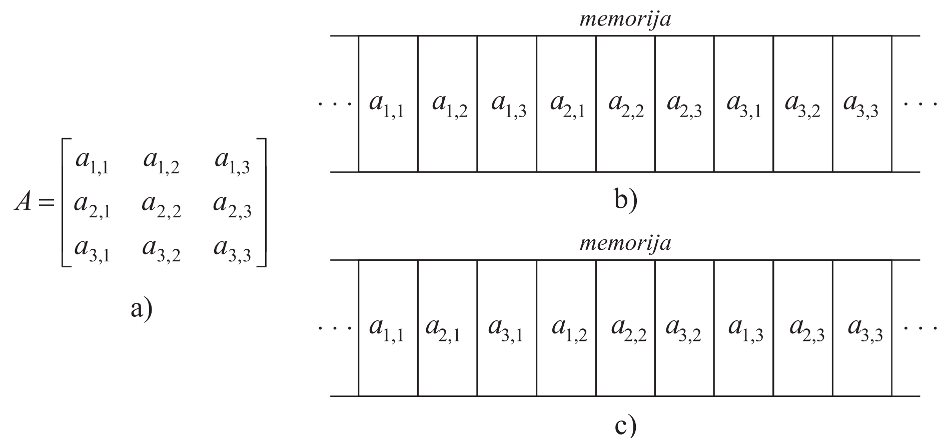
△

5.2.3 Linearizacija polja

Elementi niza su određeni jednim indeksom. U memoriji se elementi pamte u sukcesivnim lokacijama. Elementi matrice su određeni sa dva indeksa, ali se zbog strukture i organizacije operativne memorije računara takođe pamte kao niz elemenata u sukcesivnim memorijskim lokacijama.

Definicija 5.4 (Linearizacija polja)**.** Linearizacija polja je postupak ređanja elemenata višedimenzionalnog polja u memoriji kako bi se svi elementi upisali u sukcesivne memorijske lokacije.

Matrica se može linearizovati na dva načina. Oba načina su prikazana na slici 5.3: **po vrstama** i **po kolonama**. Linearizacija po vrstama izvodi se tako što se u memoriju redom upisuju elementi prve vrste, a nakon poslednjeg elementa prve vrste prvi element druge vrste. Za njim se upisuju i svi ostali elementi druge vrste, i tako redom. Kod linearizacije po kolonama u memoriju se redom upisuju elementi prve kolone, a nakon poslednjeg elementa prve kolone prvi element druge kolone, i tako redom (slika 5.3).



Slika 5.3: Linearizacija dvodimenzionalnog polja: a) matrica u matematičkom zapisu, b) smeštanje matrice po vrstama, c) smeštanje matrice po kolonama.

Ukoliko je matrica A dimenzija NxM zapamćena po vrstama, i ukoliko je adresa prvog elementa matrice $p = \&A[0][0]$, tada se adresa na kojoj se nalazi element $A[i][j]$ može izraziti kao:

$$p + i \cdot M + j. \tag{5.1}$$

Kod matrica zapamćenih po kolonama, ova adresa je:

$$p + i + j \cdot N. \tag{5.2}$$

Analogna operacija linearizacije izvršava se i prilikom linearizacije višedimenzionalnih polja.

5.2.4 Osnovne operacije sa nizovima i matricama

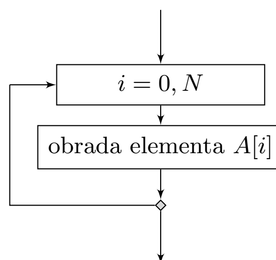
U osnovne operacije nad nekom strukturom ubrajaju se operacije koje se mogu izvršiti jednostavnim obilaskom strukture.

Definicija 5.5 (Obilazak strukture)**.** Obilazak strukture je algoritam po kome se svakom elementu strukture pristupa tačno jednom, da bi se izvršila zadata operacija nad tim elementom.

U osnovne operacije nad nizovima i matricama ubrajamo:

1. unos vrednosti elemenata,
2. prikaz,
3. određivanje minimalne i maksimalne vrednosti,
4. sumiranje vrednosti elemenata, itd.

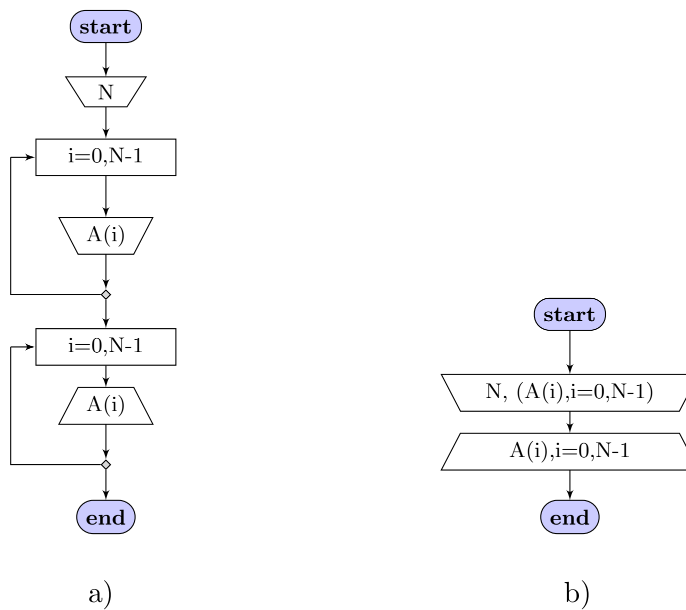
Elementi niza mogu se obići jednom *for* petljom kod koje se brojač koristi za indeksiranje elemenata niza, kao što je prikazano na slici 5.4.



Slika 5.4: Obilazak niza

Primer 5.2 (Unos i prikaz elemenata niza)**. Zadatak**: Nacrtati strukturni dijagram toka algoritma i na programskom jeziku C napisati strukturni program kojim se sa tastature zadaje broj elemenata niza, a za njim i vrednosti elemenata. Nakon unosa niza prikazati na ekranu vrednosti svih elemenata u istom redu, razdvojene zarezom.

Rešenje: Elementi niza mogu se uneti redom, jednim obilaskom niza, a zatim i prikazati kroz obilazak svih elemenata, na način prikazan na slici 5.5a. Kako se unos i prikaz elemenata niza često koristi u algoritmima, u upotrebi je i skraćeni oblik dijagrama toka za unos i prikaz elementa niza, ilustrovan na slici 5.5b. Program na



Slika 5.5: Dijagram toka algoritma za unos i prikaz elemenata niza: a) oblik prilagođen implementaciji, b) skraćeni zapis;

C-u za algoritam prikazan na slici 5.5 je sledeći:

```
1  #include "stdio.h"
2  main()
3  {
4      int A[100], N, i;
5
6      printf("Unesite␣broj␣elemenata␣niza:␣"); // poruka
7      scanf("%d",&N);     // unos broja elemenata
8
9      printf("Unesite␣elementa␣niza:␣"); // poruka
10     for (i = 0; i < N; i++)
```

```
11          scanf("%d",&A[i]);
12
13      printf("Niz␣je:␣"); // poruka
14      for (i = 0; i < N; i++)
15          printf("%d,",A[i]);
16  }
```

Program na izlazu daje:

```
Unesite broj elemenata niza: 5
Unesite elementa niza: 3 9 5 0 18
Niz je: 3,9,5,0,18,
```

Treba napomenuti da je zarez iza poslednjeg elementa bilo moguće izbeći tako što poslednji element ne bi bio prikazan u petlji, a odmah nakon petlje bi bilo potrebno dodati novi *printf* za ovaj element, ali bez zareza.

Elemente niza je moguće prikazati i jedan ispod drugog pomoću

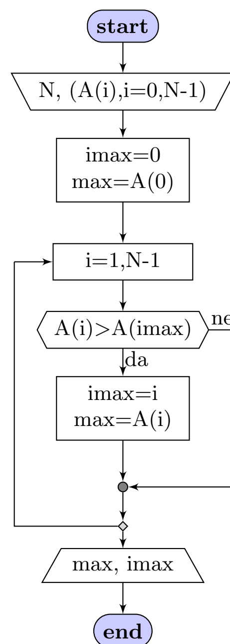
```
printf("%d\n",A[i]);
```

S obzirom na to da se prilikom deklaracije niza za broj elemenata mora navesti konstanta, uobičajeno je da se prilikom deklaracije niza zada maksimalni broj elemenata koji se očekuje od korisnika, a da se korisnik pita koliko elemenata želi da smesti u niz. U ovom primeru deklarisano je i rezervisano mesto za 100 elemenata, ali je prilikom unosa korisnik iskoristio samo prvih 5 pozicija. U ove lokacije su unete vrednosti sa tastature. Prikazane su samo unete lokacije, jer promenljiva N sadrži broj "validnih" elemenata niza. Ostalih 95 lokacija je, uslovno rečeno, ostalo prazno, odnosno neiskorišćeno.

Problem neiskorišćenog prostora je generalno problem kod statičke deklaracije. Memorija se ne koristi efikasno, ali s druge strane, veći problem predstavlja situacija ukoliko korisnik ima potrebu, u ovom primeru, za više od 100 elemenata. U tom slučaju programer mora promeniti deklaraciju, ponovo prevesti program i dati korisniku. △

Primer 5.3 (Određivanje maksimalnog elementa niza)**. Zadatak**: Nacrtati strukturni dijagram toka algoritma i na programskom jeziku C napisati strukturni program koji u nizu koji korisnik zadaje sa tastature određuje i prikazuje maksimalni element i indeks maksimalnog elementa.

Rešenje: Dijagram toka algoritma za određivanje maksimalnog elementa niza prikazan je na slici 5.6. Ovaj algoritam koncipiran je tako da se nakon unosa niza pretpostavlja da je prvi elemenat niza maksimalan, pa se redom obilazi niz, počev od drugog elementa, do kraja niza. Prilikom obilaska proverava se da li postoji veći elemenat od onog za koga se u tom trenutku smatra da je najveći. Ukoliko je elemenat koji se trenutno "posmatra" (do koga se u obilasku došlo) veći, taj elemenat se uzima za maksimalni, a pretraga se nastavlja dalje, pri čemu se sada za proveru uzima novopronađeni elemenat.



Slika 5.6: Dijagram toka algoritma za određivanje maksimalnog elementa niza

Sledeći program na C-u implementira algoritam čiji je dijagram toka prikazan na slici 5.6.

```
1  #include "stdio.h"
2  main()
3  {
4      int A[100], N, i, max, imax; //max ce pamtiti max. vrednost
5                                   //imax ce pamtiti indeks max. elementa
6
7      printf("Unesite␣broj␣elemenata␣niza:␣"); // poruka
8      scanf("%d",&N);     // unos broja elemenata
9      printf("Unesite␣elementa␣niza:␣"); // poruka
10     for (i = 0; i < N; i++)
11         scanf("%d",&A[i]);    // unos el. niza
12
13     imax = 0; // pretpostavimo da je max. na prvoj poziciji
14     max = A[0];
15     // ukoliko od drugog pa do poslednjeg elementa naidjemo
16     // na el. sa vecom vrednoscu, taj postaje max.
17     // Indeks i u petlji je strogo manji od N, to je do N-1 elementa
```

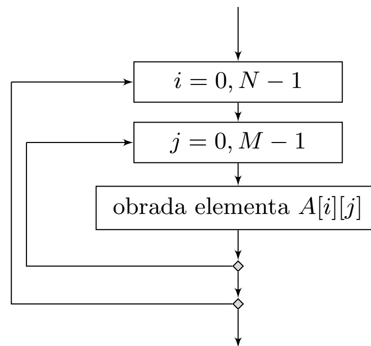
```
18      for (i = 1; i < N; i++)
19          if (A[i] > A[imax])
20          {
21              imax = i;
22              max = A[i];
23          }
24
25      printf("Maksimalni el. je %d i ima indeks %d u nizu.", max, imax);
26  }
```

Program daje sledeći izlaz:

```
Unesite broj elemenata niza: 5
Unesite elementa niza: 2 9 5 18 0
Maksimalni el. je 18 i ima indeks 3 u nizu.
```

△

Za obilazak elemenata matrice najčešće se koriste dve *for* petlje, kao što je prikazano na slici 5.7. Prikazani obilazak predstavlja obilazak matrice "po vrstama". Zamenom redosleda petlji dobija se obilazak matrice "po kolonama".

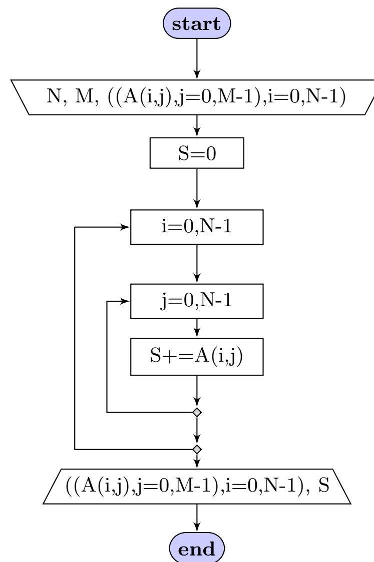


Slika 5.7: Obilazak matrice

Unos, prikaz i elementarni obilazak elemenata matrice ilustrovan je na sledećem primeru.

Primer 5.4 (Unos i prikaz elemenata matrice)**. Zadatak**: Nacrtati strukturni dijagram toka algoritma i na programskom jeziku C napisati strukturni program koji vrši sumiranje elemenata matrice. Dimenzije matrice, kao i elemente matrice zadaje korisnik sa tastature. Prikazati matricu i sumu elemenata matrice.

Rešenje: Dijagram toka algoritma prikazan je na slici 5.8. Unos i prikaz matrice obavljeni su obilaskom matrice po vrstama pomoću dve *for* petlje, kao što je dato na slici 5.7. Međutim, kao i kod nizova, često se sreće skraćeni oblik, koji umesto dve petlje u okviru kojih se nalazi blok za unos, ima samo jedan blok za unos sa tekstom `((A(i,j),j=0,M-1),i=0,N-1)`. Indeks unutrašnje petlje se piše bliže oznaci $A(i,j)$, a indeks spoljašnje petlje se navodi nakon njega. Na slici 5.8 korišćeni su skraćeni oblici unosa i prikaza.



Slika 5.8: Dijagram toka algoritma za unos, prikaz i sumiranje elemenata matrice

Program koji na C-u implementira dijagram toka algoritma sa slike 5.8 dat je u nastavku. Kod pisanja programa, nema "skraćenog oblika" unosa i prikaza, pa je obilazak neophodno implementirati pomoću dve ugnježdene *for* petlje. U 7. liniji programa unose se dimenzije u promenljive N i M, a u linijama od 10. do 12. nalaze se i petlje gde je izvršen unos elemenata.

Indeksi petlji su u granicama od 0 do $N-1$ za i, i od 0 do $M-1$ za j, a ne od 0 do 99, s obzirom na to da je matrica deklarisana kao matrica dimenzija 100x100. Potrebno je napomenuti da je ovo statička deklaracija i da je maksimalno korisniku na raspolaganju matrica dimenzija 100x100, slično kako je razmatrano kod nizova u primeru 5.2.

```
1  #include "stdio.h"
2  main()
3  {
```

```
4       int A[100][100], N, M, i, j, S=0;
5
6       printf("Unesite dimenzije matrice: "); // poruka
7       scanf("%d",&N,&M);  // unos broja elemenata
8
9       printf("Unesite elemente matrice:\n"); // poruka
10      for (i = 0; i < N; i++)
11          for (j = 0; j < N; j++)
12              scanf("%d",&A[i][j]);
13
14      for (i = 0; i < N; i++)
15          for (j = 0; j < N; j++)
16              S += A[i][j];
17
18      printf("Matrica je:\n "); // poruka
19      for (i = 0; i < N; i++)
20      {
21          for (j = 0; j < N; j++)
22              printf("%3d",A[i][j]);
23          printf("%d\n");  // da nije ove naredbe, mat. bi bila prikazana u
        jednom redu
24      }
25      printf("A suma elemenata matrice je %d.", S); // poruka
26  }
```

Izlaz iz programa, odnosno kompletan sadržaj konzole prilikom izvršenja programa dat je u nastavku.

```
Unesite dimenzije matrice: 4 4
Unesite elemente matrice:
2 4 3 1
5 8 9 9
0 1 1 1
2 2 2 2
Matrica je:
  4  2  4  30
  1  5  8  90
  9  0  1  10
  1  2  2  20
A suma elemenata matrice je 54.
```

Može se zaključiti da prilikom unosa vrednosti elemenata matrice iz konzole nije od značaja taster ⟨*enter*⟩, pa korisnik može uneti elemente tako što unosi svaku vrstu u posebnom redu.

Što se tiče prikaza, da nije naredbe iz 23. linije u programu, svi elementi matrice bili bi prikazani u jednom redu. Ovako je nakon ispisivanja svih elemenata jedne

vrste eksplicitno naređen prelazak u novi red. Algoritam nema potrebe opterećivati ovim detaljima implementacije, kao što je i učinjeno na slici 5.8. △

Generalno, za obilazak matrice potrebno je pristupiti tačno NxM elementu. Svakako je jednostavnije izraziti algoritam za obilazak matrice pomoću dve petlje, kao što je to urađeno na slici 5.7. Međutim, matricu je moguće obići i jednom *for* petljom ukoliko se iz indeksa petlje formuliše matematička zavisnost indeksa petlje i indeksa i i j elemenata matrice, slično kao što su izražene zavisnosti u jednačinama (5.1) i (5.2).

Neka je brojač *for* petlje br i neka su dimenzije matrice N i M. Indeksi se mogu izraziti kao:

$$i = \left\lfloor \frac{br}{M} \right\rfloor$$

i

$$j = br \mod M,$$

gde je $br = 0, 1, 2, \ldots, N$x$M - 1$, a $\lfloor x \rfloor$ prvi ceo broj manji od x, odnosno izraz $\lfloor a/b \rfloor$ celobrojno deljenje.

5.2.5 Sortiranje nizova

Veoma česta operacija nad elementima polja je uređivanje elemenata polja po zadatom kriterijumu. Pod uređivanjem polja se podrazumeva zamena vrednosti elementima u cilju ispunjavanja željenog kriterijuma. Najčešći kriterijum uređenosti polja sa numeričkim elementima je relacija:

$$a_i < a_{i+1}, i = 0, 1, \ldots, N - 1.$$

Za polje za koje važi navedena relacija kaže se da je uređeno, odnosno sortirano u **rastući** redosled. Za susedne elemente polja čiji su elementi sortirani u rastući redosled važi uslov da je prethodni elemenat a_i strogo manji od narednog elementa a_{i+1}.

Pored sortiranja elemenata u rastući redosled, numeričke vrednosti se mogu sortirati u neopadajući, opadajući i nerastući redosled. Za elemente niza uređene u **neopadajući** redosled važi da naredni elemenat nije manji od prethodnog, t.j. može biti veći, ili isti kao i prethodni. Za neopadajuće nizove važi relacija:

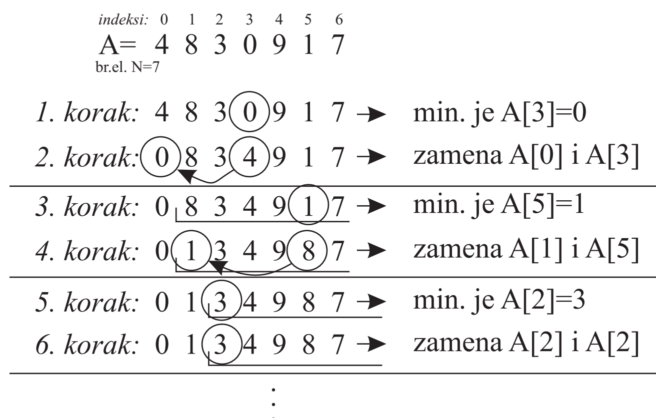
$$a_i \leq a_{i+1}, i = 0, 1, \ldots, N - 1.$$

Kod neopadajućih nizova dozvoljeno je da pojedini elementi imaju iste vrednosti.

Za elemente niza sortirane u **opadajući** redosled važi da je $a_i > a_{i+1}$, a za elemente niza sortirane u **nerastući** redosled važi da je $a_i \geq a_{i+1}$.

Postoji veliki broj različitih algoritama za sortiranje niza. Svi se oni međusobno razlikuju po načinu na koji se sortiranje vrši, a samim tim i po složenosti i brzini.

U nastavku ćemo na primeru sortiranja niza u neopadajući redosled razmotriti tri algoritma za sortiranje niza: sortiranje selekcijom, sortiranje umetanjem, i sortiranje niza zamenom susednih elemenata. Sortiranje u neki drugi redosled je trivijalno i svodi se na promenu kriterijuma uslova algoritma.



Slika 5.9: Primer sortiranja niza selekcijom

Sortiranje selekcijom

Za sortiranje selekcijom (eng. *selection sort*) može se reći da je jedan od najintuitivnijih algoritama za sortiranje elemenata niza. Tekstualni opis algoritma je sledeći:

1. Odrediti minimalni element niza $A[imin]$.
2. Pronađeni minimum postaviti na početak niza (zamena mesta $A[0]$ i $A[imin]$).
3. Odrediti minimalni element neuređenog ostatka niza (ne uzima se u obzir minimalni elemenat koji je stavljen na početak niza).
4. Novoodređeni minimum ostatka poslaviti na početak podniza u kome je taj minimum tražen.
5. Vratiti se na korak 3.

Naziv sortiranje selekcijom je dat ovom algoritmu zato što se konstantno vrši selekcija minimalnog elementa. Ovaj postupak pokazan je na slici 5.9.

Postupak sinteze algoritma uradićemo postupno, korak po korak. Svaki korak ćemo ilustrovati zasebnim dijagramom toka.

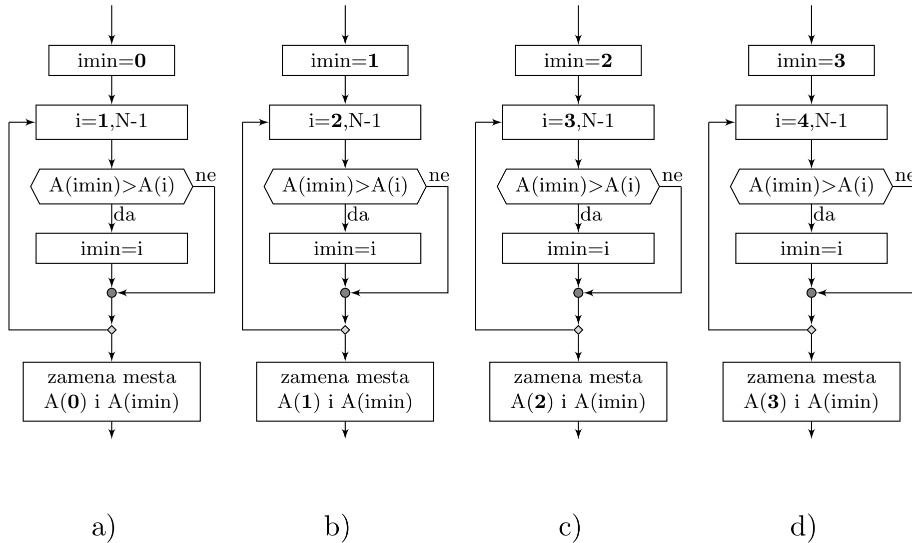
Napomena: Uobičajena praksa pri rešavanju zadataka i sinteze algoritama je da se posebni delovi algoritma posebno razrađuju. Iz "razrade" pojedinačnih koraka algoritma jednostavnije je na kraju sklopiti celinu, koja je, u stvari, i rešenje zadatka.

Dijagram toka 1. i 2. koraka algoritma prikazani su na slici 5.10a. U prvom delu ovog dijagrama toka određuje se minimalni elemenat (1. korak tekstualnog opisa algoritma), a u drugom delu se vrši zamena mesta elementima (2. korak).

Treći korak tekstualnog opisa algoritma je određivanje minimuma u nesortiranom delu niza. Nesortirani deo niza je ceo niz, sem prvog elementa $A[0]$. Na

početku se pretpostavlja da je element $A[1]$ minimalan, pa se traži dalje do kraja niza. Ovo je prikazano na slici 5.10b. Na kraju ovog koraka izvršena je zamena i postavljanje novopronađenog minimuma na početak nesortiranog dela niza.

Dalje se određuje minimum od 2. elementa pa do kraja (slika 5.10c), i taj minimum postavlja na početak. Zatim se određuje minimum počev od 3. elementa (5.10d), i tako dalje do kraja niza.

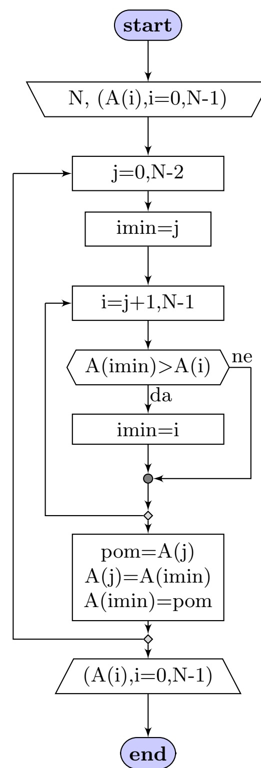


Slika 5.10: Dekompozicija algoritma za soritiranje elemenata niza metodom selekcije

Sa slika 5.10a, b, c i d može se uočiti da su ovi koraci međusobno veoma slični. Razlikuju se u tačno tri detalja: inicijalna vrednost za *imin*, početna granica *for* petlje, i mesto gde se postavlja minimum. Ove vrednosti su na početku 0,1,0 (slika 5.10a), pa 1,2,1 (slika 5.10b), nakon toga 2,3,2 (slika 5.10c), itd. Kako se unapred tačno zna koliko puta je potrebno ponoviti nalaženje minimuma i njegovo postavljanje na početak (broj elemenata niza manje 1, jer poslednji elemenat je sam i nema potrebe sortirati ga), algoritme sa slike 5.10 moguće je postaviti i izvršavati u okviru *for* petlje. Od brojača petlje će zavisiti vrednosti koje se menjaju. Ako je brojač petlje $j = 0, N-2$, tada će ove vrednosti biti $j, j+1, j$. Kompletan dijagram toka algoritma za sortiranje niza selekcijom prikazan je na slici 5.11.

Program napisan na C-u na osnovu dijagrama toka algoritma sa slike 5.11 dat je u nastavku.

```
1  #include <stdio.h>
2  main()
3  {
4      int A[50], N, i, j, imin, pom;
```



Slika 5.11: Dijagram toka algoritma za sortiranje elemenata niza metodom selekcije

```
5       // Unos niza
6       printf("Unesite␣dimenziju␣niza:␣");
7       scanf("%d", &N);
8       printf("Unesite␣elemente␣niza:␣");
9       for (i = 0 ; i < N; i++)
10          scanf("%d", A+i);
11
12      // Za svaki elemenat redom
13      for (j=0; j < N-1; j++)
14      {
15          // pronalazenje minimalnog u ostatku niza
16          imin = j;
17          for (i=j+1; i < N; i++)
18          if (A[imin] > A[i])
19              imin = i;
```

```
20          // postavljanje minim. el. na pocetak
21          pom = A[j];
22          A[j] = A[imin];
23          A[imin] = pom;
24      }
25
26      // Prikaz niza
27      for (i = 0 ; i < N; i++)
28          printf("%d,", A[i]);
29  }
```

Izgled konzole prilikom izvršavanja programa je:

```
Unesite dimenziju niza: 8
Unesite elemente niza: 4 8 3 0 9 1 7 2
0,1,2,3,4,7,8,9,
```

Sortiranje niza u opadajući redosled ovim algoritmom može se izvršiti tako što će se umesto minimuma određivati maksimalni elemenat niza.

Sortiranje umetanjem

Algoritam za sortiranje umetanjem (eng. *insertation sort*) naziv je dobio po specifičnom principu umetanja elemenata na odgovarajuće mesto u niz.

Rečima se ovaj algoritam može opisati na sledeći način: počev sleva udesno, uzeti svaki elemenat niza i pomerati ga ulevo sve do mesta gde bi taj elemenat i trebalo da se nađe u sortiranom nizu. Primer sortiranja niza po ovom algoritmu dat je na slici 5.12.

Po ovom algoritmu, kako je prikazano u koraku 1 na slici 5.12, prvi elemenat ne treba pomerati, s obzirom na to da sa njegove leve strane nema drugih elemenata. Drugi elemenat, $A[1] = 11$, je veći od prvog elementa $A[0] = 1$, pa ga takođe nema potrebe pomerati (korak 2, slika 5.12).

Treći elemenat niza iz primera sa slike 5.12 je $A[2] = 9$. Njegov levi sused je veći od njega ($A[1] = 11$), pa ga je potrebno pomerati ulevo sve dok se ne nađe elemenat od koga je veći. To je u ovom primeru $A[0] = 1$, pa se elemenat $A[2]$ umeće neposredno ispred njega. Nakon ovog umetanja prelazi se na naredni elemenat, $A[3] = 0$, itd.

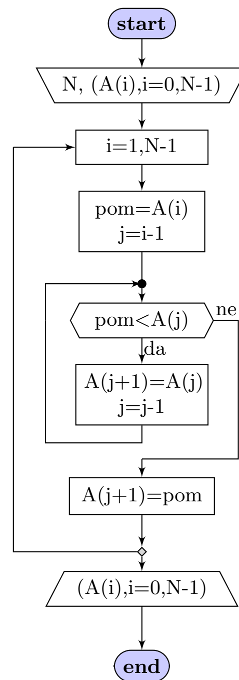
Generalno, svaki elemenat $A[i]$ se "povlači" ulevo dok se ne naiđe na elemenat $A[j]$ za koji važi $A[j] < A[i]$. Kako unapred broj pomeranja elementa nije poznat, ovo je neophodno izvršiti u petlji tipa *while*.

Dijagram toka algoritma za sortiranje umetanjem prikazan je na slici 5.13. Uslovom *while* petlje ispituje se dokle treba ići ulevo, a u samoj petlji se elementi ujedno i pomeraju za po jedno mesto udesno ($A(j + 1) = A(j)$), praveći mesto za umetanje prethodno uzetog elementa $A[i]$ u promenljivu *pom*.

Petlja tipa *while* nalazi se u *for* petlji, jer je broj elemenata koji se pomeraju unapred poznat. Potrebno je pomeriti svaki elemenat niza, sem prvog elementa $A[0]$, koji nema levih suseda. Zbog toga su granice *for* petlje $i = 1, N - 1$.

indeksi:	0	1	2	3	4	5
A= (br.el. N=6)	1	11	9	0	10	3
korak 1:	1	11	9	0	10	3
korak 2:	1	11	9	0	10	3
korak 3:	1	11	9	0	10	3
korak 4:	1	9	11	0	10	3
korak 5:	0	1	9	11	10	3
korak 6:	0	1	9	10	11	3
	0	3	1	9	10	11

Slika 5.12: Primer algoritma za sortiranje umetanjem



Slika 5.13: Dijagram toka algoritma za sortiranje elemenata niza umetanjem

Program na C-u za algoritam čiji je dijagram toka prikazan na slici 5.13 dat je u nastavku.

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

```
2  main()
3  {
4      int A[50], N, i, j, imin, pom;
5      // Unos niza
6      printf("Unesite␣dimenziju␣niza:␣");
7      scanf("%d", &N);
8      printf("Unesite␣elemente␣niza:␣");
9      for (i = 0 ; i < N; i++)
10         scanf("%d", A+i);
11
12     // sortiranje
13     for(i = 1; i < N; i++)
14     {
15         pom = A[i];
16         j = i-1;
17         while(pom < A[j] && j > -1)
18         {
19             A[j+1] = A[j];
20             j--;
21         }
22         A[j+1] = pom;
23     }
24
25     // Prikaz niza
26     for (i = 0 ; i < N; i++)
27         printf("%d,", A[i]);
28 }
```

Sortiranje niza u opadajući redosled moguće je postići negacijom uslova ($pom < A[j]$) *while* petlje.

Sortiranje zamenom susednih elemenata

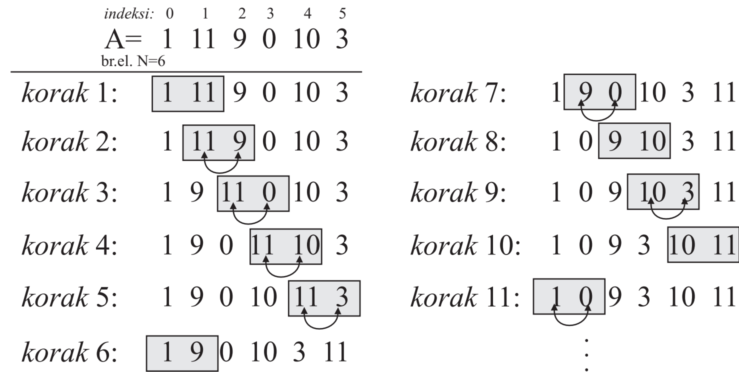
Sortiranje niza zamenom susednih elemenata (eng. *bubble sort*) je jedan od algoritamski najjednostavnijih postupaka sortiranja. Rečima se ovaj algoritam može opisati na sledeći način: krenuti s jedne strane niza i ići do drugog kraja niza; ukoliko redosled dva susedna elementa $A[i]$ i $A[i + i]$ nije u skladu sa željenim redosledom sortiranja, zameniti im mesta.

Ovaj postupak ne garantuje da će niz biti sortiran u prvom prolasku, pa je postupak potrebno ponoviti, ili dok se ne prođe kroz niz a ne načini se nijedna zamena, ili najviše N puta, gde je N dužina niza. Ovaj algoritam je na primeru pokazan na slici 5.14.

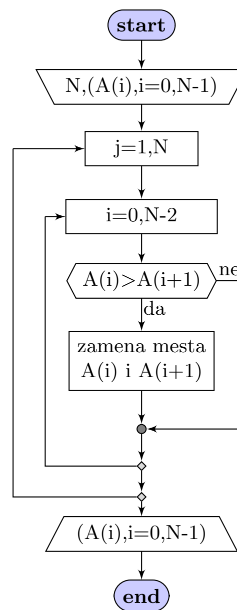
Dijagram toka ovog algoritma za sortiranje niza u rastući redosled prikazan je na slici 5.15.

Program na C-u je dat u nastavku.

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



Slika 5.14: Primer sortiranja niza zamenom susednih elemenata



Slika 5.15: Dijagram toka algoritma za sortiranje elemenata niza zamenom susednih elemenata

```
2  main()
3  {
4      int A[50], N, i, j, imin, pom;
5      // Unos niza
6      printf("Unesite␣dimenziju␣niza:␣");
```

```
7       scanf("%d", &N);
8       printf("Unesite␣elemente␣niza:␣");
9       for (i = 0 ; i < N; i++)
10          scanf("%d", A+i);
11
12      // sortiranje
13      for(j = 1; j <= N; j++)
14          for(i = 0; i < N-1; i++)
15              if(A[i] > A[i+1])
16              {
17                  pom = A[i];
18                  A[i] = A[i+1];
19                  A[i+1] = pom;
20              }
21
22      // Prikaz niza
23      for (i = 0 ; i < N; i++)
24          printf("%d,", A[i]);
25  }
```

5.2.6 Karakteristični delovi matrice

U nekim slučajevima je potrebno izvršiti obradu samo pojedinih karakterističnih delova matrice, kao što su elementi jedne vrste ili kolone, glavne ili sporedne dijagonale, ili elementi trougaonih podmatrica iznad ili ispod ovih dijagonala.

Na slici 5.16 prikazana je matrica A dimenzija 4x4 sa istaknutim elementima jedne vrste i kolone, elementima na glavnoj i sporednoj dijagonali, kao i sa istaknutim elementima iznad i ispod ovih dijagonala.

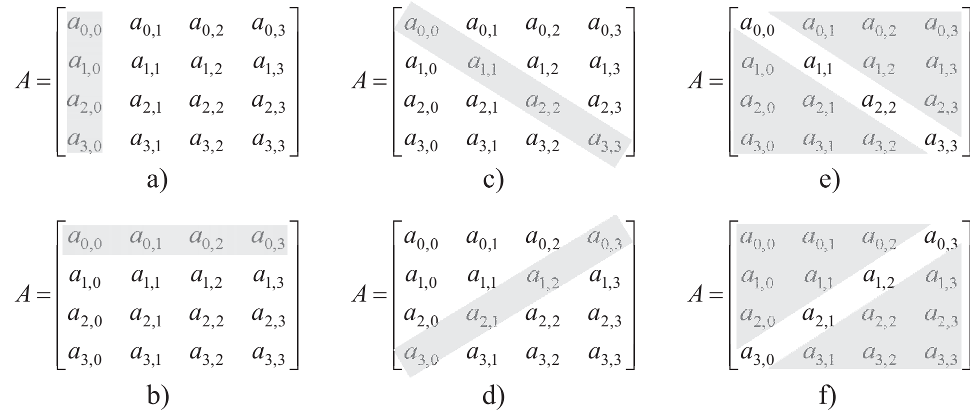
Kao što se sa slike 5.16 može uočiti, postoje određene pravilnosti za indekse po kojima su elemenati grupisani. Na primer, indeksi i i j prve kolone matrice (slika 5.16a) su $(0,0),(1,0),(2,0),(3,0)$, odnosno, drugačije zapisano, indeksi elemenata u jednoj koloni su:

$$\begin{aligned} i &= 0,1,\ldots,N-1 \\ j &= c^{\underline{ta}}, \end{aligned} \tag{5.3}$$

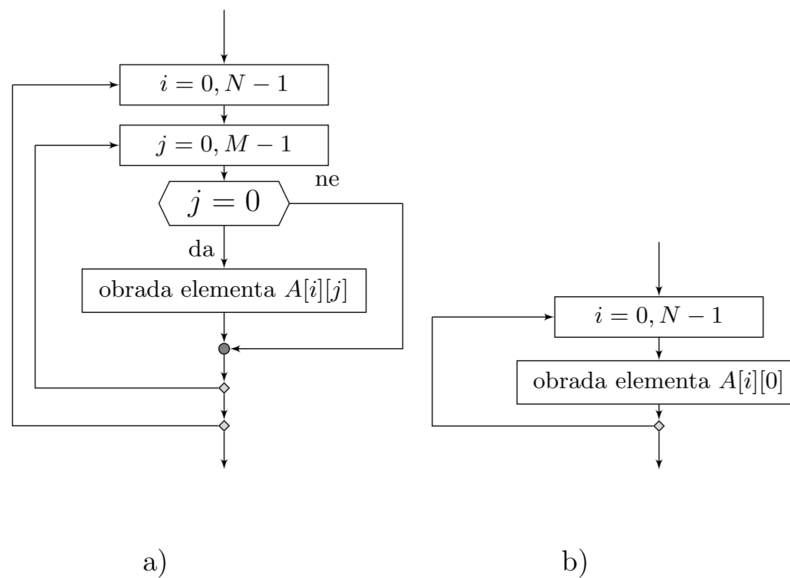
gde je $c^{\underline{ta}}$ konstanta koja označava indeks kolone. Drugim rečima, drugi indeks je konstantan, a prvim se selektuju različiti elementi.

Na osnovu izraza (5.3), moguće je sastaviti različite algoritme za obilazak elemenata jedne vrste matrice. Na slici 5.17 prikazana su dva dijagrama toka algoritma za obilazak elemenata kolone sa indeksom 0.

Na slici 5.17a prikazan je obilazak vrste sa indeksom 0 pomoću klasičnog obilaska matrice sa dve *for* petlje. Za ovu vrstu važi da je $j = c^{\underline{ta}} = 0$, pa se kod ovog algoritma pristupa svim elementima matrice, a obrađuju se samo oni za koje je ispunjen uslov `j=0`.



Slika 5.16: Karakteristične grupe elemenata matrice: a) elementi prve kolone, b) elementi prve vrste, c) elementi glavne dijagonale, d) elementi sporedne dijagonale, e) elementi iznad i ispod glavne dijagonale, f) elementi iznad i ispod sporedne dijagonale;



Slika 5.17: Dijagram toka algoritma za obilazak elemenata jedne kolone matrice: a) pomoću dve *for* petlje, b) pomoću jedne *for* petlje.

Algoritam sa slike 5.17a pristupa svim elementima matrice, kojih ima $N \cdot M$, i za svaki elemenat vrši proveru uslova, da bi obradio samo N elemenata. Rešenje

prikazano na slici 5.17b je efikasnije. Kod algoritma sa slike 5.17b obilazak je izvršen u jednoj petlji, a pristupa se samo onim elementima koje je potrebno obraditi. Iskorišćena je činjenica iz jednačine (5.3) da je drugi indeks konstantan, a da se prvi indeks menja u zadatim granicama. Kod ovog algoritma nema nijedne provere uslova i nepotrebnih pristupa elementima koji se ne obrađuju. Ovaj algoritam se izvršava brže.

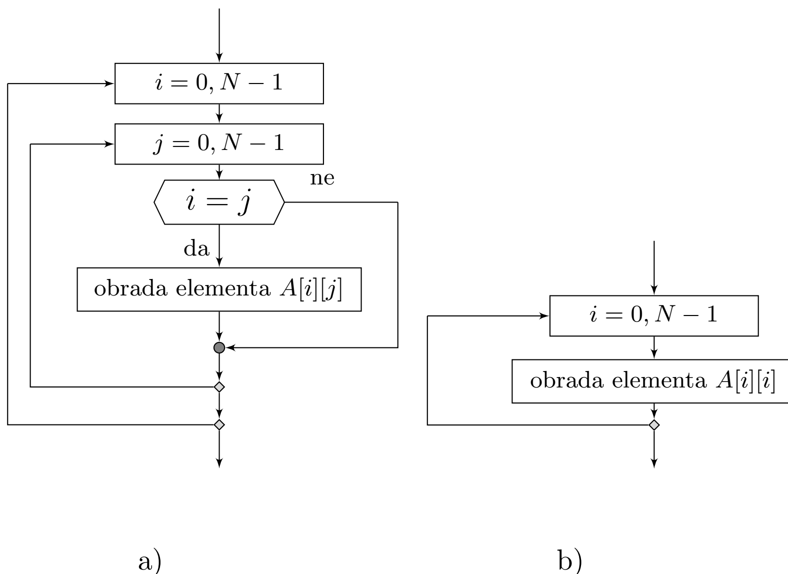
Sve što važi za pristup elementima jedne kolone važi i za elemente jedne vrste (slika 5.16b). Za elemente jedne vrste važi da je prvi indeks konstantan, a drugi se menja, t.j.

$$\begin{aligned} i &= c^{\underline{ta}} \\ j &= 0, 1, \ldots, M-1. \end{aligned}$$

Što se tiče dijagonala, sa slike 5.16c imamo da su indeksi i i j elemenata na glavnoj dijagonali matrice $(0,0), (1,1), (2,2), (3,3)$, odnosno važi uslov:

$$i = j. \tag{5.4}$$

Na osnovu ovoga mogu se takođe sastaviti dva različita algoritma za obilazak elemenata glavne dijagonale kvadratne matrice[4], koji su prikazani su na slici 5.18.



Slika 5.18: Dijagram toka algoritma za obilazak elemenata glavne dijagonale matrice: a) pomoću dve *for* petlje, b) pomoću jedne *for* petlje

[4]Matrica mora biti kvadratna da bi imalo smisla razmatrati dijagonale.

Algoritam sa slike 5.18b izvršava se brže, iz istih razloga kao kod primera sa slike 5.17b. Kod algoritma sa slike 5.18b iskorišćena je činjenica da su indeksi elemenata jednaki pa se pristupa samo elementima $A(i,i)$, redom za $i = 0, 1, \ldots, N-1$.

Elementi sporedne dijagonale sa slike 5.16d su $(3,0), (2,1), (1,2), (0,3)$. Može se uočiti da sa porastom vrednosti indeksa i, vrednost indeksa j opada, pa je samim tim zbir indeksa konstantan[5] i iznosi:

$$i + j = N - 1. \tag{5.5}$$

Sve ostalo što važi za algoritme za pristup elementima glavne dijagonale važi i kod elemenata sporedne dijagonale. Razlika je što bi u ovom slučaju uslov bio `i+j=N-1` za slučaj obilaska sa dve petlje, a u slučaju obilaska jednom petljom obrađivao bi se elemenat `A[i][N-1-i]`.

Sa slika 5.16e i 5.16f, može se zaključiti da za pojedine delove matrice važe uslovi za glavnu i sporednu dijagonalu dati izrazima (5.4) i (5.5), s tim da umesto znaka jednakosti (=) treba da stoji "je manje" (<), ili "je veće" (>), zavisno od grupe elemenata. Na primer, za elemente iznad glavne dijagonale važi $i < j$ (slika 5.16). Ako ovim elementima pridodamo i elemente na glavnoj dijagonali imamo da je $i \leq j$.

Zbog preglednosti, ponovićemo sve uslove, uz dodatak uslova za kvazi-glavne i kvazi-sporedne dijagonale[6]:

1. i-ta vrsta – $i = c^{\underline{ta}}$,
2. j-ta kolona – $j = c^{\underline{ta}}$,
3. glavna dijagonala – $i = j$,
4. iznad glavne dijagonale – $i < j$,
5. ispod glavne dijagonale – $i > j$,
6. kvazi-dijagonale – $i = j + c^{\underline{ta}}$,
7. sporedna dijagonala – $i + j = N - 1$,
8. iznad sporedne dijagonale – $i + j < N - 1$,
9. ispod sporedne dijagonale – $i + j > N - 1$,
10. kvazi-sporedne dijagonale – $i + j = N - 1 + c^{\underline{ta}}$,

Napomena: Sve prethodno navedene uslove moguće je direktno primeniti u uslovu u obilasku sa dve *for* petlje. Međutim, za obilazak jednom *for* petljom nešto je

[5]Vrednost zbira indeksa elemenata na sporednoj dijagonali zavisi od granica indeksa elemenata matrice. Kod programskih jezika kod kojih je prvi elemenat $a_{1,1}$, a poslednji $a_{N,N}$ uslov bi bio $i + j = N + 1$.

[6]Kvazi-glavne i kvazi-sporedne dijagonale su dijagonale koje su paralelne glavnoj, odnosno sporednoj dijagonali.

složenije izvesti zavisnosti kada su u pitanju trougaoni skupovi elemenata iznad i ispod glavne i sporedne dijagonale.

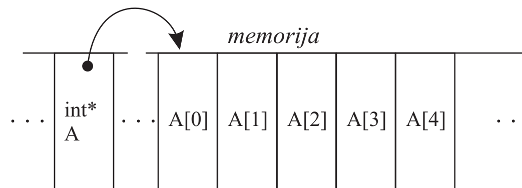
Ako bi se htelo optimalno pristupiti samo elementima, recimo, iznad sporedne dijagonale na način sličan načinu prikazanom na slici 5.18 jednom *for* petljom, potrebno je odrediti granice petlje i zavisnost indeksa i i j od brojača petlje. Brojač petlje br bi bio u granicama od 0 do $\frac{N^2}{2} - \frac{N}{2}$, jer je ukupan broj elemenata matrice N^2, a polovinu elemenata je potrebno umanjiti i za polovinu dijagonale da bi se dobio broj elemenata koji su strogo iznad dijagonale.

U ovom slučaju potrebno bi bilo odrediti zavisnosti indeksa $i = f_1(br)$ i $j = f_2(br)$ po kojima bi se u jednoj *for* petlji sa brojačem $br = 0, \frac{N^2}{2} - \frac{N}{2}$ pristupalo direktno elementima matrice npr. iznad glavne dijagonale pomoću $A[f_1(br), f_2(br)]$.

Jednostavnije rešenje je obilazak pomoću dve *for* petlje, ali tako da granice indeksa unutrašnje petlje zavise od indeksa spoljašnje petlje. Na ovaj način je takođe moguće izbeći ispitivanje uslova i bespotreban pristup svakom elementu matrice.

5.2.7 Nizovi i pokazivači

Prilikom deklaracije niza u memoriji se rezerviše onoliko sukcesivnih lokacija koliko je zahtevano u deklaracionoj naredbi. Pored ovoga rezerviše se i još jedna lokacija pokazivačkog tipa u koju se upisuje adresa prvog elementa niza. Ovo je prikazano na slici 5.19.



Slika 5.19: Struktura podataka u memoriji prilikom deklaracije jednodimenzionalnog polja

Pokazivaču na prvi elemenat niza može se pristupiti navođenjem samo imena niza, bez indeksa. Na primer, ukoliko je niz deklarisan kao

```
int A[10];
```

pokazivač na prvi elemenat niza je samo ime niza, odnosno `A`. Promenljiva A je po tipu pokazivač, odnosno u ovom primeru njen tip je $int*$. To znači da važi ekvivalencija

$$A \equiv \&A[0],$$

gde je A identifikator niza, a & unarni operator referenciranja. Međutim, imajući u vidu pokazivačku algebru, važi i

$$A + i \equiv \&A[i].$$

Drugim rečima, adresu i-tog elementa moguće je dobiti sabiranjem identifikatora niza i celog broja i.

Po ovome, za unos niza se u okviru *scanf* može navoditi bilo $\&A[i]$, kako je navođeno u prethodnim primerima, ili $A + i$, što je ilustrovano sledećim delom programa.

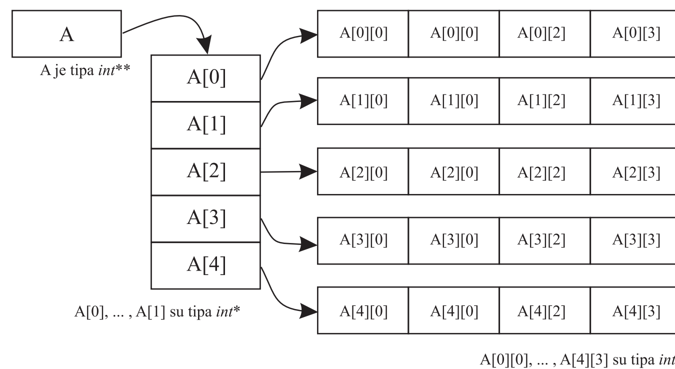
```
1 int A[10], i;
2 for (i=0; i < 10; i++)
3     scanf("%d",A+i);
```

S druge strane, ukoliko se za pristup elementima niza koristi adresa prvog elementa sa pomerajem i do željenog elementa, i operator dereferenciranja, može se napisati ekvivalencija:

$$*(A + i) \equiv A[i].$$

Drugim rečima, elementu niza $A[i]$ može se pristupiti dereferencirajući adresu prvog elementa pomerenu za i elemenata.

Kod deklaracije matrice situacija u memoriji je nešto drugačija. Detaljan prikaz strukture koja se u memoriji rezerviše prilikom deklaracije matrice A dimenzija 5x4 prikazana je na slici 5.20.



Slika 5.20: Struktura podataka u memoriji prilikom deklaracije dvodimenzionalnog polja

Prilikom deklaracije matrice rezerviše se prostor za jedan pokazivač, koji pokazuje na niz pokazivača, gde svaki pokazivač u ovom nizu dalje pokazuje na početak jedne vrste matrice (slika 5.20). Pokazivaču na niz pokazivača pristupa se navođenjem samo simboličkog imena matrice. Na primeru datom na slici 5.20 ovaj pokazivač je označen identifikatorom A. Kako je ovo pokazivač na element pokazivačkog tipa, po tipu ovaj podatak je `(int*)* A;`.

Bitna osobina ove reprezentacije koja se često koristi u C-u je da se navođenjem samo prvog indeksa matrice dobija pokazivač na početak vrste čiji je indeks naveden, što je po tipu isto kao i običan niz, s obzirom na to da je ime niza samo za sebe pokazivač.

Drugačije rečeno, ako je matrica deklarisana kao `int A[10][10];`, tada je `A[3]` po tipu niz, i to niz elemenata četvrte vrste.

5.3 Stringovi

"String" je uobičajeni naziv za jednodimenzionalno polje čiji su elementi karakteri. Stringovi se koriste za pamćenje tekstualnih podataka.

Kako se tekstualni podaci često upotrebljavaju, mnogi programski jezici imaju ugrađenu podršku za stringove kroz dodatne tipove podataka među osnovnim tipovima podataka. Za razliku od drugih programskih jezika, programski jezik C nema osnovni tip podataka za stringove, već stringove implementira kao nizove karaktera.

Zbog ovoga se može reći da na osnovnom nivou C nema dobru podršku za stringove, ne toliko zbog nedostatka tipa, koliko zbog nedostatka operatora za rad sa stringovima. Programski jezik C nema operatore kojima se mogu izvršavati osnovne operacije nad stringovima, već je ove operacije potrebno rešiti algoritamski, kao da se radi sa numeričkim nizovima.

Alternativa je korišćenje gotovih funkcije za rad sa stringovima iz standardne biblioteke, kojih ima dovoljno, a o kojima će više biti reči u poglavlju 6.6.2.

5.3.1 Deklaracija i inicijalizacija stringova

String se statički deklariše kao niz karaktera na sledeći način:

$$\langle string \rangle \quad ::= \quad \textbf{char}\ \langle identifikator \rangle\ '['\ \langle broj_elemenata \rangle\ ']'$$

Na primer, string S sa maksimalno 80 karaktera može se deklarisati kao:

```
char S[80];
```

U ovaj string moguće je upisati i zapamtiti najviše 80 simbola. Kako se radi o statičkoj deklaraciji, maksimalni broj simbola mora biti numerička konstanta definisana u fazi prevođenja programa. Kao i kod nizova, ni pri deklaraciji stringova nije dozvoljeno navođenje promenljive u uglastim zagradama za dužinu stringa.

Svaki pojedinačni element stringa je po tipu karakter. Imajući u vidu sintaksu za navođenje karakter konstanti, sledeći deo programa dodeljuje vrednost prethodno definisanom stringu.

```
1 S[ 0] = 'O';
2 S[ 1] = 'v';
3 S[ 2] = 'o';
4 S[ 3] = '␣';
```

```
 5 S[ 4] = 'j';
 6 S[ 5] = 'e';
 7 S[ 6] = '␣';
 8 S[ 7] = 's';
 9 S[ 8] = 't';
10 S[ 9] = 'r';
11 S[10] = 'i';
12 S[11] = 'n';
13 S[12] = 'g';
```

S obzirom na to da je nizove moguće inicijalizovati prilikom deklaracije navođenjem vrednosti pojedinačnih elemenata u okviru vitičastih zagrada neposredno u deklaraciji, elementi stringa se mogu deklarisati na sledeći način:

```
\index{inicijalizacija}
char S[]={'O','v','o',' ','j','e',' ','s','t','r','i','n','g'};
```

Kako je u deklaraciji nizova moguće izostaviti dimenziju, ova činjenica je iskorišćena u prethodnom primeru.

Kao dodatak koji pojednostavljuje zapis, inicijalizaciju prilikom deklaracije stringa moguće je izvršiti tako što se stringu dodeli literal umesto pojedinačnog navođenja karaktera u uglastim zagradama:

```
char S[]="Ovo je string";
```

5.3.2 *Null-terminated* stringovi

Zbog statičke deklaracije, prostor koji se rezerviše za string može biti, i uglavnom jeste, veći od količine teksta koji programer upisuje u string. Zbog ovoga je na neki način potrebno obezbediti informaciju o broju simbola u stringu.

Za potrebe numeričkih nizova obično se informacija o broju elemenata čuva u dodatnoj numeričkoj promenljivoj. Informacija o dužini stringa u C-u čuva se u samom stringu. Ovo se postiže tako što se nakon poslednjeg validnog slimbola u prvi naredni element upisuje vrednost 0. Kaže se da se ovi stringovi "završavaju" nulom (eng. *null-terminated*).

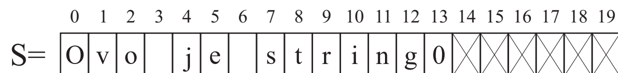
Definicija 5.6 (*Null-terminated* string)**.** *Null-terminated* string je string u koji je nakon poslednjeg validnog simbola upisana nula.

Što se ASCII tabele tiče (slika 4.3), numerički kod za simbol koji označava dekadnu cifru 0 je 48 ($0x30$ heksadekadno), a kod 0 rezervisan je za specijalne namene i za njega nije vezan nijedan simbol na tastaturi. Ovo ga čini idealnim za "kodiranje" kraja stringa.

Na primer, ukoliko se deklariše string S maksimalne dužine 20, i prilikom deklaracije inicijalizuje na sledeći način:

```
char S[20] = "Ovo je string";
```

u memoriji će biti rezervisan prostor za 20 karaktera, prvih 13 će biti popunjeni odgovarajućim simbolima iz navedene inicijalizacije, a u 14. element stringa će biti upisana 0 (element sa indeksom 13). Nakon ovog elementa, sadržaj preostalih elemenata nije bitan. Ovo je ilustrovano na slici 5.21.



Slika 5.21: Primer sadržaja *null-terminated* stringa

Celobrojna konstanta 0 naziva se i **"*null*" konstanta** (izgovara se "*nal*"). Ova konstanta postoji i među karakter konstantama i oznaka joj je '\0', a piše se između znakova navoda kao i sve karakter konstante. Da bi se konstanta razlikovala od simbola za cifru 0, koji je '0', dodata joj je kontra-kosa crta (eng. *backslash*) da označi da se radi o specijalnom simbolu.

S obzirom na prethodnu diskusiju, ukoliko se inicijalizuje string prilikom deklaracije bez navođenja dimenzije, u memoriji će biti rezervisan prostor za sve navedene simbole u inicijalizaciji, plus još jedan karakter za *null* konstantu. Na primer, ukoliko se deklariše string S na sledeći način:

```
char S[] = "Ovo je string";
```

u memoriji će biti rezervisan prostor za 14 karaktera: 13 karaktera uključujući i 2 razmaka, i jedan *null* karakter, kao da je inicijalizacija zapravo izvršena na sledeći način:

```
char S[] = "Ovo je string\0";
```

5.3.3 Unos i prikaz stringova

Za unos i prikaz stringova moguće je koristiti funkcije ***printf*** i ***scanf*** iz biblioteke ***stdio.h*** sa konverzionim karakterom '**%s**'.

Primer 5.5 (Unos i prikaz stringova)**.** Sledeći primer ilustruje unos teksta sa tastature u string R, u koji je moguće zapamtiti maksimalno 80 karaktera[7], i prikaz unetog teksta na ekranu monitora.

```
1 #include "stdio.h"
2 main()
3 {
4     char R[80];
5     scanf("%s",R);
6     printf("\nUneti␣string␣je:␣\"%s\"␣",R);
7 }
```

[7]Standardna veličina konzole je 80x25 karaktera, što je nasleđeno od starih operativnih sistema koji su radili samo u karakter modu, za razliku od grafičkog moda današnjih operativnih sistema. String od 80 karaktera može da prihvati unos jednog reda teksta sa konzole standardne veličine.

Funkcija *scanf* u delu gde je naveden format unosa ima konverzioni karakter '%s', što označava da se sa tastature očekuje unos tekstualnog podatka. U drugom delu se nalazi identifikator stringa R.

Naime, kako je u poglavlju 3.6.1 rečeno, *scanf* zahteva navođenje adrese promenljive u koju se upisuje vrednost uneta sa tastature. S obzirom na to da je sam identifikator niza zapravo adresa prvog elementa u nizu ($R \equiv \&R[0]$), nema potrebe za stringove u okviru *scanf* funkcije navoditi operator referenciranja.

Oznake '\"' u formatu prikaza u 6. liniji ovog primera navedene su da bi string koji se prikazuje bio prikazan pod navodnicima. Pošto je navodnik karakter koji se koristi za označavanje početka i kraja literala, ukoliko sam literal treba da sadrži karakter ", tada se ispred karaktera '"' dodaje simbol '\' da bi označio da se radi o specijalnom karakteru, a ne o kraju stringa.

Izvršenjem prethodnog programa dobija se sledeći izgled konzole:

```
Uvod u programiranje i programski jezik C.

Uneti string je: "Uvod"
```

Funkcija *scanf* ima "formatirani" ulaz, što znači da programer mora navesti tačan redosled i tip unosa koji se očekuje od korisnika. Na primer, za unos dva cela i jednog realnog broja potrebno je zadati format unosa "%d%d%f".

Konverzioni karakter '%s' označava unos **samo jedne reči**, pa je zbog toga prethodni program u string R upisao samo prvu reč do prvog blanko znaka, što se vidi iz navedenog prikaza. Potrebno je napomenuti da funkcija *scanf* automatski dodaje *null* karakter na kraj unetog teksta, kako bi se znalo gde je kraj korisničkog unosa i prilikom prikaza (ili obrade) prikazivao samo "korisni" deo stringa.

Konverzioni karakter '%s' u slučaju funkcije *printf* prikazuje ceo string od zadate početne adrese, R u ovom primeru, pa do kraja stringa označenog *null* karakterom. Imajući u vidu pokazivačku aritmetiku, sledeći prikaz:

```
printf("%s",R+1);
```

za prethodni primer unosa prikazao bi

```
vod
```

△

S obzirom na to da je u prethodnom primeru unešena rečenica u slobodnom formatu na govornom jeziku, formatirani unos funkcije *scanf* predstavljao je ograničenje, jer se retko kada unapred zna koliko će reči imati takva rečenica koju zadaje korisnik. Sledeći primer ilustruje slučaj kada je formatirani unos prednost.

Primer 5.6 (Primer unosa više od jedne reči sa tastature)**.** Sledeći program ilustruje unos imena i prezimena u posebne stringove i njihov nezavisni prikaz.

```
1 #include "stdio.h"
2 main()
3 {
4     char Ime[25], Prez[25];
5     scanf("%s%s", Ime, Prez);
6     printf("Ime:␣%s\nPrezime:␣%s", Ime, Prez);
7 }
```

Kako se od korisnika očekuje unos dve reči, zadat je format unosa "%s%s". Prva reč se upisuje u string *Ime*, a druga u string *Prez*.

Izgled konzole prilikom izvršenja programa dat je u nastavku.

```
Petar Peric
Ime: Petar
Prezime: Peric
```

△

Primer 5.7 (Primer unosa rezultata ispita). **Zadatak**: Napisati program koji od korisnika traži da sa tastature unese brojeve indeksa, imena i prezimena dva studenta, kao i broj poena koji su ostvarili na ispitu i prikaže podatke o studentu sa većim brojem poena.

Rešenje: Od korisnika se očekuje unos celobrojne vrednosti za broj indeksa, unos imena i prezimena, kao i unos celobrojne vrednosti za broj poena, pa je format unosa za svakog studenta "%d%s%s%d". Rešenje zadatka dato je u nastavku.

```
1  #include "stdio.h"
2  main()
3  {
4      char ime1[25], prez1[25], ime2[25], prez2[25]; // imena i prezimena
5      int bri1, p1, bri2, p2;     // brojevi indeksa i poeni
6
7      scanf("%d%s%s%d", &bri1, prez1, ime1, &p1); // prvi student
8      scanf("%d%s%s%d", &bri2, prez2, ime2, &p2); // drugi student
9
10     // ko ima veci broj poena?
11     if (p1 > p2)
12         printf("%s␣%s␣sa␣brojem␣indeksa␣%d␣je␣bolji,␣jer␣ima␣%d␣poena,␣sto␣je␣
       vise␣od␣%d␣poena,␣koliko␣ima␣%s␣%s.", ime1, prez1, bri1, p1, p2, ime2,
       prez2);
13     else
14         printf("%s␣%s␣sa␣brojem␣indeksa␣%d␣je␣bolji,␣jer␣ima␣%d␣poena,␣sto␣je␣
       vise␣od␣%d␣poena,␣koliko␣ima␣%s␣%s.", ime2, prez2, bri2, p2, p1, ime1,
       prez1);
15 }
```

Izgled konzole prilikom izvršenja programa dat je u nastavku.

```
12999 Peric Petar 65
12345 Lazic Lazar 100

Lazar Lazic sa brojem indeksa 12345 je bolji, jer ima 100
poena, sto je vise od 65 poena, koliko ima Petar Peric.
```

△

Nije uvek potrebno imati formatirani ulaz. Postoji puno primera kada je potrebno uneti tekst za koji unapred nije poznato koliko ima reči u njemu.

Funkcija ***gets()*** iz biblioteke ***string.h*** tekst unet sa tastature upisuje u zadati string, od početka unosa pa dok korisnik ne pritisne taster ⟨*enter*⟩ na tastaturi, bez vođenja računa koliko je reči korisnik uneo. Kao i *scanf* i *gets* na kraj unetog teksta u stringu dodaje *null* karakter kao oznaku za kraj.

Sintaksa funkcije *gets* je sledeća:

$$\langle gets \rangle \quad ::= \quad gets(\ \langle identifikator_stringa \rangle \ )$$

Primer 5.8 (Unos stringa funkcijom *gets*)**.** Sledeći primer ilustruje unos celog reda teksta sa tastature i prikaz unetog teksta.

```
1 #include "stdio.h"
2 main()
3 {
4     char R[80];
5     gets(R);
6     printf("\nUneti␣string␣je:␣\"%s\"␣",R);
7 }
```

Izgled konzole prilikom izvršenja programa dat je u nastavku.

```
Uvod u programiranje i programski jezik C.

Uneti string je: "Uvod u programiranje i programski jezik C."
```

△

5.3.4 Osnovne operacije nad stringovima

U osnovne operacije za rad sa stringovima ubrajaju se

1. dodela vrednosti,
2. poređenje sadržaja dva stringa, i
3. konkatenacija (nadovezivanje) dva stringa.

Neki programski jezici imaju operatore kojima je moguće izvoditi ove operacije. C nema operatore za rad sa stringovima, već je potrebno ove probleme rešavati algoritamski u programu, ili koristiti neke od specijalizovanih biblioteka standardnih funkcija, o čemu će biti više reči u poglavlju 6.6.2.

Što se algoritamskog predstavljanja tiče, osnovne operacije za rad sa stringovima mogu se predstavljati operatorski, što skraćuje pisanje i čini algoritam čitljivijim. Za operatore se obično koriste neki od operatora dostupnih u drugim programskim jezicima.

Tako na primer, za dodelu vrednosti stringa stringu je u algoritamskoj reprezentaciji moguće koristiti oznake $string1 = string2$, ili $string1 \leftarrow string2$.

Operacija poređenja stringova se u algoritmima takođe može predstaviti operatorski, navođenjem binarnog operatora '=', ili '≠'. Za leksikografsko poređenje sadržaja stringova mogu se koristiti operatori $<, >, \leq$ i $\geq$. Poređenje stringova vrši se po leksikografskom redosledu na osnovu engleske abecede, odnosno, generalno rečeno, po poziciji karaktera u ASCII tabeli, jer je u programiranju moguće porediti i simbole koji ne moraju nužno biti slova.

Za konkatenaciju obično se koriste operatori $+$, $||$, ili &. Konkatenacija stringova u algerotmima može se predstaviti na sledeći način:

```
s1 = "Ovo";
s2 = " je";
s3 = " string";
S = s1+s2+s3;
```

Dodela i kopiranje vrednosti

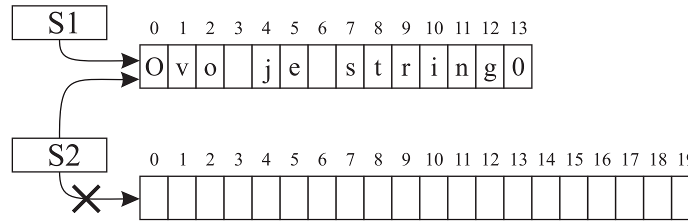
Kao što je već rečeno, programski jezik C nema operatore za rad sa stringovima, uključujući i operator dodele vrednosti u nekoj formi simbola '=', koji bi sadržaj jednog stringa mogao da dodeli sadržaju drugog stringa.

Međutim, u C-u je sintaksno ispravno napisati sledeće:

```
1  main()
2  {
3      char S1[] = "Ovo je string";
4      char S2[20];
5      S2 = S1;
6  }
```

Ipak, prethodna dodela vrednosti ne znači da će sadržaj jednog stringa biti kopiran u drugi string. Naime, sam indetifikator niza u C-u je pokazivač na prvi element niza, pa je prethodno napisana dodela zapravo **dodela adrese pokazivaču**. Ovo je ilustrovano na slici 5.22.

Nakon dodele vrednosti $S2 = S1$ oba pokazivača sadrže istu adresu, i to adresu niza karaktera $S1$. Nadalje se nizu $S2$ iz programa ne može više pristupiti. Ovaj memorijski prostor ostaje rezervisan, ali ga je nemoguće koristiti, jer je adresa početka niza prilikom dodele izgubljena.



Slika 5.22: Semantički neispravna dodela vrednosti stringa stringu

Ukoliko se nakon dodele vrednosti $S2 = S1$ navede

```
printf("%s", S2);
```

biće prikazan string "*Ovo je string*", ali, formalno govoreći, ovo je vrednost stringa $S1$, a ne kopirana vrednost u string $S2$.

Da bi se sadržaj jednog stringa kopirao u drugi, neophodno je prebacivati jedan po jedan simbol iz stringa $S1$ u string $S2$, od početka stringa pa do kraja označenog "nal" simbolom. Program koji kopira sadržaj stringa S1 u string $S2$ dat je u nastavku.

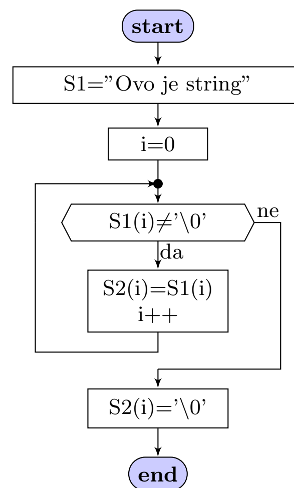
```
1  main()
2  {
3      char S1[] = "Ovo␣je␣string";
4      char S2[20];
5      int i = 0;
6      while (S1[i] != '\0')
7      {
8          S2[i] = S1[i];
9          i++;
10     }
11     S2[i] = '\0';
12 }
```

Dijagram toka algoritma ovog programa prikazan je na slici 5.23.

Neophodno je na kraju u string $S2$ upisati i "nal" karakter, jer je dati algoritam takav da zbog uslova i tipa petlje ovo neće uraditi. Ukoliko se ne upiše kraj stringa program bi prikazivao karaktere ili do kraja ili do prve nule koja se slučajno zatekla u nekom elementu. Da je umesto *while* petlje algoritam rešen *do-while* petljom, ne bi bilo potrebe upisivati simbol za kraj, već bi bio "prekopiran" u samoj petlji.

Poređenje

Poređenje stringova je operacija kojom se proverava da li su sadržaji dva stinga jednaki, a eventualno ako nisu, i koji string je leksikografski ispred koga. Pod jednakošću se podrazumeva da su na odgovarajućim mestima u stringu upisani isti karakteri.



Slika 5.23: Dijagram toka algoritma za kopiranje sadržaja jednog stringa u drugi

U C-u je sintaksno ispravno napisati sledeće poređenje:

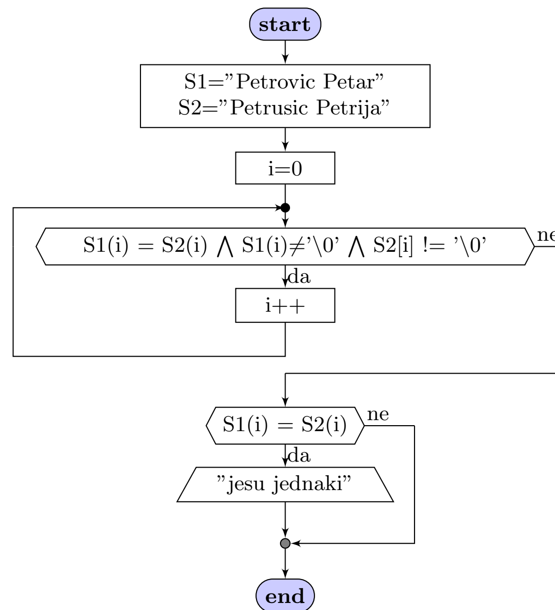
```
1  main()
2  {
3      char S1[] = "Algoritmi␣i␣programiranje";
4      char S2[] = "Algoritmi␣i␣programiranje";
5      if (S2 == S1)
6          printf("jesu␣jednaki");
7  }
```

Bez obzira što su sadržaji stringova $S1$ i $S2$ iz prethodnog primera očigledno jednaki, poređenje u 5. liniji koda poredi pokazivače, odnosno adrese početka nizova, a ne sadržaj. S obzirom na to da se radi o različitim stringovima, ove adrese su uvek različite (vidi sliku 5.22). Zbog ovoga će ishod poređenja iz prethodnog programa uvek biti negativan.

Da bi se poredili stringovi potrebno je proveriti jednakost svih karaktera na odgovarajućim pozicijama od početka, pa do oznake za kraj stringa. Moguće je sastaviti više različitih algoritama koji obavljaju ovaj zadatak, a jedno moguće rešenje prikazano je dijagramom toka algoritma na slici 5.24.

Program napisan na osnovu algoritma čiji je dijagram toka prikazan na slici 5.24 dat je u nastavku.

```
1  main()
2  {
```



Slika 5.24: Dijagram toka algoritma za poređenje stringova

```
3       char S1[] = "Petrovic Petar";
4       char S2[] = "Petrusic Petrija";
5       int i = 0;
6       while(S1[i] == S2[i] && S1[i] != '\0' && S2[i] != '\0')
7           i++;
8       if (S1[i] == S2[i])
9           printf("jesu jednaki");
10  }
```

Brojač i u *while* petlji povećava se za 1, čime se prelazi na naredni simbol, pod uslovom da su odgovarajući simboli u stringovima jednaki ($S1[i] = S2[i]$) i da se nije došlo do kraja prvog ($S1[i]! =' \backslash 0'$), ni drugog stringa ($S1[i]! =' \backslash 0'$).

Kako petlja ima tri uslova, nakon izlaska iz petlje ne zna se zbog čega se iz petlje izašlo: da li zato što su karakteri na koje se naišlo različiti, ili se došlo do kraja nekog od stringova. Jednostavnim ispitivanjem nekog od uslova petlje može se zaključiti razlog izlaska, što je i učinjeno u 8. liniji koda.

Uslov u 8. liniji programa moguće je proširiti tako da se zna i leksikografski redosled sadržaja stringova[8]. Ukoliko se iz petlje izašlo zato što su pronađeni ra-

[8]Određivanje leksikografskog redosleda sadržaja stringova je operacija koja se koristi kod sortiranja spiskova imena i prezimena po prezimenu.

zličiti simboli na odgovarajućim pozicijama, međusobni odnos tih simbola u ASCII tabeli odgovara leksikografskom odnosu stringova. Ovo proširenje dato je sledećim kodom.

```
1     if (S1[i] == S2[i])
2        printf("jesu jednaki");
3     else
4        if (S1[i] < S2[i])
5           printf("S1 je leksikografski ispred S2");
6        else
7           printf("S2 je leksikografski ispred S1");
```

S obzirom na to da su kodovi za odgovarajuća mala i velika slova u ASCII tabeli različiti, ishod poređenja dva stringa od kojih jedan počinje malim a drugi velikim slovom, bio bi taj da su stringovi različiti. Na primer, string "jezik C", se razlikuje od stringa "jezik c".

Ukoliko se želi izbeći ovaj problem, obično se pre poređenja sva slova oba stringa prevode u mala ili velika slova.

Napomena: Proveru da li je slovo zapamćeno u promenljivoj c veliko moguće je izvršiti ispitivanjem uslova $c \geq$'A'$\bigwedge c \leq$'Z', što daje informaciju da li se simbol zapamćen u promenljivoj c nalazi u ASCII tabeli između slova 'A' i 'Z'. Ukoliko je u promenljivoj c veliko slovo, moguće ga je prevesti u malo dodavanjem razlike koja postoji između velikih i malih slova u ASCII tabeli, i koja je konstantna za sva slova (slika 4.3): c+('a'-'A').

Konkatenacija

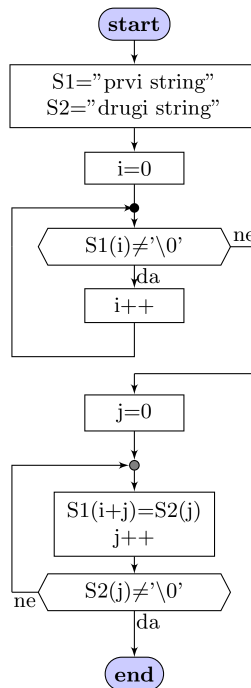
Konkatenacija je operacija kojom se sadržaj jednog stringa nadovezuje na drugi string.

Algoritam operacije konkatenacije ima tri faze:

1. određivanje kraja prvog stringa,
2. uklanjanje simbola za kraj prvog stringa, i
3. kopiranje svih karaktera iz drugog stringa na odgovarajuće pozicije u prvi string.

Pretpostavimo da su stringovi deklarisani tako da u prvom stringu ima dovoljno mesta za nadovezivanje drugog. Dijagram toka jednog mogućeg rešenja algoritma kojim se sadržaj stringa S2 nadovezuje na sadržaj stringa S1 prikazan je na slici 5.25. Kod ovog rešenja prva faza određivanja dužine stringa $S1$ implementirana je *while* petljom, a druge dve faze u kojima se uklanja kraj prvog stringa i nadovezuje sadržaj drugog stringa implementirane su *do-while* petljom. Program napisan na osnovu dijagrama toka sa slike 5.25 dat je u nastavku.

```
1 main()
2 {
```



Slika 5.25: Dijagram toka algoritma za konkatenaciju stringova

```
3       char S1[80] = "prvi string";
4       char S2[80] = "drugi string";
5           // Faza 1: Naci kraj sadrzaja S1
6       int i = 0;
7       while(S1[i] != '\0')
8           i++;
9           // Faza 2 i Faza 3: uklanjanje '\0' iz S1
10          // i kopiranje S2 u S1 ukljucijuci i novi kraj '\0'
11          int j = 0;
12          do
13          {
14                  S1[i+j] = S2[j];
15                  j++;
16          }
17      while(S2[j] != '\0');
18  }
```

5.3.5 Nizovi stringova

Kako je string sam po sebi niz karaktera, niz stringova je matrica karaktera.

Niz stringova se koristi kada je potrebno zapamtiti više stringova tako da je pojedinačne stringove moguće indeksirati. Razlog za indeksiranje je pojednostavljivanje pristupa svim elementima redom radi vršenja neke obrade. Ovo je jednostavno implementirati iz tela petlje, tako da svaka iteracija petlje obrađuje po jedan string.

Na primer, niz od 20 stringova, gde je svaki string maksimalne dužine 80 karaktera, moguće je zapamtiti u matrici:

```
char M[20][80];
```

Interesantno je u ovom slučaju iskoristiti činjenicu da se kod programskog jezika C, ukoliko se navede samo prvi indeks matrice, dobija pokazivač na niz (poglavlje 5.2.7, slika 5.20). Tako, `M[0]` iz prethodnog primera je pokazivač na prvu vrstu matrice, odnosno prvi string. Drugom stringu se može pristupiti sa `M[1]`, itd. Pristup `M[0][0]` daće prvi karakter prvog stringa.

Sledeći program ilustruje upotrebu niza stringova.

Primer 5.9 (Niz stringova)**. Zadatak**: Napisati program na programskom jeziku C koji od korisnika zahteva unos 10 prezimena, a zatim određuje koliko prezimena počinje slovom 'M' i prikazuje ih.

Rešenje: Rešenje zadatka dato je u nastavku.

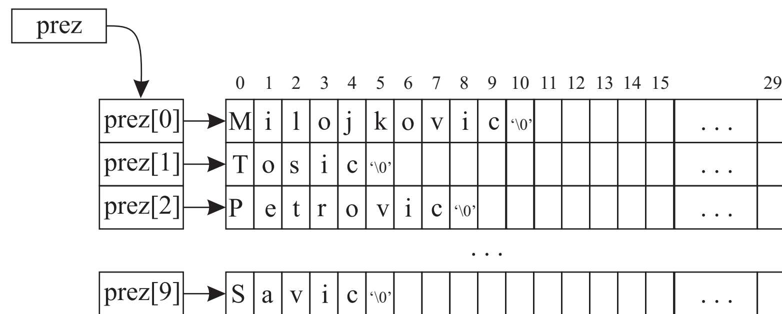
```
1  #include "stdio.h"
2  #include "string.h"
3  main ()
4  {
5      char prez[10][30];  // 10 prezimena, max. duzine 30
6      // Unos
7      int i, br=0;
8      for (i = 0; i < 10; i++)
9          scanf("%s", prez[i] ); // i-to prezime
10     // Prebrojavanje i prikaz
11     printf("Prezimena␣koja␣pocinju␣na␣M␣su:␣");
12     for (i = 0; i < 10; i++)
13         if ( prez[i][0] == 'M' )         // prvo slovo je M?
14         {
15             printf("%s,␣", prez[i]);   // prikazi i-to prez.
16             br ++;                      // povecaj brojac
17         }
18     printf("i␣ima␣ih␣%d.", br);
19 }
```

Izgled konzole prilikom izvršenja programa je:

```
Milojkovic
Tosic
```

```
Petrovic
Stanisavljevic
Vojinovic
Simic
Ilic
Milosavljevic
Mihajlovic
Savic
Prezimena koja pocinju na M su: Milojkovic,
Milosavljevic, Mihajlovic, i ima ih 3.
```

Potrebno je napomenuti da se prilikom unosa svakog pojedinačnog stringa funkciji *scanf* iz 9. linije programa prosleđuje po jedan string. Ova funkcija uneti string upisuje u odgovarajuću vrstu matrice i na kraj stringa obavezno dodaje oznaku za kraj stringa '\0'. Ovo je prikazano na slici 5.26. △



Slika 5.26: Izgled niza stringova iz primera 5.9

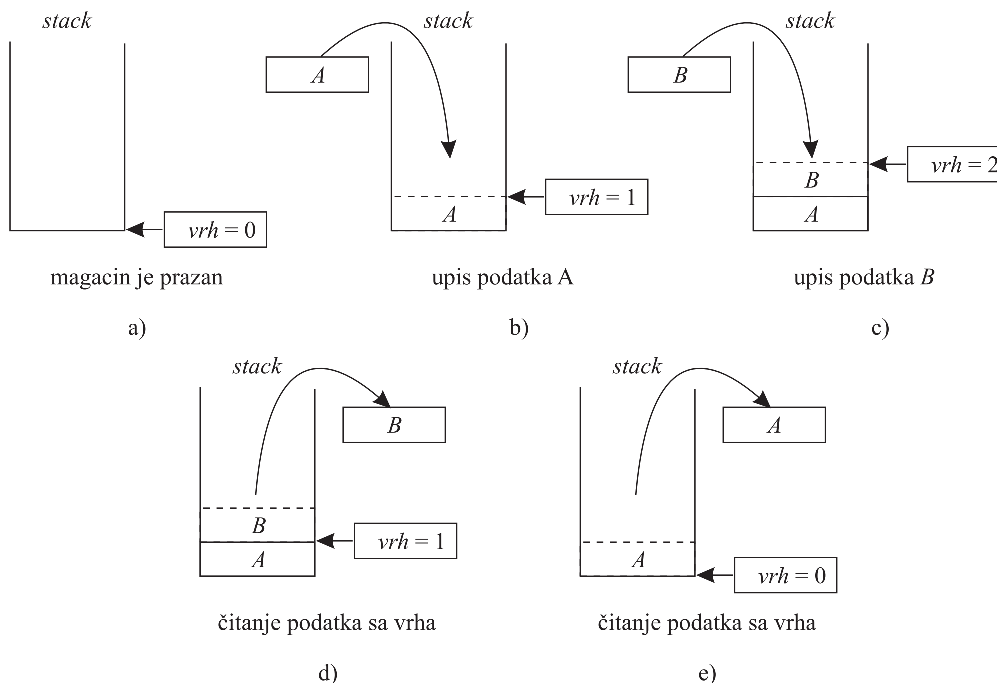
5.4 Magacin i red

Magacin (eng. *stack*, čita se "stek") i red (eng. *queue*, čita se "kju") su strukture podataka koje se u memoriji smeštaju kao jednodimenzionalna polja. Za razliku od polja, kod ovih se struktura ne može direktno pristupiti bilo kom elementu strukture, već postoje specifične operacije za rad sa ovim strukturama.

5.4.1 Magacin

Magacin je dobio naziv po specifičnom načinu upisa i čitanja podataka. Iz ugla korisnika magacina (programera), u magacin se mogu upisivati podaci redom, jedan po jedan, ali se pri čitanju podaci "vade" iz magacina tako što se iz magacina prvo čita i uklanja podatak koji je poslednji upisan. Narednim čitanjem uzima se vrednost narednog podatka i on se takođe uklanja iz magacina, itd.

Strukturu podataka magacina čine linearno jednodimenzionalno polje i jedan celobrojni podatak koji sadrži informaciju o trenutnom broju podataka u magacinu. Ova struktura prikazana je na slici 5.27. Na slici 5.27 prikazan je postupak upisa dva podatka, A i B, u magacin sa nazivom *stack*. Promenljiva *vrh* označava vrh magacina, što je ujedno i informacija o broju elemenata u magacinu. Obično se kaže da se podatak koji se upisuje smešta "na vrh" magacina, a da se podatak koji se čita, čita "sa vrha" magacina. Iz magacina nije moguće pročitati proizvoljan podatak, već se operacija čitanja odnosi isključivo na poslednji upisani podatak, koji se ujedno i briše iz magacina.



Slika 5.27: Koncept upisa podataka u magacin i čitanja podataka iz magacina

Na slici 5.27a prikazan je prazan magacin. Upis podatka A ilustrovan je na slici 5.27b. Podatak A upisan je na vrh magacina, čime je promenljiva koja ukazuje na vrh postala $vrh = 1$. Na ovaj način, promenljiva vrh predstavlja indeks niza gde je potrebno upisati naredni podatak, ali je ujedno i trenutni broj popunjenih mesta u magacinu. Na identičan način izvršen je upis promenljive B, što je prikazano na slici 5.27c.

Čitanje sa vrha magacina prikazano je na slici 5.27d. Na vrhu magacina u tom trenutku našao se podatak B, pa će on biti pročitan i uklonjen ovom operacijom. Na slici 5.27e dat je naredni korak čitanja podatka iz magacina.

Jedine dve operacije koje su dozvoljene nad ovom strukturom su:

1. upis na vrh magacina, i
2. čitanje sa vrha magacina.

Neka je M identifikator niza koji implementira magacin, a N maksimalni dozvoljeni broj elemenata u magacinu, tada je pseudokod operacije upisa podatka A u magacin:

if $vrh < N$ **then**
  $M(vrh) = A$;
  $vrh = vrh + 1$;
end

Drugim rečima, ako magacin nije pun ($vrh < N$), podatak se upisuje i vrh se pomera za jedno mesto. Operacija čitanja podatka sa vrha magacina se u pseudo kodu može predstaviti kao:

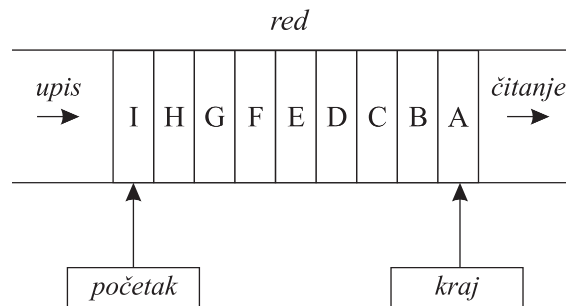
if $vrh > 0$ **then**
  $vrh = vrh - 1$;
  čitanje $M(vrh)$;
end

Kaže se da je magacin struktura tipa LIFO, od engleskih reči *Last In First Out*, što znači poslednji upisani element se čita prvi.

5.4.2 Red

Za razliku od magacina, koji je LIFO struktura, red je linearna struktura tipa FIFO - *First In First Out*, što znači da će prvi podatak koji će biti pročitan biti onaj podatak koji je prvi i upisan, odnosno onaj koji najduže "čeka" u redu.

I red, kao i magacin, u osnovi čini linearno jednodimenzionalno polje. Operacije čitanja i upisa se razlikuju po poziciji na kojima se upisuju i čitaju podaci. Naime, kod reda se podaci uvek upisuju na početak reda, a čitaju sa kraja reda. Ovo je ilustrovano na slici 5.28.



Slika 5.28: Koncept upisa podataka u red i čitanja podataka iz reda

Pored jednodimenzionalnog niza, za pamćenje ove strukture neophodne su još dve promenljive za pamćenje pozicije početka i pozicije kraja. I pri upisu na početak

liste, kao i pri čitanju sa kraja liste, odgovarajuće promenljive se umanjuju za 1 (ili povećavaju za 1, zavisno od ugla gledanja, odnosno implementacije).

Zbog konačne veličine niza kojim se implementira ova struktura, pri smanjivanju (odnosno povećavanju) promenljivih koje označavaju početak i kraj reda, moraju se ispitivati uslovi o položaju ovih promenljivih u odnosu na granice niza. Ove operacije su van okvira ovog udžbenika i ovde neće biti razmatrane.

Ova struktura takođe nalazi veliku primenu u programiranju i računarstvu uopšte. Na primer, svakom priključku (*portu*) mrežnih uređaja pridružuje se po jedan red čekanja za smeštaj mrežnih paketa.

5.5 Nelinearne strukture

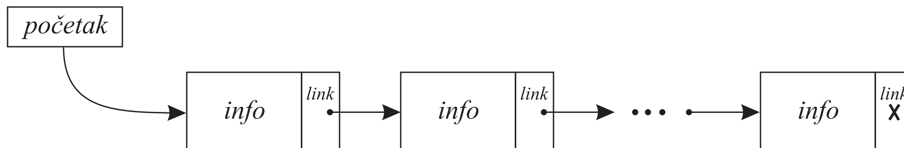
Lančane liste, stabla i grafovi su strukture kod kojih se elementi, za razliku od polja, magacina i redova, obično ne smeštaju u sukcesivne memorijske lokacije. Lančane liste, bez obzira na ovu osobinu, spadaju u grupu linearnih struktura, dok su stabla i grafovi nelinearne strukture.

5.5.1 Lančane liste

Definicija 5.7 (Lančane liste)**.** Lančana lista je linearna struktura podataka kod koje podaci nisu smešteni u sukcesivne memorijske lokacije, i kod koje svaki element, sem prvog i poslednjeg, ima tačno dva suseda.

Lančane liste se najčešće implementiraju samoredefencirajućim strukturama, na sličan način kako je implementirana ulančana struktura u primeru 4.29 na 205. strani. Elementi liste nazivaju se čvorovima liste. Svaki čvor ima informacioni deo i deo koji pokazuje na naredni elemenat liste.

Pored čvorova liste, uz listu se obavezno pamti i jedna promenljiva pokazivačkog tipa, koja ukazuje na prvi elemenat liste. Ova struktura je prikazana na slici 5.29.



Slika 5.29: Lančana lista

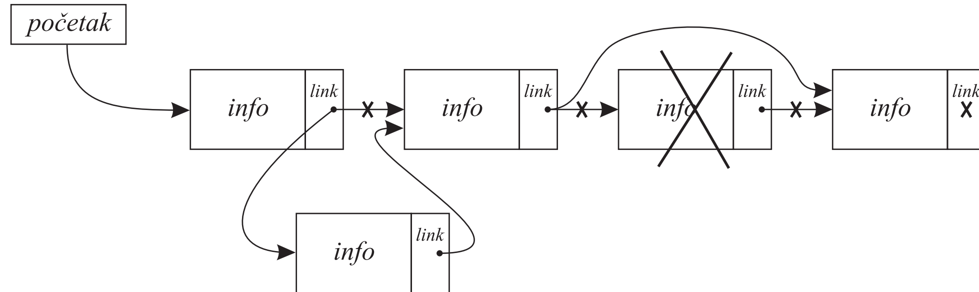
Čvorovi liste sa slike 5.29, kao što je rečeno, ne moraju biti u sukcesivnim memorijskim lokacijama. Baš zbog ove osobine se u svakom čvoru liste pamti pokazivač na naredni čvor, kako bi kretanje kroz listu bilo moguće (obilazak liste od početka do kraja).

Postoji nekoliko tipova lančanih listi, koje se međusobno razlikuju po broju i smeru veza između suseda. Lista sa slike 5.29 je jednostruko spregnuta lista.

Najvažnije operacije nad listama su:

- obilazak liste,
- dodavanje čvorova, i
- brisanje čvorova.

Koncept dodavanja i brisanja čvorova prikzan je na slici 5.30.



Slika 5.30: Koncept dodavanja čvorova u listu i brisanja čvorova liste

Ova struktura takođe nalazi veliku primenu u programiranju i računarstvu uopšte. Na principu koji je prikazan na slici 5.29 fizički se pamte delovi jednog fajla na disku, jer nije uvek slučaj da je pronađeni slobodan prostor na disku dovoljan da se upiše ceo fajl, već često treba "povezati" puno manjih praznih prostora u veću celinu. Ovaj problem se naziva problemom fragmentacije prostora i predmet je operativnih sistema.

5.5.2 Stabla i grafovi

Definicija 5.8 (Stabla)**.** Stablo je nelinearna struktura podataka kod koje podaci nisu u sukcesivnim memorijskim lokacijama, i kod koje svaki čvor stabla ima tačno jednog prethodnika, sem jednog specijalnog čvora koji se naziva korenom stabla, i može imati proizvoljan broj sledbenika.

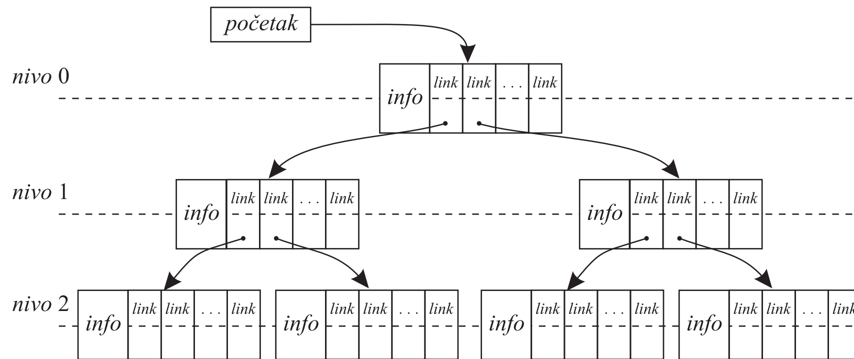
Struktura stabla prikazana je na slici 5.31.

Stablo se takođe, kao i lančane liste, najčešće implementira samoreferencirajućim strukturama. Svako stablo ima koren stabla, na koji ukazuje pokazivačka promenljiva koja označava početak stabla (slika 5.31). Logički posmatrano, čvorovi stabla se dele u nivoe. Koren stabla je jedini čvor na nultom nivou. Čvorovi poslednjeg nivoa nazivaju se listovima stabla.

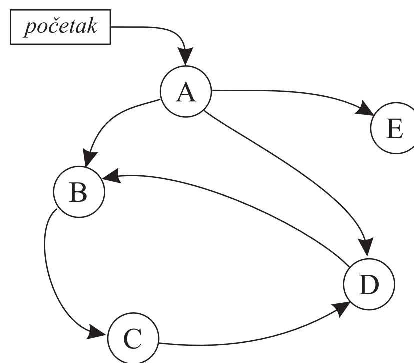
Definicija 5.9 (Grafovi)**.** Graf u programiranju predstavlja nelinearnu strukturu podataka kod koje podaci nisu u sukcesivnim memorijskim lokacijama, i kod koje svaki čvor grafa može imati proizvoljan broj prethodnika i sledbenika.

Primer grafa prikazan je na slici 5.32.

Za pamćenje strukture sa slike 5.32 mogu se koristiti samoreferencirajuće strukture, ali se neretko koriste i matrice. Kod matrične reprezentacije svakom čvoru



Slika 5.31: Struktura stabla



Slika 5.32: Struktura grafa

odgovara po jedna vrsta i po jedna kolona, odnosno čvoru i odgovara i-ta kolona i i-ta vrsta, a na postojanje veze (grane) između čvorova i i j obično ukazuje vrednost elementa $a_{i,j} = 1$. Vrednost $a_{i,j} = 0$ sugeriše da su čvorovi nepovezani.

Matrična reprezentacija grafa sa slike 5.32 prikazana je na slici 5.33. Na slici je posebno naglašena vrednost 1 u preseku vrste koja odgovara čvoru B i kolone koja odgovara čvoru C, što predstavlja postojanje grane između ova dva čvora.

$G =$

	A	B	C	D	E
A	0	1	0	1	1
B	0	0	1	0	0
C	0	0	0	1	0
D	0	1	0	0	0
E	0	0	0	0	0

Slika 5.33: Matrična reprezentacija grafa

Kontrolna pitanja

1. Da li je sledeća deklaracija polja A sintaksno ispravna? Obrazložiti odgovor.

```
1  main()
2  {
3      int N = 5;
4      int A[N];
5      // ...
6  }
```

2. Objasniti i na proizvoljnom primeru ilustrovati inicijalizaciju prilikom deklaracije jednodimenzionalnog i dvodimenzionalnog polja. Koje se dimenzije mogu izostaviti kada se polje inicijalizuje prilikom deklaracije?

3. Napisati program kojim se deklariše niz A sa 100 celobrojnih elemenata i u prvi element upisuje vrednost 1, a u poslednji vrednost 100.

4. Nacrtati strukturni dijagram toka algoritma i na programskom jeziku C napisati strukturni program koji elemente niza inicijalizovanog prilikom deklaracije prikazuje u jednom redu, međusobno odvojene zarezom. Voditi računa da se iza poslednjeg elementa niza ne prikazuje zarez.

5. Nacrtati strukturni dijagram toka algoritma i na programskom jeziku C napisati strukturni program kojim se sa tastature zadaje broj elemenata niza, a za njim redom i vrednosti elemenata niza. Nakon unosa niz prikazati u obrnutom redosledu.

6. Nacrtati strukturni dijagram toka algoritma i na programskom jeziku C napisati strukturni program koji vrši rotaciju niza A sa N elemenata za jedno mesto udesno, tako da vrednost elementa a_0 prelazi u a_1, a_1 u a_2, itd, a poslednji element a_{N-1} prelazi na prvu poziciju a_0.

7. Nacrtati strukturni dijagram toka algoritma i na programskom jeziku C napisati strukturni program koji iz zadatog celobrojnog niza izbacuje element sa k-te pozicije. Sve elemente sa desne strane izbačenog elementa pomeriti za jedno mesto ulevo, kako bi se popunio "prazan" prostor.

8. Nacrtati strukturni dijagram toka algoritma i na programskom jeziku C napisati strukturni program koji vrši sortiranje niza A sa N elemenata u rastući redosled metodom selekcije. Broj elemenata niza i elemente niza zadaje korisnik na početku programa.

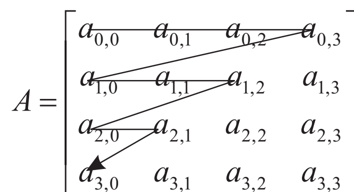
9. Nacrtati strukturni dijagram toka algoritma koji od matrice $A_{N\mathrm{x}M}$, čije dimenzije i elemente zadaje korisnik, formira i prikazuje transponovanu matricu $A^T_{M\mathrm{x}N}$. Za elemente transponovane matrice važi da je:

$$a^T_{i,j} = a_{j,i}.$$

10. Nacrtati strukturni dijagram toka algoritma za obilazak elemenata:
    (a) i-te vrste matrice,
    (b) j-te kolone matrice,
    (c) iznad glavne dijagonale,
    (d) ispod sporedne dijagonale.

11. Nacrtati strukturni dijagram toka algoritma koji sumira vrednosti elemenata iznad sporedne dijagonale matrice $A_{N\mathrm{x}N}$. Obilazak matrice obaviti pomoću dve *for* petlje, gde granice unutrašnje petlje zavise od indeksa spoljašnje petlje, i na taj način izbeći potrebu za proverom uslova da li se na osnovu indeksa posmatrani element nalazi iznad sporedne dijagonale (slika 5.18). Primer obilaska prikazan je na slici 5.34, a algoritmom treba direktno pristupiti elementima u sledećem redosledu:

$$a_{0,0}, a_{0,1}, a_{0,2}, a_{0,3}, a_{0,0}, a_{1,0}, a_{1,1}, a_{1,2}, a_{2,0}, a_{2,1}, a_{3,0}.$$



Slika 5.34: Primer obilaska matrice 4x4

12. Nacrtati strukturni dijagram toka algoritma i na programskom jeziku C napisati strukturni program koji određuje i prikazuje koliko je malih, a koliko velikih slova u zadatom stringu.

13. Nacrtati strukturni dijagram toka algoritma i na programskom jeziku C napisati strukturni program koji određuje i prikazuje koliko ukupno reči sadrži tekst koji zadaje korisnik.

14. Nacrtati strukturni dijagram toka algoritma i na programskom jeziku C napisati strukturni program koji određuje koji je od dva stringa zadata od strane korisnika duži.

15. Nacrtati strukturni dijagram toka algoritma i na programskom jeziku C napisati strukturni program koji sva velika slova u stringu koji zadaje korisnik prevodi u mala.

16. Nacrtati strukturni dijagram toka algoritma i na programskom jeziku C napisati strukturni program koji ispituje da li je uneti string palindrom. String je palindrom ukoliko se isto čita i sleva udesno i zdesna ulevo. Prikazati odgovarajuću poruku.

17. Nacrtati strukturni dijagram toka algoritma i na programskom jeziku C napisati strukturni program koji pronalazi prvu poziciju na kojoj se string *str1* javlja kao podstring u stringu *str2*. Stringove zadaje korisnik na početku programa.

18. Nacrtati strukturni dijagram toka algoritma i na programskom jeziku C napisati strukturni program koji u zadatom stringu određuje redni broj najduže reči. Reči u rečenici su razdvojene blanko znacima (jednim ili više). Prikazati redni broj najduže reči.

19. Nacrtati strukturni dijagram toka algoritma i na programskom jeziku C napisati strukturni program koji određuje najduži string iz zadatog niza stringova.

20. Neka je magacin predstavljen nizom A u kome se može zapamtiti maksimalno 100 celih brojeva, i neka je niz inicijalizovan prilikom deklaracije sledećim vrednostima:
$$\{4, 5, 3, 7, 8\}.$$
Inicijalna vrednost promenljive vrh je 5. Napisati program na C-u kojim se po principu rada magacina u ovaj niz upisuju nove vrednosti 2 i 9, a nakon toga iz magacina čita i prikazuje ukupno 5 podataka. Koji će podaci i u kom redosledu biti prikazani?

6

Funkcije

Funkcija je izdvojena programska celina (potprogram), čiji je zadatak da na osnovu određenog algoritma transformiše ulazne podatke u novi podatak. Ulazni podaci funkcije nazivaju se **parametri funkcije**, a izlazni podatak se naziva **rezultat funkcije**.

Funkcije u programiranju dobile su naziv na osnovu koncepta matematičkih funkcija. Funkcija se u matematici može predstaviti kao:

$$y = f\left(x_1, x_2, \ldots, x_n\right),$$

gde je sa f označena funkcija, sa x_1 do x_N parametri, a sa y rezultat. Funkcija ima više ulaznih parametara i jedan izlazni rezultat.

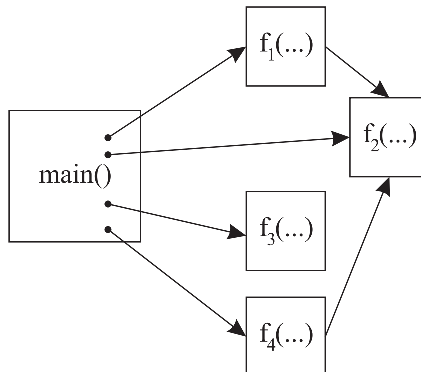
U programiranju se pored funkcija koristi i koncept *procedura*. Uobičajeno je u programskim jezicima da se funkcijom naziva potprogram koji vraća jedan rezultat, a da se potprogram koji može vratiti više rezultata naziva procedurom. Mnogi programski jezici imaju i drugačije sintaksne konstrukcije za definisanje ova dva tipa potprograma. Dakle, procedura je potprogram koji može imati više ulaznih parametara, ali za razliku od funkcija može imati i više izlaznih rezultata. U C-u procedure kao poseban koncept ne postoje. Za implementaciju procedura u C-u koriste se funkcije, sa specifičnim načinom prenosa parametara, o kom će biti reči u kasnijim poglavljima.

U zavisnosti od toga ko implementira funkciju, funkcija može biti korisnička ili standardna (bibliotečka). Korisničke funkcije piše sam programer, dok se standardne funkcije isporučuju uz kompajler.

Pomoću funkcija se može izvršiti dekompozicija problema i sam problem rešiti jednostavnije. Dovoljno je da programer pozove funkciju na izvršenje i preda joj parametre, nakon čega će dobiti očekivani rezultat. U slučaju standardnih funkcija nije neophodno da programer zna po kom algoritmu funkcija radi da bi mogao da koristi funkciju. Kaže se da funkcija sakriva (*enkapsulira*, *apstrahuje*) sam postupak kojim ostvaruje svoju funkcionalnost.

Funkcije se mogu pozvati na izvršenje iz glavnog programa, ili iz druge funkcije. Primer dekompozicije programa koji sadrži funkcije prikazan je na slici 6.1. U pri-

meru poziva prikazanom na slici 6.1, glavni program $main()$ koristi funkcionalnosti funkcija f_1, f_2, f_3 i f_4, a funkcije f_1 i f_4 koriste funkcionalnosti funkcije f_2.



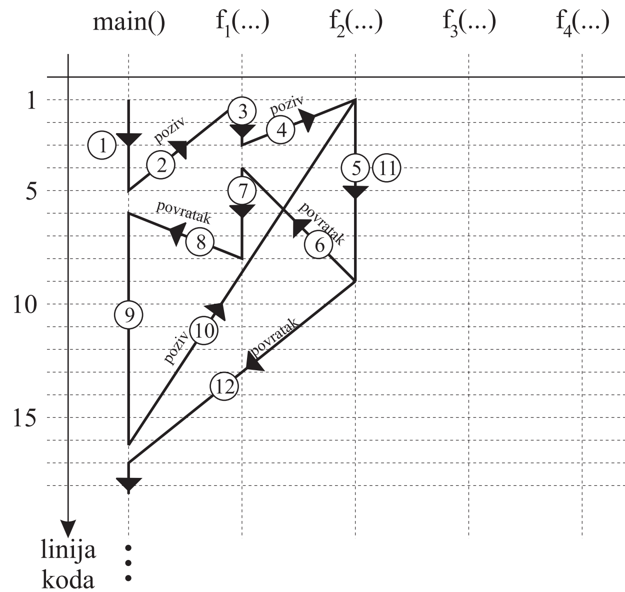
Slika 6.1: Dekompozicija problema pomoću funkcija

Kada glavni program ili druga funkcija pozove neku funkciju na izvršenje, pozivajuća funkcija se prekida i procesor se dodeljuje pozvanoj funkciji sve dok se ta funkcija ne završi. Tek nakon završetka pozvane funkcije pozivajuća funkcija nastavlja sa izvršenjem od naredbe koja se nalazi neposredno nakon poziva funkcije. Primer vremenskog dijagrama izvršenja programa sa više funkcija prikazan je na slici 6.2. Glavni program $main()$ u primeru sa slike 6.2 u 5. liniji koda poziva funkciju f_1, koja u svojoj 3. liniji poziva funkciju f_2, itd.

6.1 Sintaksa funkcija u C-u

Za implementaciju korisničke funkcije, potrebno je definisati funkciju. Definicija funkcije je deo koda koji sadrži:

1. naziv funkcije,
2. opis ulaznih parametara i rezultata, i
3. telo funkcije.



Slika 6.2: Vremenski dijagram izvršenja programa sa više funkcija

Definicija i parametri funkcije

Definicija funkcije u ANSI C-u i novijim kompajlerima se razlikuje. Sintaksa definicije funkcije u ANSI C-u je sledeća:

⟨*definicija_fje*⟩ ::=
    ⟨*tip*⟩ ⟨*ime*⟩ ([⟨*lista_parametara*⟩])
    [⟨*definicija_parametara*⟩]
    {
        ⟨*telo_funkcije*⟩
    }

⟨*tip*⟩ ::= **char** | **int** | ··· | **void**
⟨*ime*⟩ ::= ⟨*identifikator*⟩
⟨*lista_parametara*⟩ ::= ⟨*prom._1*⟩ [; ⟨*prom._2*⟩ ...]
⟨*definicija_parametara*⟩ ::= ⟨*tip_1*⟩ ⟨*prom._1*⟩ [, ⟨*tip_1*⟩ ⟨*prom._2*⟩ ...]

Na početku deklaracije nalazi se ⟨*tip*⟩ koji označava tip vrednosti koju funkcija određuje i vraća. Tip je obavezan, a ako funkcija ne treba da vrati vrednost, za tip se navodi **void**. Iza tipa navodi se identifikator imena funkcije, za kojim u malim zagradama stoji lista parametara. Ovi parametri se nazivaju i fiktivni parametri.

Definicija 6.1 (Fiktivni parametri funkcije)**.** Fiktivni parametri funkcije su simbolička imena za vrednosti koje se funkciji prosleđuju prilikom poziva.

U fiktivne parametre upisuju se vrednosti prilikom poziva, i oni postoje u memoriji samo za vreme izvršavanje funkcije.

Lista parametara je jednostavno lista identifikatora imena parametara razdvojenih simbolom ';'. Za listom parametara navodi se i definicija parametara, gde se svakom imenu parametra pridružuje i tip. Telo funkcije čine naredbe C-a koje se izvršavaju kada se funkcija pozove.

Sintaksa definicije funkcije u C99 i novijim kompajlerima razlikuje se u odnosu na sintaksu ANSI C-a po tome što se umesto liste parametara u okviru malih zagrada direktno navodi definicija parametara:

$$\langle definicija_fje \rangle \quad ::= \quad \langle tip \rangle \, \langle ime \rangle \; (\; [\; \langle definicija_parametara \rangle \;] \;) \; \{ \; \langle telo_funkcije \rangle \; \}$$

$$\langle definicija_parametara \rangle \quad ::= \quad \langle tip_1 \rangle \, \langle prom._1 \rangle \, [, \langle tip_1 \rangle \, \langle prom._2 \rangle \ldots]$$

Telo funkcije i povratna vrednost

Telo funkcije, samo po sebi, ima istu strukturu kao i telo glavnog programa. Za pisanje naredbi u telu funkcije važe sva pravila kao i za glavni program.

Funkcija može imati lokalne promenljive koje su vidljive i mogu se koristiti samo iz funkcije, i postoje u memoriji samo za vreme izvršavanja funkcije.

Što se strukture tiče, ukoliko se radi o ANSI C-u, deklaracija lokalnih promenljivih funkcije nalazi se na samom početku tela funkcije, pre prve izvršne naredbe funkcije, kao što je slučaj i sa glavnim programom. Kod novijih kompajlera deklaracija se može naći bilo gde u telu funkcije, uz ograničenje da promenljiva postoji u memoriji i može se koristiti počev od naredbe gde je deklarisana, pa nadalje, ali ne i u naredbama koje se nalaze pre deklaracije.

Funkcija se završava i upravljanje se vraća pozivajućem programu nakon poslednje naredbe funkcije. U C-u postoji i naredba za eksplicitni završetak funkcije - **return**.

Naredba *return* vraća upravljanje i eventualno može vratiti rezultat funkcije pozivajućem programu. Ovom naredbom se može vratiti vrednost nekog od osnovnih ili izvedenih tipova. Statički deklarisana polja u okviru funkcije nije moguće direktno vratiti naredbom *return*. Sintaksa poziva naredbe return je:

$$\langle poziv_return \rangle \quad ::= \quad \textbf{return}[\langle promenljiva \rangle \mid \langle konstanta \rangle \mid \langle pokazivac \rangle \mid \langle izraz \rangle \mid \langle struktura \rangle];$$

Poziv naredbe *return* se može naći bilo gde u funkciji, ali ukoliko iza naredbe *return* ima drugih naredbi, nijedna od ovih naredbi se neće izvršiti.

Napomena: Tip funkcije mora biti isti kao i tip vrednosti koju vraća naredba *return*. Ukoliko je naredba *return* navedena bez parametara, tada tip funkcije mora biti *void*.

Poziv funkcije

Napisana funkcija će se izvršiti tek kada se pozove iz glavnog programa ili neke druge funkcije. Sintaksa poziva funkcije je ista kod svih verzija kompajlera, i u EBNF se može predstaviti na sledeći način:

$$\langle poziv_funkcije\rangle \quad ::= \quad [\langle promenljiva\rangle =]\,\langle ime\rangle\,([\langle stvarni_parametri\rangle])\,;$$

Prilikom poziva funkcije na izvršenje navodi se ime i lista **stvarnih parametara** u malim zagradama, a povratna vrednost se opciono može upisati u promenljivu. Stvarni parametri su promenljive, konstante, izrazi ili povratne vrednosti iz drugih funkcija. Generalno, stvarni parametri su konkretne vrednosti koje se u trenutku poziva funkcije prenose funkciji.

Napomena: Broj stvarnih parametara i njihov tip mora odgovarati po redosledu i broju fiktivnih parametara.

Prilikom poziva funkcije, za svaki fiktivni parametar kreira se posebna promenljiva i u nju se upisuje vrednost odgovarajućeg stvarnog parametra.

Prema sintaksi definicije funkcije, funkcija može biti i bez parametara. Ukoliko funkcija nema parametre, tada se prilikom poziva funkcije male zagrade obavezno navode, ali se stvarni parametri ne navode, kao što je i naznačeno uglastim zagradama u sintaksi poziva funkcije.

Slično važi i za povratnu vrednost, s tim da bez obzira da li funkcija vraća neku povratnu vrednost ili ne, na programeru je da li će ovu vrednost prilikom poziva funkcije sačuvati u neku promenljivu.

Primer 6.1 (Funkcija bez argumenata i povratne vrednosti)**.** Sledeći primer ilustruje definiciju i poziv funkcije. Funkciji se u ovom primeru ne prenose parametri i funkcija nema povratnu vrednost.

```
1 #include "stdio.h"
2 void logo()
3 {
4     printf("Univerzitet u Nisu\nElektronski fakultet");
5 }
6 main()
7 {
8     logo();
9 }
```

Ovaj program na izlazu daje:

```
Univerzitet u Nisu
Elektronski fakultet
```

△

Primer 6.2 (Funkcija za određivanje maksimuma dva broja)**. Zadatak**: Na programskom jeziku C napisati funkciju koja određuje i vraća veći od dva broja preneta preko parametara funkcije. U glavnom programu uneti dva broja i pozivom funkcije odrediti koji je od unetih brojeva veći.

Rešenje: Rešenje ovog zadatka u ANSI C-u dato je u nastavku.

```
1  #include <stdio.h>
2  int maksimum(a,b)
3          int a; int b;
4  {
5      int pom;
6      if (a > b)
7          pom = a;
8      else
9          pom = b;
10     return pom;
11 }
12 main()
13 {
14     int x,y,M;
15     scanf("%d%d",&x,&y);
16
17     M = maksimum(x,y);
18
19     printf("Broj␣%d␣je␣veci.",M);
20 }
```

Definicija funkcije nalazi se od 2. do 11. linije programa. Kako je u 2. liniji programa navedeno, funkcija vraća rezultat koji je tipa *int*. Funkcija je nazvana *maksimum*, a fiktivni parametri su a i b. Ove promenljive postoje samo za vreme izvršenja funkcije i u njih se upisuju odgovarajuće vrednosti stvarnih argumenata. Definicija parametara je u 3. liniji.

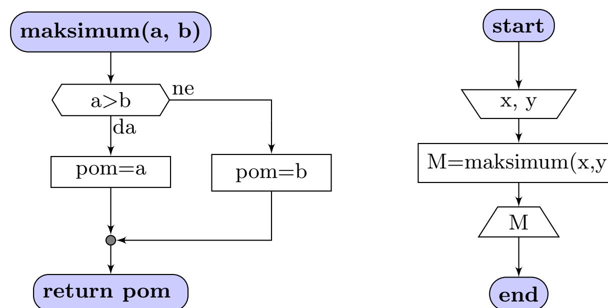
Ova funkcija ima jednu lokalnu promenljivu (*pom*), koja je deklarisana u 5. liniji koda. U 10. liniji se pozivom naredbe *return* eksplicitno završava funkcija i vraća vrednost koja se u tom trenutku nalazi u promenljivoj *pom*.

U glavnom programu se nakon unosa vrednosti poziva funkcija. Poziv funkcije sa stvarnim argumentima je u 17. liniji koda. Vrednost koju funkcija vrati nakon izvršenja upisuje se u promenljivu M. Izgled kozole prilikom izvršenja programa je u nastavku.

```
6 8
Broj 8 je veci.
```

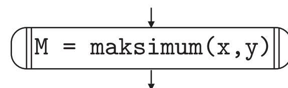
Dijagram toka algoritma programa datog u ovom primeru prikazan je na slici 6.3.

Uobičajeno je da se dijagram toka funkcije crta na isti način kao i dijagram toka glavnog programa. U bloku za početak funkcije navodi se ime i fiktivni parametri. S obzirom na to da su tipovi podataka stvar implementacije, a ne algoritma, nije neophodno navesti tip funkcije, kao ni tipove argumenata. Umesto bloka za kraj algoritma može se navesti naredba *return*. Poziv funkcije navodi se u bloku obrade, kao što je to prikazano na dijagramu toka glavnog programa na slici 6.3.



Slika 6.3: Dijagram toka algoritma sa funkcijom za određivanje maksimuma dva broja

U literaturi se za poziv funkcije sreće i poseban blok, prikazan na slici 6.4, u kome pre poziva funkcije može, ali i ne mora stajati engleska reč za poziv "*call*:".



Slika 6.4: Blok u dijagramu toka algoritma za poziv funkcije

△

Prototip funkcije

Definicija funkcije u C-u može se naći ispred ili iza glavnog programa $main()$. Kompajler u toku prevođenja programa čita program od početka do kraja. Ukoliko je definicija funkcije iza glavnog programa, a poziv te funkcije u glavnom programu, tada će kompajler javiti grešku da je naišao na poziv nepoznate funkcije. Generalno, greška će se javiti ukoliko se poziv konkretne funkcije, posmatrajući celu strukturu programa, nalazi pre definicije te funkcije.

Ukoliko je funkcija definisana pre poziva, ovaj problem se neće javiti. Ukoliko se poziv funkcije nalazi pre definicije, potrebno je na početku programa (pre poziva

funkcije) **deklarisati** funkciju. Deklaracija funkcije je sintaksno identična definiciji, s tim da se u deklaraciji ne navodi telo funkcije. Deklaracija funkcije se naziva i **prototip** funkcije. Deklaracijom se kompajleru stavlja na znanje da će kasnije biti navedena definicija funkcije, kako ne bi javljao grešku kada naiđe na poziv funkcije.

Prototip funkcije u ANSI C-u je:

$$\langle prototip \rangle \quad ::= \quad \langle tip \rangle \, \langle ime \rangle \, (\; [\; \langle lista_parametara \rangle \;] \;) \, ;$$

U novijim verzijama C kompajlera sintaksa prototipa funkcije je:

$$\langle prototip \rangle \quad ::= \quad \langle tip \rangle \, \langle ime \rangle \, (\; [\; \langle definicija_parametara \rangle \;] \;) \, ;$$

Primer 6.3 (Prototip funkcije za određivanje maksimuma dva broja)**.** U ovom primeru dato je rešenje zadatka iz primera 6.2, napisano za C99 i novije kompajlere, ali tako da je funkcija definisana tek nakon poziva. Zbog toga je neophodno pre glavnog programa (generalno, pre poziva funkcije) deklarisati funkciju. Deklaracija funkcije se u ovom primeru nalazi u 2. liniji programa.

```
1  #include <stdio.h>
2  int maksimum(int a, int b); // deklaracija (prototip)
3  main()
4  {
5      int x,y,M;
6      scanf("%d%d",&x,&y);
7
8      M = maksimum(x,y);  // poziv
9
10     printf("Broj %d je veci.",M);
11 }
12 int maksimum(int a, int b)  // definicija
13 {
14     int pom;
15     if (a > b)
16         pom = a;
17     else
18         pom = b;
19     return pom;         // povratna vrednost
20 }
```

△

6.2 Prenos parametara

Preko parametara funkcije funkciji se mogu preneti promenljive svih osnovnih i izvedenih tipova, uključujući i pokazivače i strukture. U C-u postoje dva tipa prenosa parametara:

1. prenos parametara po vrednosti, i
2. prenos parametara po referenci.

6.2.1 Prenos po vrednosti

Prenos parametara po vrednosti pokazaćemo na primeru programa 6.1.

Program 6.1

```
1  #include "stdio.h"
2  int f(int a)
3  {
4      int pom;
5      pom = a + 1;
6      a = a - 1;
7      return pom;
8  }
9  void main()
10 {
11     int x,y;
12     x = 5;
13
14     y = f(x);
15
16     printf("%d %d",x, y);
17 }
```

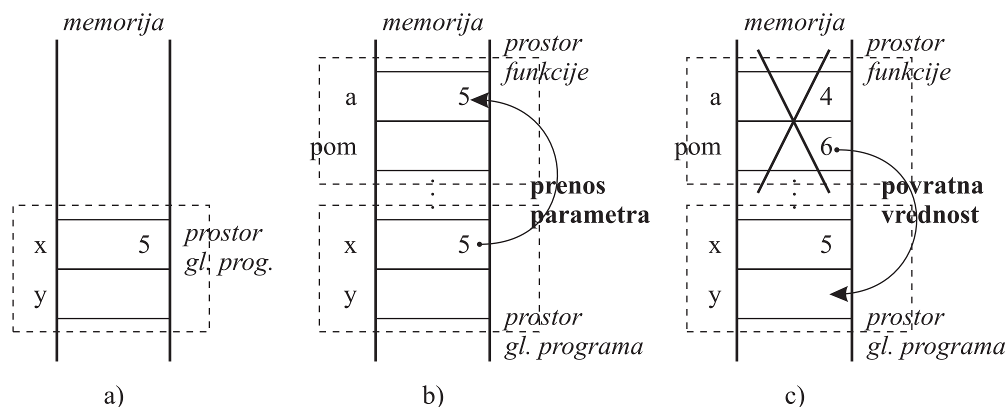
U programu 6.1 definisana je funkcija `int f(int a)`, koja ima jedan fiktivni parametar a, jednu lokalnu promenljivu *pom* i vraća rezultat tipa *int*. U glavnom programu se ova funkcija poziva sa stvarnim argumentom x i rezultat se upisuje u promenljivu y.

Primera radi, funkcija je napisana tako da u promenljivu *pom* upisuje vrednost argumenta a uvećanu za 1 (5. linija programa 6.1), a odmah zatim vrednost argumenta a smanjuje za 1. Ključno pitanje glasi: **Šta će prikazati glavni program?**

S obzirom na to da je vrednost promenljive $x = 5$, na osnovu koda programa, u promenljivu y će biti upisana vrednosti promenljive *pom*, koja je jednaka vrednosti argumenta povećanoj za 1, t.j. $y = 6$.

S druge strane, intuitivno se na osnovu koda programa 6.1 nameće i odgovor da će vrednost promenljive x biti smanjena za 1, s obzirom na to da ova promenljiva u pozivu odgovara fiktivnom argumentu a, čija se vrednost u funkciji smanjuje. Međutim, ovo nije tačno. Vrednost promenljive x se u ovom slučaju ne menja.

Da bismo objasnili mehanizam prenosa parametara i razlog zbog koga se vrednost promenljive x ne menja, razmotrićemo situaciju u memoriji nakon ključnih linija koda. Izgled memorije prikazan je na slici 6.5. Izgled memorije neposredno **pre** poziva funkcije u 14. liniji koda prikazan je na slici 6.5a, a neposredno **nakon poziva** funkcije, kada se prenese vrednost parametra i kreira pomoćna promenljiva u deklaraciji iz 4. linije, prikazana je na slici 6.5b.



Slika 6.5: Izgled memorije pri prenosu parametara po vrednosti za primer programa 6.1

Prilikom poziva funkcije, kreira se poseban memorijski prostor za parametre i lokalne promenljive funkcije, koji postoji u memoriji samo za vreme izvršavanja funkcije i briše se nakon završetka (slika 6.5b). Za fiktivni parametar a kreirana je odgovarajuća promenljva u memorijskom prostoru funkcije i u nju je upisana vrednost odgovarajućeg stvarnog argumenta. Zbog ovoga se ovakav načina prenosa parametara naziva prenos parametara **po vrednosti**.

Situacija u memoriji neposredno pre završetka funkcije naredbom *return* prikazana je na slici 6.5c. Vrednost parametra a u memorijskom prostoru funkcije i lokalne promenljive *pom* je promenjena izvršavanjem naredbi funkcije. Vrednost promenljive *pom* će nakon naredbe *return* biti upisana u promenljivu y. Međutim, promena parametra a će se "izgubiti", jer će ovaj prostor nakon završetka funkcije biti oslobođen.

Po ovome, odgovor na prethodno postavljeno pitanje je da će vrednosti koje će biti prikazane biti $x = 5$ i $y = 6$.

Napomena: Zaključak koji je potrebno izvesti iz ovog primera je da se kod prenosa parametara po vrednosti sadržaj argumenata može menjati, ali da se ta promena ne odražava na odgovarajuće stvarne argumente, t.j. promena parametara nije vidljiva u glavnom programu.

Fiktivni parametri funkcije mogu imati ista imena kao i promenljive u programu. Potrebno je naglasiti da se u ovom slučaju radi o različitim promenljivama, iako se isto zovu, koje se nalaze u dva različita dela memorije. Sledeći program ilustruje ovu situaciju.

```
1  #include "stdio.h"
2  int f(int x, int y)
3  {
4      return x+y;
5  }
```

```
6  main()
7  {
8      int x,y=2,z=3;
9      x = f(y,z);
10     printf("%d",x);
11 }
```

Da ne bi došlo do zabune dati program treba posmatrati ovako: u pozivu funkcije u 9. liniji koda, prvom parametru funkcije je predata vrednost y, a drugom vrednost z. Kako god se ti parametri u funkciji zovu, oni su za glavni program "prvi" i "drugi" parametar, i u prvi fiktivni parametar će biti upisana vrednost prvog stvarnog parametra, a u drugi druga. Sama funkcija određuje i vraća zbir prvog i drugog parametra. Dakle, u promenljivu x u liniji 9 će biti upisan zbir promenljivih glavnog programa y i z, odnosno vrednost 5.

6.2.2 Prenos po referenci

Kod prenosa parametara po vrednosti promena parametara funkcije je dozvoljena, ali ta promena nije vidljiva van funkcije. Po ovome, jedina vrednost koju funkcija može da vrati je vrednost vraćena naredbom *return*.

Prenos parametara po referenci je mehanizam C-a pomoću koga je moguće učiniti promenu vrednosti parametara vidljivom van funkcije. Na ovaj način je moguće vratiti više od jedne vrednosti iz funkcije.

Prenos parametara po referenci objasnićemo na primeru programa 6.2. Program 6.2 sličan je programu 6.1 uz sledeće sintaksne razlike:

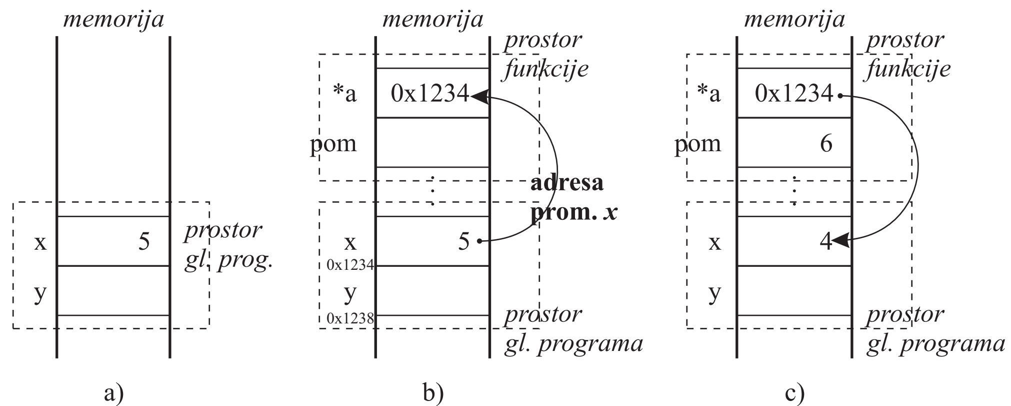
1. funkciji se, umesto celobrojne promenljive, **prenosi pokazivač** na celobrojnu promenljivu (2. linija koda),
2. s obzirom na to da se funkciji prenosi pokazivač, u svim izrazima u funkciji gde je potrebna vrednost promenljive, figuriše **dereferencirani pokazivač** ($*a$ u 5. i 6. liniji koda),
3. kako je parametar funkcije pokazivač, **stvarni argument poziva** funkcije je adresa promenljive, a ne vrednost promenljive ($\&x$ u 14. liniji koda).

Program 6.2

```
1  #include "stdio.h"
2  int f(int* a)
3  {
4      int pom;
5      pom = *a + 1;
6      *a = *a - 1;
7      return pom;
8  }
9  void main()
10 {
```

```
11     int x,y;
12     x = 5;
13
14     y = f(&x);
15
16     printf("%d␣%d",x, y);
17 }
```

Uz ove, može se reći neznatne sintaksne izmene, od kojih je ključna ta da je parametar funkcije pokazivač (2. linija programa 6.2), a ostale su samo posledica ove izmene, dobija se konceptualno drugačiji mehanizam prenosa parametara. Ovaj mehanizam ilustrovan je na slici 6.6. Na slici 6.6 prikazani su sadržaji memorijskih lokacija u ključnim trenucima izvršenja programa.



Slika 6.6: Izgled memorije pri prenosu parametara po referenci za primer programa 6.2

Na slici 6.6a prikazana je situacija u memoriji neposredno pre poziva funkcije u 14. liniji programa 6.2. U memorijsku lokaciju promenljive x upisana je vrednost, a kako funkcija još uvek nije pozvana, prostor za argumente i lokalne promenljive u memoriji nije alociran (slika 6.6a).

Izgled memorije nakon poziva funkcije i deklaracije lokalne promenljive *pom* prikazan je na slici 6.6b. U memorijskom prostoru funkcije alociran je deo za argument a i za lokalnu promenljivu *pom*. Pri pozivu funkcije argumentu a odgovara adresa na kojoj se nalazi promenljiva x, t.j. $\&x$, pa se ova adresa upisuje u promenljivu predviđenu za argument, a ne sama vrednost argumenta kao kod prenosa parametara po vrednosti. Na slici je primera radi za adresu promenljive x uzeta vrednost 0x1234.

S obzirom na to da je u pokazivaču a adresa, uvek kada se radi sa dereferenciranim pokazivačem $*a$ u funkciji zapravo se radi sa stvarnom vrednošću u **memo-**

rijskom prostoru glavnog programa. Ovo je prikazano na slici 6.6c. Linija 6 programa 6.2 smanjuje vrednost $*a$ za 1, što je prikazano na slici 6.6.

Nakon što se funkcija završi, njen memorijski prostor se oslobađa, ali promene ostaju vidljive u memorijskom prostoru glavnog programa. Zbog ove izmene možemo reći da je funkcija glavnom programu vratila dve vrednosti: jednu "regularno" preko *return* i drugu indirektno preko promene parametara.

Prenosom parametara po referenci može se postići da funkcija vrati više rezultata glavnom programu, čime se funkcije u C-u približavaju konceptu procedura drugih programskih jezika.

Napomena: Funkcija $scanf(\ldots)$ za učitavanje sa tastature, za razliku od funkcije $printf(\ldots)$ zahteva da kao argumenti budu navedene adrese promenljivih. Razlog za ovo je upravo prenos parametara po referenci, kako bi vrednosti koje korisnik unese bile vidljive u glavnom programu. Da se radi sa prenosom po vrednosti, funkcija $scanf$ ne bi mogla da vrati vrednosti.

6.2.3 Nizovi i matrice kao parametari funkcije

Nizovi, i generano polja, se funkciji prenose po referenci.

Prenos nizova preko parametara funkcije

Ukoliko je niz u glavnom programu deklarisan kao

```
int A[10];
```

tada funkciji treba preneti adresu početka niza. Za ovaj primer deklaracija funkcije bila bi

```
void funkcija(int* A);
```

Da bi se u funkciji imala informacija o broju elemenata niza, obično se preko parametra funkcije uz niz prenosi i celobrojni podatak sa informacijom o broju elemenata. Tako prethodna deklaracija postaje:

```
void funkcija(int* A, int N);
```

S obzirom na to da se radi o prenosu parametara funkciji preko reference, sve promene koje se izvrše nad elementima niza biće vidljive nakon završetka funkcije.

Elementima niza u funkciji moguće je pristupiti korišćenjem operatora dereferenciranja `*(A+i)`, ili klasično, korišćenjem operatora za indeksiranja niza '[' i ']', kao `A[i]`.

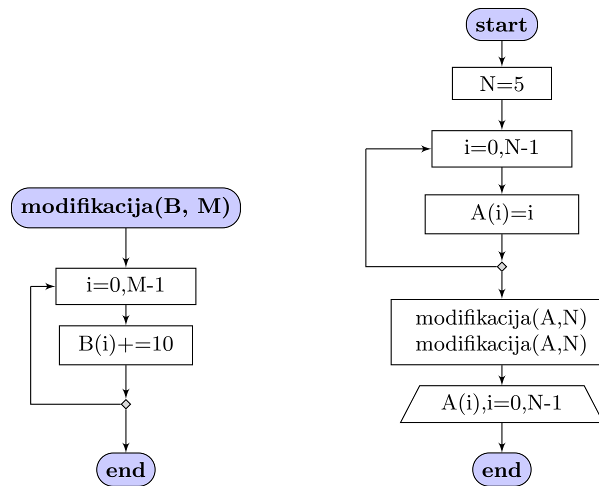
Prilikom pozivanja funkcije, na mestu fiktivnog argumenta gde je pokazivač na niz, s obzirom na to da se prenosi adresa, dovoljno je navesti samo ime niza.

Prenos nizova preko parametara funkcije ilustrovaćemo na sledećem primeru.

Primer 6.4 (Prenos niza preko parametara)**. Zadatak**: Napisati funkciju na C-u koja svaki elemenat niza prenetog preko parametara funkcije povećava za 10. U glavnom programu inicijalizovati elemente niza tako da je $a_i = i$, i vrednost svakog elementa korišćenjem formirane funkcije povećati za 20.

Rešenje: Dijagram toka algoritma prikazan je na slici 6.7.

Funkcija elemente niza prenetog preko fiktivnog parametra B uvećava za 10. Dužina niza je preneta preko fiktivnog parametra M. Da bi se elementi niza inicijalizovanog u glavnom programu povećali za 20, funkcija je pozvana dva puta. Program na C-u je dat u nastavku.



Slika 6.7: Dijagram toka algoritma sa funkcijom za modifikaciju vrednosti elemenata niza

```
1  #include "stdio.h"
2  void modifikacija(int* B, int M)
3  {
4      int i;
5      for(i = 0; i < M; i++)
6          B[i] += 10;            // svaki elemenat povecamo za 10
7  }
8  main()
9  {
10     int A[5], N=5, i;
11     for(i = 0; i < N; i++)
12         A[i] = i;              // A[0]=0, A[1]=1, itd.
13
14     modifikacija(A,N); // jednom pozvana f-ja
15     modifikacija(A,N); // jos jednom
```

```
16
17      for(i = 0; i < N; i++)
18         printf("%d,",A[i]);
19  }
```

Za nazive fiktivnih argumenata funkcije korišćeni su identifikatori koji su različiti od identifikatora promenljivih u glavnom programu, mada, kako je ranije rečeno i pokazano na primeru, svejedno je, s obzirom na to da su promenljive u glavnom programu i funkciji nezavisne. Funkcija nema povratnu vrednost koju vraća preko *return*, pa je zato tip funkcije *void*. Međutim, promene koje funkcija izvrši vidljive su van funkcije, a s obzirom na to da se funkcija poziva dva puta, program će na izlazu dati sledeće vrednosti:

```
20,21,22,23,24,
```

△

Umesto deklaracije iz 2. linije programa primera 6.4, moguće je koristiti i

```
void modifikacija(int B[], int M)
```

ili

```
void modifikacija(int B[5], int M)
```

U svakom od ovih slučajeva deklaracije funkcije radi se o istom mehanizmu prenosa parametra po referenci, a obe oznake `B[]` i `B[5]` signaliziraju kompajleru da se radi o pokazivaču `int* B`.

Polja iz funkcije nije moguće vratiti preko naredbe *return*. Imajući u vidu da je sam naziv niza pokazivač na prvi podatak u nizu, razmotrićemo ovo na primeru programa 6.3.

Program 6.3

```
1  #include "stdio.h"
2  int* niz()
3  {
4      int A[]={1,2,3,4,5};
5      return A;
6  }
7  main()
8  {
9      int *p;
10     p = niz();
11     for (int i = 0; i < 5; i++)
12        printf("%d␣",p[i]);
13 }
```

Funkcija *niz*() na početku deklariše niz A, čime se u memoriji rezerviše prostor za ovaj niz i u njega upisuju navedene inicijalne vrednosti.

Problem je u tome što se ovaj niz nalazi u memorijskom prostoru funckije, pa će funkcija preko *return* vratiti ispravnu adresu, ali na toj adresi neće biti niza kada se izvršenje vrati glavnom programu. Glavni program zbog toga prikazuje neke vrednosti koje se nalaze na toj adresi nakon oslobađanja memorijskog prostora funkcije. Izlaz iz programa je:

```
167 3734284 257070416 -422090225 -2
```

Matrice kao parametri funkcije

Matrice se takođe prenose funkciji preko reference. Promena vrednosti elemenata matrice u funkciji je zbog ovoga vidljiva van funkcije.

Kako je na slici 5.20 (strana 240) prikazano, samo ime matrice je dvostruki pokazivač (u primeru sa slike 5.20 je `int**`). Ukoliko se funkciji prenose i dimenzije matrice, matricu celobrojnih elemenata dimenzija NxM moguće je preneti funkciji na sledeći način:

```
void funkcija(int** A, int N, int M);
```

Ovakav način prenosa matrice funkciji komplikuje pristup elementima matrice u funkciji, jer je potrebno za svaki elemenat preračunati njegovu poziciju na osnovu njegovih indeksa i dimenzija matrice, a pristup izvršiti imajući u vidu reprezentaciju sa slike 5.20.

Olakšavajuća okolnost je mogućnost da se kompajleru signaliziraju **deklarisane** dimenzije matrice, onako kako su deklarisane u glavnom programu, na sledeći način:

```
void funkcija(int A[100][100], int N, int M);
```

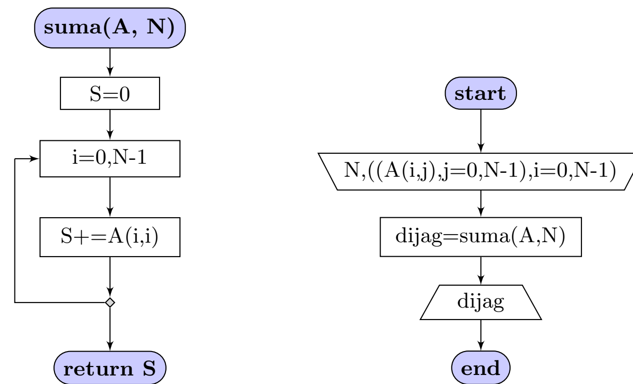
Napomena: Dimenzije koje se navode u uglastim zagradama moraju biti celobrojne konstante.

Ukoliko je deklaracija data na prethodno opisani način, elementima matrice je u funkciji moguće pristupati klasično, pomoću operatora [ ], kao na primer A[0][0]. U tom slučaju kompajler sam vodi računa o reprezentaciji.

S obzirom na to da kompajler ima evidenciju o ukupnom broju elemenata, iz deklaracije funkcije se prva dimenzima može izostaviti:

```
void funkcija(int A[][100], int N, int M);
```

Primer 6.5 (Funkcija za sumiranje elemenata glavne dijagonale matrice)**. Zadatak**: Nacrtati strukturni dijagram toka algoritma i na programskom jeziku C napisati strukturnu funkciju koja određuje i vraća sumu elemenata glavne dijagonale matrice prenete preko parametara. U glavnom programu sa tastature uneti dimenziju kvadratne matrice i elemente matrice, i korišćem formirane funkcije odrediti sumu elemenata na dijagonali. Prikazati sumu elemenata.



Slika 6.8: Dijagram toka algoritma sa funkcijom za sumiranje elemenata na glavnoj dijagonali matrice

Rešenje: Dijagram toka algoritma prikazan je na slici 6.8.

Rešenje zadatka dato je u nastavku. U funkciji i u glavnom programu korišćene su iste oznake za matricu i dimenziju matrice, pa je potrebno napomenuti da se bez obzira na ovo radi o različitim lokacijama za A, u jednom slučaju kao fiktivnom argumentu funkcije, i A kao stvarnom argumentu u drugom slučaju.

```
1  #include "stdio.h"
2  int suma(int A[30][30], int N)
3  {
4      int S = 0, i;
5      for (i = 0; i < N; i++)
6          S+=A[i][i];
7      return S;
8  }
9  main()
10 {
11     int A[30][30], N,i,j,dijag;
12     scanf("%d",&N);
13     for (i = 0; i < N; i++)
14         for (j = 0; j < N; j++)
15             scanf("%d",&A[i][j]);
16     dijag = suma(A,N);
17     printf("Suma␣je␣%d",dijag);
18 }
```

Izgled konzole prilikom izvršenja programa dat je u nastavku.

```
4
1 2 3 4
```

```
5 6 7 8
8 7 6 5
4 3 2 1
Suma je 14
```

△

Primer 6.6 (Funkcija za određivanje minimalnih vrednosti vrsta matrice)**. Zadatak**: Na programskom jeziku C napisati funkciju koja određuje minimalne vrednosti elemenata svake vrste celobrojne matrice prenete preko parametara funkcije. U glavnom programu inicijalizovati matricu i pozivom funkcije odrediti minimume svih vrsta.

Rešenje: S obzirom na to da funckija treba da vrati onoliko celih brojeva koliko matrica ima vrsta, ove vrednosti moguće je vratiti u formi elemenata niza. Program je dat u nastavku. Kako se niz prenosi po referenci, niz je statički deklarisan u glavnom programu i prenet je funkciji preko jednog od parametara.

```
1  #include "stdio.h"
2  void minimumi(int Mat[3][3], int Dim1, int Dim2, int* Niz)
3  {
4      int S = 0, i,j, min;
5      for (i = 0; i < Dim1; i++)
6      {
7          min = Mat[i][0];
8          for (j = 0; j < Dim2; j++)
9          if (Mat[i][j]< min)
10             min = Mat[i][j];
11         Niz[i] = min;
12     }
13 }
14 main()
15 {
16     int A[3][3] = {{1,2,3},{4,5,6},{7,8,9}}; // matrica
17     int N=3, B[3]; // dimenzija matrice i "prazan" niz
18     minimumi(A,N,N,B); // Broj i pozicija stravnih arg. mora se poklapati sa
       fiktivnim
19     for(int i = 0; i < N; i++)
20         printf("Minimum␣%d.␣vrste␣je␣%d.\n",i,B[i]);
21 }
```

Funkcija ima ulogu da vrednosti koje određuje upiše u niz prenet preko parametara. Funkcija ne deklariše niz, pošto je niz deklarisan u glavnom programu, već samo upisuje vrednosti u njega, i to: vrednost minimalnog elementa vrste sa indeksom 0 na nultu poziciju u nizu, vrste sa indeksom 1 na prvu poziciju, itd.
△

6.3 Funkcija *main*

Glavni program *main*() u C-u, koji je obavezan u svakom programu i od koga kreće izvršenje programa, zapravo predstavlja "glavnu" funkciju. Do sada je kao oznaka početka glavnog programa korišćen identifikator *main* bez navođenja parametara i povratne vrednosti. Međutim, kao i svaka druga funkcija u C-u, i funkcija *main*() može imati parametre i povratnu vrednost.

Ako se parametri funkcije *main*() ne navedu, zagrade, kao sastavni deo sintakse svake funkcije, su obavezne. Sintaksa funkcije *main*() je sledeća:

$$\langle glavna_funkcija\rangle \quad ::= \quad [\ \langle tip\rangle\ ]\ \textbf{main}\ '('\ [\langle argumenti\rangle]\ ')' \ \{ \ \langle telo_funckije_main\rangle \ \}$$

$$\langle tip\rangle \quad ::= \quad \textbf{int} \mid \textbf{void}$$
$$\langle argumenti\rangle \quad ::= \quad \textbf{int}\ argc, \textbf{char} ** \ argv$$

Sledeći program ilustruje ispravno napisanu definiciju funkcije *main* sa parametrima i povratnom vrednošću, na osnovu datog opisa sintakse.

```
1 int main(int argc, char** argv)
2 {
3     // ...
4 }
```

Identifikator *main* u C-u rezervisan je za glavni program. Funkcija *main*() ima opcione argumente koji se navode u malim zagradama, kao i opcioni tip, odnosno povratnu vrednost. Ako se agrumenti izostave, zagrade su obavezne, kao što se iz sintaksne definicije vidi, a ako se povratna vrednost izostavi podrazumeva se tip *void*.

Napomena: Neka razvojna okruženja i kompajleri, kao što je na primer okruženje *Microsoft Visual Studio 2010* zahtevaju eksplicitno navođenje tipa glavne funkcije i javljaju sintaksnu grešku ako se tip ne navede, pa je kod ovih kompajlera neophodno glavnu funkciju navoditi kao `void main()`. Neke ranije verzije ovog okruženja javljaju upozorenje (*warning*), ali nastavljaju sa prevođenjem programa.

Parametri i povratna vrednost funkcije *main*() služe za komunikaciju programa sa operativnim sistemom.

Povratna vrednost je obično ceo broj koji operativnom sistemu daje status završetka programa. Uobičajeno je da povratna vrednost 0 označava da se program završio bez grešaka. Naredba kojom se povratna vrednost vraća operativnom sistemu je *return*. Na primer:

```
1 int main()
2 {
3     // ...
4     return 0;
5 }
```

Ukoliko programer smatra da je došlo do neke greške prilikom izvršenja programa, kao na primer neispravan unos korisnika, ili slično, može obavestiti operativni sistem na kraju programa vraćajući vrednost različitu od 0.

Parametri funkcije *main* su parametri koje operativni sistem može predati programu prilikom pokretanja. Postoje dva načina kako je iz operativnog sistema moguće predati parametre programu prilikom pokretanja:

1. *Drag&Drop* – ukoliko se program pokreće iz *Windows* okruženja, moguće je pokrenuti program prevlačenjem neke datoteke i oslobađanjem preko izvršnog *.exe* programa (*drag&drop*), ili preko prečice na izvršni pogram (npr. na *desktop*-u). Tada operativni sistem kao parametar programu prenosi putanju i naziv datoteke koja je prevučena na izvršni fajl. Primer ove funkcionalnosti je *Microsoft Office*, koji otvara dokumenat prevlačenjem na prečicu programa koja se nalazi npr. na *desktop*-u *Windows* okruženja.

2. *CommandPrompt* – Ukoliko se u komandnom prozoru, u kome se izvršavaju konzolne aplikacije, pokrene program navođenjem imena programa, iza imena programa moguće je navesti proizvoljan broj parametara. Parametri su međusobno razdvojeni sa po jednim blanko znakom.

Operativni sistem programu prenosi dva parametra:

- *argc* – celobrojni podatak čija je vrednost jednaka ukupnom broju prenetih parametara, naziv je dobio kao skraćenica od engleskih reči ***arg**ument **c**ount* (prev. broj parametara), i
- *argv* – matrica karaktera (*char***), odnosno niz stringova, parametar je dobio naziv kao skraćenica od engleskih reči ***arg**ument **v**alues* (prev. vrednosti parametara).

Svaki string u ovom nizu stringova *argv* je po jedan parametar koji operativni sistem prenosi programu. Kao prvi string se obavezno prenosi ime programa, a zavisno od operativnog sistema i putanja do njega.

Primer 6.7 (Prikazivanje parametara funkcije *main*)**.** U ovom primeru ilustrovana je upotreba povratne vrednosti i parametara funkcije *main*(). Program prikazuje sve parametre koji su preneti preko operativnog sistema.

Svakom parametru, od ukupno *argc* parametara, pristupa se u *for* petlji radi prikaza *i*-tog stringa iz niza parametara sa *argv*[*i*].

```
1 #include "stdio.h"
2 int main(int argc, char** argv)
3 {
```

```
4      int i;
5      printf("Preneto␣je␣ukupno␣%d␣parametara.\n",argc);
6      for (i = 0; i < argc; i++)
7          printf("%d.␣parametar␣je␣%s.\n",i,argv[i]);
8      return 0;
9  }
```

Neka je naziv izvršne datoteke je `aip.exe`. Tada je, na primer, ovaj program sa parametrima iz konzole moguće pozvati navođenjem:

```
aip.exe prvi 8 /aip 3125
```

Prvi parametar je string, a drugi ceo broj. Bez obzira na tip parametra, svi se parametri programu prenose kao nizovi karaktera. Treći parametar koji je naveden je string "/aip", a četvrti 3125. Izgled konzole prilikom izvršenja programa prikazan je u nastavku.

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

c:\Users\vciric\VS2010\aip\debug>aip.exe prvi 8 /aip 3125
Preneto je ukupno 5 parametara.
0. parametar je aip.exe.
1. parametar je prvi.
2. parametar je 8.
3. parametar je /aip.
4. parametar je 3125.

c:\Users\vciric\VS2010\aip\debug>
```

△

Umesto deklaracije funkcije *main*

```
int main(int argc, char** argv)
```

drugi parametar, imajući u vidu da se radi o nizu stringova, moguće je navesti na sledeći način:

```
int main(int argc, char* argv[])
```

6.4 Rekurzivne funkcije

Rekurzija je konstrukcija kod koje se pojam koji se definiše koristi za definiciju samog tog pojma.

Definicija 6.2 (Rekrzivna funkcija)**.** Rekurzivna funkcija je funkcija koja u svom telu poziva samu sebe.

Da bi rekurzija kod funkcija bila upotrebljiva, potrebno je u nekom trenutku uslovno prekinuti rekurziju. Deo tela funkcije koji sadrži uslov za prekid rekurzije naziva se **kraj rerurzije**. Rekurzivne funkcije obično imaju dva dela: deo za kraj rekurzije, i deo za nastavak rekurzije u kom pozivaju samu sebe.

Kada rekurzivna funkcija u telu funkcije naiđe na poziv iste te funkcije, izvršenje trenutne instance funkcije se zaustavlja, a kreira se i izvršenje započinje nova instanca iste te funkcije. Za narednu instancu se kaže da je na **drugom nivu** po dubini rekurzije. Funkcija na prvom nivou će izvršenje nastaviti tek kada se završi funkcija na drugom nivou. Ukoliko funkcija na drugom nivou poziva dalje samu sebe, pokrenuće treći nivo, itd.

Definicija 6.3 (Dubina rekurzije)**.** Dubina rekurzije je maksimalni broj nivoa do koga se ide prilikom poziva rekurzivne funkcije, prilikom rešavanja konkretnog problema.

Primer 6.8 (Rekurzivna funkcija za izračunavanje faktorijela broja)**.** Matematička definicija faktorijela broja n data je jednačinom (2.6) u primeru 2.23 (strana 69). Funkcija koja određuje i vraća faktorijel broja na osnovu jednačine (2.6) i algoritma predstavljenog dijagramom toka na slici 2.40 data je u nastavku.

```
1 int fakt(int n)
2 {
3     int i, F=1;
4     for(i=1;i<=n;i++)
5         F*=i;
6     return F;
7 }
```

Matematička definicija faktorijala (2.6) može biti data i rekurzivno kao:

$$n! = \begin{cases} nije\ def., & n < 0 \\ 1, & n = 0 \\ n \cdot (n-1)! \end{cases} . \qquad (6.1)$$

Deo izraza (6.1), $n! = n \cdot (n-1)!$, predstavlja rekurzivni deo, jer za definisanje pojma faktorijela koristi pojam koji se definiše. Kraj rekurzije u ovoj definiciji je deo jednačine $n! = 1$, za $n = 0$. Rekurzivna funkcija na osnovu jednačine (6.1) je data u okviru sledećeg programa.

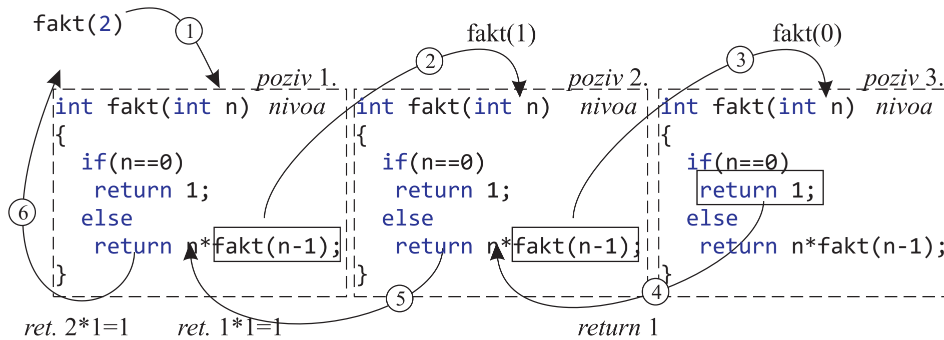
```
1 #include "stdio.h"
2 int fakt(int n)
3 {
4     if(n==0)
5         return 1;
6     else
7         return n*fakt(n-1);
8 }
9 void main()
```

```
10  {
11          printf("Faktorijel␣broja␣5␣je␣%d.", fakt(5) );
12  }
```

Izlaz koji ovaj program daje je:

```
Faktorijel broja 5 je 120.
```

Ilustracija poziva ove rekurzivne funkcije data je na slici 6.9. Kako bi slika bila jednostavnija i preglednija, na slici 6.9 prikazan je poziv rekurzivne funkcije za početnu vrednost $n = 2$. Na slici je strelicama označen redosled poziva i povratka iz različitih nivoa rekurzije, kao i vrednosti kojima se određene instance pozivaju i vrednosti koje vraćaju. Svakoj strelici na slici 6.9 pridružen je redni broj u redosledu izvršenja. △



Slika 6.9: Ilustracija poziva rekurzivne funkcije na primeru funkcije za izračunavanje faktoriela

U cilju realizacije rekurzije, prilikom prelaska sa jednog nivoa rekurzije na naredni nivo, kompajler na stek pamti stanje u kom je trenutna funkcija zaustavljena. Povratkom na prethodni nivo rekurzije, sa vrha steka se uzimaju neophodne informacije, kako bi izvršenje moglo da se nastavi sa mesta gde se stalo.

Napomena: Ukoliko rekurzivna funkcija nema uslov za kraj rekurzije, rekurzivni pozivi nisu ograničeni i dubina rekurzije se povećava sve dok ne napuni stek koji je kompajler rezervisao za potrebe rekurzije. Ovakvi programi se obično završavaju greškom "prekoračenje steka" (eng. *stack overflow*).

Primer 6.9 (Rekurzivna funkcija za određivanje NZD brojeva)**.** NZD se rekurzivno može definisati kao:

$$NZD(n,m) = \begin{cases} n, & n = m \\ NZD(n-m,m), & n > m \\ NZD(m-n,n), & n < m \end{cases} . \tag{6.2}$$

Rekurzivna funkcija za određivanje NZD broja je data u nastavku.

```
1  int NZD(int n, int m)
2  {
3      if(n==m)
4          return n;
5      else
6          if (n>m)
7              return NZD(n-m,m);
8          else
9              return NZD(m-n,n);
10 }
```

△

Rekurzivne funkcije nalaze veliku primenu kod algoritama za obilazak struktura podataka. Obilazak nelinearnih struktura jednostavnije je implementirati rekurzivno.

6.5 Memorijske klase promenljivih

Svaka promenljiva u C-u, pored tipa, ima i svoju memorijsku klasu kojoj pripada.

Definicija 6.4 (Memorijska klasa)**.** Memorijska klasa promenljive utvrđuje postojanost (trajnost) i područje važenja podataka u memoriji.

Postoje 4 memorijske klase:

1. automatska,
2. eksterna,
3. statička, i
4. registarska klasa.

6.5.1 Automatska klasa i lokalne promenljive

Za promenljivu koja je deklarisana u okviru glavnog programa ili neke druge funkcije podrazumeva se da pripada automatskoj (*auto*) klasi. Ove promenljive su takođe poznate i kao **lokalne promenljive**, jer su lokalne za glavni program, odnosno funkciju u kojoj su deklarisane i nemaju značenje, niti postoje van te funkcije.

Ključna reč za deklaraciju promenljivih automatske klase je **auto**, mada se ova ključna reč podrazumeva i ako se ne navede. Sve do sada razmatrane i korišćene deklaracije promenljivih su podrazumevale ključnu reč *auto* ispred tipa. Sledeći program ilustruje eksplicitnu deklaraciju promenljive u klasi *auto*.

```
1  main()
2  {
3      auto int x;
4      x = 3;
5  }
```

Kod ANSI C-a deklaracije promenljivih se po sintaksi nalaze na početku glavnog programa, odnosno funkcija, pa je najmanja jedinica lokalnosti promenljive u ANSI C-u funkcija. Noviji kompajleri dozvoljavaju da funkcija bude lokalna na nivou bilo kog bloka, pa i bloka petlje ili grananja. Kod ovih kompajlera promenljivu je moguće deklarisati bilo gde, s tim da je lokalna u bloku u kome se nalazi.

Primer 6.10 (Lokalna promenljiva u bloku tela petlje)**.** Sledeći primer ilustruje lokalnost promenljive na nivou bloka tako što promenljive i, pom i F deklariše i koristi samo u bloku tela petlje.

```
1  #include "stdio.h"
2  void main()
3  {
4      auto int x;
5      for (int i = 0; i < 10 ; i++)
6      {
7          auto int pom = 0; // eksplicitno receno da je auto
8          pom = pom + i;
9          int F = 1;        // podrazumeva se klj. rec auto
10         F = F * i;
11     }
12     pom = 2;
13     x = 3;
14 }
```

Promenljiva x je lokalna za glavni program i može joj se pristupiti iz bilo kog dela glavnog programa, kako je i pokazano u primeru. Međutim, kako je promenljiva *pom* lokalna za blok tela *for* petlje, naredba u 12. liniji je **sintaksno neispravna**, jer ova promenljiva ne postoji za vreme izvršenja tog dela koda. Sintaksno ispravno bi bilo umesto naredbe u liniji 12. pisati:

```
int pom = 2;
```

Na ovaj način bi se ponovo izvršila deklaracija promenljive i promenljiva bi postojala do kraja programa. Vrednost promenljive koja je bila u promenljivoj u telu petlje bi u ovom slučaju u međuvremenu bila izgubljena i ne bi se vratila ponovnom deklaracijom.

Preciznije rečeno, vrednost promenljive *pom* gubi se nakon svake iteracije petlje, tako da ovaj program na kraju neće sumirati sve indekse u promenljivu *pom*, kako to na prvi pogled izgleda, već će u svakoj iteraciji petlje vrednost 0 biti sabrana sa vrednošću trenutnog indeksa.

Potrebno je napomenuti da promenljiva F postoji samo u 9. i 10. liniji. △

Od pozicije na kojoj je promenljiva deklarisana zavisi vidljivost promenljive iz drugih delova programa, kao i trajanje promenljive u memoriji. Promenljiva ne mora postojati u memoriji sve vreme izvršenja programa, što je osobina koja se može iskoristiti za uštedu memorijskog prostora i optimizaciju programa.

6.5.2 Eksterna klasa i globalne promenljive

Promenljivama eksterne klase moguće je pristupiti iz bilo koje funkcije programa. Ove promenljive se nazivaju i globalnim promenljivama.

Definicija 6.5 (Globalna promenljiva)**.** Globalna promenljiva je promenljiva deklarisana van glavnog programa i drugih funkcija, kojoj se može pristupiti iz svih delova programa.

Globalne promenljive postoje u memoriji za sve vreme izvršenja programa i imaju globalnu vidljivost.

Primer 6.11 (Globalne promenljive)**.** U sledećem programu promenljive a, b, c i d deklarisane su u 3. liniji koda, van glavnog programa i ostalih funkcija. Zbog pozicije deklaracije, ove promenljive smatraju se globalnim promenljivama i moguće im je pristupiti iz svih delova programa.

```
1  #include "stdio.h"
2
3  int a,b,c,d;    // globalna deklaracija
4
5  void f1()
6  {
7      c = a + b;
8  }
9  void f2()
10 {
11     d = a * b;
12 }
13 void main()
14 {
15     a = 5; b = 6;
16     f1();
17     f2();
18     printf("%d␣%d",c,d);
19 }
```

Na početku, glavni program u 15. liniji pristupa promenljivama i upisuje vrednosti u promenljive a i b. Nakon toga, glavni program poziva funkcije $f1()$ i $f2()$, bez prenosa parametara, i bez povratne vrednosti. Međutim, funkcije rade sa globalnim promenljivama, pa su vrednosti koje je glavni program upisao u promenjive a i b vidljive u funkcijama, kao što su i rezultati izvršenja funkcija takođe vidljivi u glavnom programu kroz globalne promenljive c i d. $\triangle$

Imajući u vidu vidljivost i vreme trajanja globalnih promenljivih, čini se da uz globalne promenljive prenos parametara preko parametara funkcije nije neophodan. Međutim, glavni nedostatak je što se tako napisanim funkcijama ne mogu predati bilo koji podaci, već ove funkcije rade samo sa globalnim promenljivama za koje su

napisane, što im umanjuje upotrebljivost. Naravno, u svakom trenutku programer može simulirati prenos parametara upisom podataka u globalne promenljive, ali je ovo neoptimalno jer ove promenljive zauzimaju mesto u memoriji i kada nisu potrebne.

Na primer, funkcija $f1$ iz primera 6.11 uvek sabira sadržaj globalnih promenljivih a i b.

Globalnim deklaracijama mogu se deklarisati i polja, ali i druge strukture podataka. Sledeći program ilustruje globalnu deklaraciju niza.

Primer 6.12 (Niz kao globalna struktura podataka)**.** U ovom primeru glavni program u globalno definisani niz upisuje podatke koje korisnik unese sa tastarure i poziva funkciju. Funkcija nema parametre, već koristi globalne podatke, a rezultat vraća naredbom *return*.

```
1  #include "stdio.h"
2  int a[50], N;   // globalna deklaracija
3  int sum()
4  {
5      int S = 0;
6      for (int i = 0; i < N; i++)
7          S += a[i];
8      return S;
9  }
10 void main()
11 {
12     // Unos
13     scanf("%d",&N);            // br. elemenata
14     for (int i = 0; i<N; i++)
15         scanf("%d",a+i);       // elementi
16     printf("%d", sum() );
17 }
```

△

Kada se radi sa relativno velikim programima koji se sastoje od više od jednog fajla, ukoliko je globalna promenljiva deklarisana u jednom fajlu, a koristi se u drugom fajlu, kompajler će javiti grešku. Da bi se rešio ovaj problem uvedena je ključna reč **extern** koja se navodi uz deklaraciju promenljive u onom fajlu u kom se promenljiva koristi. Ova ključna reč signalizira kompajleru da je promenljiva deklarisana kao globalna, ali je deklarisana u nekom drugom fajlu.

Primer 6.13 (Primer upotrebe ključne reči *extern*)**.** Neka prvi fajl sadrži deklaraciju promenljive i poziv funkcije koja se nalazi u drugom fajlu, na način dat u sledećem programu.

```
1  int prom=5;
2  main()
3  {
```

```
4     prikazi_prom();
5 }
```

Tada se drugi fajl korišćenjem ključne reči *extern* referencira na promenljivu iz prvog fajla i prikazuje njenu vrednost kada se pozove funkcija *prikazi_prom*().

```
1 extern int prom;
2 void prikazi_prom()
3 {
4     printf("%i\n", prom);
5 }
```

△

Kada se globalne promenljive koriste za komunikaciju između različitih fajlova, neophodno je kompajleru istovremeno predati na prevođenje sve fajlove.

6.5.3 Statičke promenljive

Statički deklarisane promenljive su lokalne promenljive koje ne gube sadržaj nakon završetka bloka u kom su deklarisane. Za deklaraciju statičkih promenljivih ispred oznake tipa navodi se ključna reč **static**.

Primer 6.14 (Primer upotrebe statičkih promenljivih)**.** Sledeći program sadrži funkciju sa jednom statičkom promenljivom (a) i jednom automatskom promenljivom (b). Funkcija prikazuje vrednosti ove dve promenljive, a nakon prikaza inkrementira vrednosti obe promenljive za 1. Glavni program poziva funkciju 5 puta.

```
1  #include "stdio.h"
2  void prikazi()
3  {
4      int a;
5      static int b;
6
7      printf("%d %d\n", a,b);
8      a++; b++;
9  }
10 void main()
11 {
12     for (int i = 0; i < 5; i++)
13         prikazi();
14 }
```

Statička promenljiva b, za razliku od automatske promenljive a, zadržava vrednost i nakon završetka funkcije, pa su vrednosti koje program prikazuje:

```
0 0
```

```
0 1
0 2
0 3
0 4
```

△

6.5.4 Registarske promenljive

Registarske promenljive su slične automatskim promenljivama po lokalnosti i trajanju. Registarske promenljive kompajler, u cilju bržeg izvršavanja programa, pamti u registrima procesora, umesto u operativnoj memoriji računara. Na ovaj način je moguće postići značajno brži pristup prilikom čitanja i upisa podataka.

Ključna reč kojom se promenljiva deklariše u registarskoj klasi je **register** i navodi se u deklaraciji ispred oznake tipa.

Primer 6.15 (Primer upotrebe registarskih promenljivih)**.** Upotreba registarskih promenljivih ilustrovana je na primeru sledećeg programa.

```
1 #include "stdio.h"
2 void main()
3 {
4     register unsigned int x = 0;
5     register short i;
6     for (i = 0; i < 30000; i++)
7         x=x+i;
8 }
```

△

Imajući u vidu da je broj registara procesora ograničen i relativno mali, u registre procesora treba smeštati samo one promenljive kojima se veoma često pristupa i čije bi smeštanje u registre procesora dovelo do ubrzanja izvršenja programa.

Ukoliko programer deklariše više promenljivih u ovoj klasi od raspoloživog broja registra, neke promenljive neće biti smeštene u registre već u operativnu memoriju računara. Može se reći da ključna reč *register* označava želju programera, a ne striktnu naredbu kompajleru.

6.6 Standardne funkcije C-a

Kompajler za programski jezik C dolazi sa velikim brojem gotovih funkcija, grupisanih na osnovu njihove namene u posebne biblioteke, tzv. *header* fajlove. Korišćenjem ovih funkcija programer može značajno uštedeti na vremenu potrebnom za razvoj aplikacije, tako što će umesto da samostalno razvija pojedine algoritme, koristiti gotove funkcije iz biblioteka funkcija. Veliki broj ovih funkcija isti je i može se

naći kod C kompajlera različitih proizvođača. Ove funkcije nazivamo *standardnim funkcijama C-a*.

Kako postoji veliki broj funcija, a za upotrebu svake funkcije je potrebno znati dejstvo funkcije, njeno ime, parametre i povratnu vrednost, za svaki kompajler se može naći obimna dokumentacija koja opisuje funkcije[1]. Sve funkcije se dokumentuju na isti način navođenjem:

1. deklaracije funkcije,
2. opisa dejstva funkcije,
3. opisa parametara,
4. opisa povratne vrednosti funkcije,
5. naziva biblioteke u kojoj se funkcija nalazi, i
6. navođenjem kratkog primera upotrebe funkcije.

U ovom poglavlju obradićemo najčešće korišćene funkcije iz standardnih biblioteka C-a. Biće obuhvaćene funkcije za matematička izračunavanja, funkcije za rad sa stringovima, funkcije za rad sa fajlovima na disku i funkcije za dinamičku alokaciju memorije.

6.6.1 Funkcije za matematička izračunavanja

Funkcije za matematička izračunavanja nalaze se u standardnoj biblioteci *math.h*. Ove funkcije implementiraju standardne matematičke funkcije kao što su stepen, koren, sinus, kosinus, apsolutna vrednost, i sl.

Većina funkcija iz *math.h* rade sa realnim brojevima. Tip podataka koji ove funkcije koriste je *double*.

Napomena: Bez obzira na tip podataka u deklaraciji funkcije, treba imati u vidu da u C-u postoji implicitna konverzija tipa, pa će svaki stvarni parametar koji nije istog tipa kao fiktivni parametar funkcije automatski biti preveden u taj format. Isto važi i za povratne vrednosti.

Funkcije koje za parametar imaju ugao ili vraćaju ugao, uglove izražavaju u radijanima.

Deklaracije osnovnih trigonometrijskih funkcija standardne biblioteke *math.h* su:

- `double sin(double x);` – sinus broja x,
- `double cos(double x);` – kosinus broja x,
- `double tan(double x);` – tangens broja x.

[1]Uz okruženje Microsoft Visual Studio isporučuje se dokumentacija koja se naziva *Microsoft Developer Network* - MSDN, i koja sadrži opis svih funkcija dostupnih u bibliotekama. Ova dokumentacija javno je dostupna i na Internetu.

Inverzne trigonometrijske funkcije osnovnih trigonometrijskih funkcija su `asin`, `acos`, i `atan`.

Pored navedenih funkcija, standardna biblioteka *math.h* sadrži i hiperboličke funkcije `sinh`, `cosh`, i `tanh`, kao i njihove inverzne funkcije `asinh`, `acosh`, i `atanh`.

Primer 6.16 (Trigonometrijske funkcije iz standardne biblioteke)**.** Sledeći program ilustruje upotrebu funkcije *sin* iz standardne biblioteke *math.h*. Program prikazuje redom sve vrednosti sinusa brojeva od počev 0 do 2π sa korakom 0.1.

```
1  #include "stdio.h"
2  #include "math.h"
3  #define Pi 3.1415
4  main()
5  {
6      float alfa;
7      for (alfa = 0; alfa < 2*Pi; alfa += 0.1)
8      {
9          float sa = sin(alfa);
10         printf("%5.3f\n",sa);
11     }
12 }
```

Napomena: Da se u ovom primeru umesto funkcije $printf$ koristi neka funkcija za iscrtavanje piksela na ekranu, i da se iscrtavaju pikseli na koordinatama $(alfa, sa)$, program bi grafički prikazao izgled funkcije *sin*.

△

Neke od osnovnih eksponencijalnih funkcija koje su sastavni deo standardne biblioteke *math.h* su:

- `double pow(double x, double y);` – funkcija određuje i vraća x^y,
- `double exp(double x);` – funkcija određuje i vraća vrednost e^x,
- `double log(double x);` – prirodni logaritam broja $x, log_e(x)$.
- `double log10(double x);` – logaritam broja x za osnovu 10, $log_{10}(x)$
- `double sqrt(double x);` – kvadratni koren broja x, $\sqrt{x}$

Često korišćena funkcija je i funkcija za određivanje apsolutne vrednosti broja. Deklaracija ove funkcije je

```
double fabs(double x);
```

Ova funkcija određuje i vraća apsolutnu vrednost broja x, $|x|$.

Primer 6.17 (Primer upotrebe funkcije za određivanje apsolutne vrednosti broja).
Zadatak: Napisati program koji od korisnika zahteva unos realnih brojeva i prikazuje apsolutnu vrednost razlike svaka dva susedna uneta broja. Program završiti kada je razlika dva uzastopna broja manja od 1.

Rešenje: Kako su u svakom trenutku za rešenje ovog problema neophodne samo poslednje dve unete vrednosti, dovoljno je poslednje dve unete vrednosti pamtiti u dve promenljive.

```
1  #include "stdio.h"
2  #include "math.h"
3  main()
4  {
5      float xp, xn;
6      scanf("%f", &xn);
7      do
8      {
9          xp = xn;
10         scanf("%f", &xn);
11         printf("Razlika je %5.2f.\n", fabs(xn-xp) );
12     }
13     while( fabs(xn-xp) > 1.0 );
14 }
```

Program vrednosti koje korisnik unosi pamti u promenljivama xp i xn, uz pomeranje xn u xp, kako bi "oslobodio" mesto za naredni unos. Nakon unosa nove vrednosti u xn određuje se i prikazuje apsolutna vrednost razlike unetih brojeva, a zatim se ova vrednost "pomera" u xp da bi se naredna vrednost mogla uneti u xn.

Izgled konzole prilikom izvršenja programa dat je u nastavku.

```
4.2
-1.7
Razlika je  5.90.
3.7
Razlika je  5.40.
2.2
Razlika je  1.50.
2.0
Razlika je  0.20.
```

△

6.6.2 Funkcije za rad sa stringovima

Obrada tekstualnih podataka u okviru programa je veoma česta. Zbog ovoga programski jezici u manjoj ili većoj meri imaju podršku za osnovne operacije za obradu tekstualnih podataka.

Programski jezik C nema operatorsku podršku za rad sa stringovima, ali zato ima veliki broj funkcija koje se nalaze u standardnoj biblioteci *string.h*.

U C-u je moguće direktno manipulisati karakterima stringova, kao što je pokazano u poglavlju 5.3.4, ili je moguće koristiti neku od standardnih funkcija. Naravno, ma koliko veliki broj standardnih funkcija bio, uvek postoje problemi i tipovi obrade koji nisu pokriveni funkcijama, pa je u ovakvim slučajevima neophodno vršiti direktnu manipulaciju nad karakterima stringova. Svakako, čak i ako je problem složeniji, standardne funkcije i kombinacija standardnih funkcija mogu se koristiti kako bi se pojednostavila implementacija bar dela problema.

U daljem tekstu obradićemo neke od najčešće korišćenih funkcija.

strlen

Funkcija *strlen* se koristi za određivanje dužine stringa. Poslednja tri slova naziva funkcije zadata su tako da asociraju na namenu funkcije - ***len**gth*, odnosno u prevodu sa engleskog, dužina.

1. **Deklaracija funkcije**

   ```
   size_t strlen(char* S);
   ```

2. **Opis dejstva**

   Funkcija određuje veličinu stringa u karakterima, odnosno, imajući u vidu veličinu podataka tipa *char*, u bajtovima, ne uključujući karakter za kraj stringa.

3. **Parametri**

   Funkcija ima jedan parametar, koji predstavlja pokazivač na početak stringa čija se dužina određuje. Prilikom poziva funkcije potrebno je kao stvarni parametar navesti samo identifikator stringa, jer je identifikator niza u C-u pokazivač na početak niza, t.j. po tipu je sam identifikator stringa pokazivač na podatak tipa *char*.

4. **Povratna vrednost**

   Funkcija vraća dužinu stringa izraženu u broju bajtova. Ova vrednost ujedno je jednaka i indeksu pozicije u stringu gde se javlja karakter '\0' za označavanje kraja stringa. Takođe je jednaka i broju bajtova koji string zauzima u memoriji.

   Tip povratne vrednosti je **size_t**. Tip *size_t* je numerički tip, koji se koristi za neoznačene cele brojeve i, kako je standardom definisano, minimalna dužina podataka ovog tipa treba da iznosi 16 bitova. Po ISO C standardu C99, namena tipa *size_t* je deklaracija promenljivih koje sadrže podatak o veličini memorije. Naime, s obzirom na to da su svi ostali tipovi u ANSI C-u orjentisani ka veličini registara procesora i mogućnostima procesora, uveden je novi numerički tip *size_t* koji je memorijski-orjentisan. Ovaj tip treba da

ponudi dovoljan opseg vrednosti da se može zapamtiti veličina u bajtovima bilo kog objekta u memoriji. Konverzija iz ovog tipa u bilo koji drugi tip je, kao i za sve druge tipove, implicitna.

Primer 6.18 (Funkcija *strlen*)**.** Sledeći program određuje i vraća broj karaktera u stringu S. Bez obzira što funkcija *strlen* vraća podatak tipa *size_t*, u ovom primeru iskorišćena je mogućnost implicitne konverzije, pa je rezultat koji funkcija vraća dodeljen promenljivoj tipa *int*. Program će u ovom primeru prikazati vrednost 12.

```
1 #include "string.h"
2 #include "stdio.h"
3 void main()
4 {
5     char S[] = "Hello world!";
6     int rez = strlen(S);
7     printf("%d", rez );
8 }
```

△

strcpy

Funkcija *strcpy* kopira sadržaj jednog stringa u drugi. Poslednja tri slova u nazivu funkcije su skraćenica engleske reči ***copy*** **-** kopija, kopirati.

1. **Deklaracija funkcije**

   ```
   char* strcpy(char* S1, const char* S2);
   ```

2. **Opis dejstva**

   Funkcija vrši kopiranje sadržaja stringa $S2$ u string $S1$.

3. **Parametri**

   Funkcija za parametre ima dva stringa: $S1$ i $S2$. Ovi parametri su po tipu adrese na kojima se nalaze karakteri (*char**), pa je prilikom poziva funkcije potrebno proslediti samo identifikatore stringova.

   $S1$ je odredišni string, a $S2$ je izvorišni string. Sadržaj stringa $S2$ se kopira u $S1$, brišući postojeći sadržaj stringa $S1$. U $S1$ se kopira i karakter za označavanje kraja iz stringa $S2$.

   Prilikom poziva za parametar $S1$ neophodno je navesti promenljivu, dok parametar $S2$ može biti bilo promenljiva ili literal (poglavlje 3.3.5, strana 99). Ovo je posledica toga što funkcija prilikom poziva neće promeniti sadržaj stringa $S2$, a hoće sadržaj stringa $S1$. Zbog ovoga u deklaraciji funkcije uz parametar $S2$ stoji da je parametar tipa **const char***.

4. **Povratna vrednost**
   Funkcija vraća pokazivač na prvi karakter odredišnog stringa.

Primer 6.19 (Funkcija *strcpy*)**.** Sledeći program ilustruje slučaj kada je za drugi parametar funkcije *strcpy* naveden literal.

```
1 #include "string.h"
2 void main()
3 {
4     char S[80];
5     strcpy(S, "Hello world!");
6 }
```

U ovom programu nije iskorišćena povratna vrednost. Generalno, bez obzira da li funkcija u C-u ima povratnu vrednost ili ne, ako ova vrednost nije potrebna, ne mora se upisati u promenljviu.

Da je bilo potrebno uzeti povratnu vrednost, bilo bi potrebno deklarisati još jedan pokazivač, nrp. `char* rez;` i pozvati funkciju kao:

```
rez = strcpy(S, "Hello world!");
```

Napomena: Potrebno je napomenuti da s obzirom na to da C nema operator dodele '=' koji za operande može imati stringove, funkcija *strcpy* predstavlja najjednostavniji način za upis vrednosti u string u toku izvršenja programa. Operator dodele postoji, ali samo za inicijalizaciju prilikom deklaracije, ali ne i za dodelu vrednosti stringu u toku izvršenja programa.

△

Primer 6.20 (Funkcija *strcpy* sa fleksibilnim adresama)**.** Parametri funkcije *strcpy* su adrese, ali nigde nije eksplicitno rečeno da ove adrese moraju biti adrese prvih karaktera stringova koji se prenose. Imajući ovo u vidu, ovom funkcijom je moguće postići interesantne efekte.

Sledeći program na mesto druge reči prvog stringa kopira drugu i treću reč drugog stringa. Preciznije, program sadržaj stringa y, počev od 7. karaktera, kopira u string x počev od 9. karaktera.

```
1 #include "stdio.h"
2 #include "string.h"
3 void main()
4 {
5     char x[80] = "Elementi matematicke analize";
6     char y[80] = "Osnove algoritama i programiranja";
7     strcpy(x+9, y+7);
8     printf("%s",x);
9 }
```

Rezultat izvršenja ovog programa je string $x =$*"Elementi algoritama i programiranja"*. △

strncpy

Funkcija *strncpy* se koristi za kopiranje tačno definisanog broja karaktera jednog stringa u drugi.

1. **Deklaracija funkcije**

   ```
   char* strncpy(char* S1, const char* S2, size_t N);
   ```

2. **Opis dejstva**

   Funkcija vrši kopiranje N karaktera iz stringa $S2$ u string $S1$.

3. **Parametri**

   $S1$ – početna adresa na koju se kopira sadržaj stringa $S2$. Ovaj parametar pri pozivu mora biti identifikator promenljive, jer se sadržaj ovog stringa menja u toku izvršenja funkcije.

   $S2$ – početna adresa sa koje se kopira. Ovaj parametar pri pozivu može biti bilo identifikator promenljive ili literal. Vrednost ovog parametra se neće promeniti prilikom izvršenja funkcije.

   N – broj karaktera koji se iz stringa $S2$ kopira u string $S1$. Ukoliko ovim brojem nije obuhvaćen karakter za kraj stringa u okviru $S2$, ovaj karakter neće biti iskopiran. O ovome je potrebno voditi računa, jer ukoliko se kopiranjem izbriše karakter za kraj stringa $S1$, a ne iskopira se karakter za kraj stringa iz $S2$, prikaz rezultata i druge funkcije neće moći da odrede kraj stringa, pa će prikazivati sve karaktere do kraja deklarisanog niza karaktera.

4. **Povratna vrednost**

   Funkcija vraća pokazivač na prvi karakter odredišnog stringa.

Primer 6.21 (Funkcija *strncpy*)**.** Sledeći program ilustruje upotrebu funkcije *strncpy*.

```
1 #include "string.h"
2 void main()
3 {
4     char odr[80] = "Hello world!";
5     char izv[80] = "world");
6     char* rez;
7     rez = strncpy(odr,izv,2);
8 }
```

Rezultat izvršenja funkcije je

```
rez = odr = "wollo world!"
```

△

strcmp

Funkcija *strcmp* upoređuje sadržaj dva stringa. Poslednja tri slova u nazivu funkcije su skraćenica engleske reči ***comp**are* - uporediti.

1. **Deklaracija funkcije**

```
int strcmp ( const char* str1, const char* str2 );
```

2. **Opis dejstva**

   Funkcija upoređuje sadržaje stringova prenetih preko parametara i vraća informaciju o tome da li su stringovi jednaki, a ako nisu, informaciju o tome koji je string ispred koga u leksikografkom redosledu. Pod jednakošću se podrazumeva jednakost sadržaja stringova sve do karaktera za kraj stringa, bez obzira na to da li u deklaraciji stringovi imaju istu maksimalnu dužinu.

3. **Parametri**

   $str1$ **i** $st2$ – početne adrese stringova koji se porede. Pri pozivu funkcije za svaki od parametara funkcije moguće je navesti bilo promenljive ili literale. Ova funkcija ne menja sadržaj prenetih parametara (*const char**).

4. **Povratna vrednost**

   Funkcija vraća jednu od sledećih numeričkih vrednosti:

   < 0 – negativnu vrednost, ako je ASCII kod prvog različitog karaktera u stringovima ima manju vrednost u stringu $str1$ nego u stringu $str2$.

   0 – sadržaj stringova je identičan.

   > 0 – pozitivnu vrednost, ako je prvi različiti karakter u stringovima ima veću vrednost u stringu $str1$ nego u stringu $str2$.

Primer 6.22 (Funkcija *strcmp*)**.** S obzirom na to da funkcija vraća numeričku vrednost, imajući u vidu da su numeričke vrednosti u C-u istovremeno i logičke vrednosti, može se reći da funkcija vraća logičku vrednost *false* ako su stringovi jednaki, a vrednost *true*, ako nisu. Ova činjenica je iskorišćena u sledećem primeru, gde je povratna vrednost funkcije negirana unarnim operatorom '!', kako bi vrednost bila *true*, ako su stringovi jednaki.

Program određuje da li je vrednost stringa S jednaka zadatoj konstanti *Hello world*.

```
1 #include "string.h"
2 void main()
3 {
4     char S[] = "Hello world!";
5     if (!strcmp(S,"Hello world!")
6         printf("Stringovi su jednaki.");
7 }
```

△

strncmp

Funkcija *strncmp* upoređuje sadržaj prvih N karaktera dva stringa.

1. **Deklaracija funkcije**

```
int strncmp ( const char* str1,
    const char* str2, size_t N );
```

2. **Opis dejstva**

   Funkcija upoređuje prvih N karaktera stringova prenetih preko parametara i vraća informaciju o tome da li su jednaki, a ako nisu vraća informaciju o tome koji je string ispred koga u leksikografskom redosledu.

3. **Parametri**

   $str1$ **i** $st2$ – početna adrese stringova koji se porede.

   N – broj karaktera koji su uključeni u poređenje, računajući od početka prenetih stringova.

4. **Povratna vrednost**

   Funkcija vraća iste numeričke vrednosti kao i funkcija *strcmp*.

Primer 6.23 (Funkcija *strcmp*)**.** Sledeći program vrši pretragu i prikazuje koliko puta se u zadatoj rečenici javlja ”da”. Program je napisan tako da string x sadrži tekst u kom se traži, a prva dva karaktera stringa r sadrže traženu reč.

```
1  #include "stdio.h"
2  #include "string.h"
3  void main()
4  {
5      char x[] = "Da li mozete samostalno da napisete slican primer da trazi bilo 
       koju rec, a ne samo da?";
6      char r[] = "da ne";
7      int br = 0;
8      for (int i = 0; i < strlen(x) - 1; i++)
9          if ( !strncmp(x+i,r,2) )
10             br++;
11     printf("%d",br);
12 }
```

Funkciji *strncmp* u ovom primeru se u svakoj iteraciji *for* petlje za početak stringa prenosi sledeći karakter u odnosu na prethodnu iteraciju, tako da se prvo proveravaju prvi i drugi karakter, pa u narednoj iteraciji drugi i treći, pa treći i četvrti, itd. △

strcat

Operaciju konkatenacije stringova moguće je izvršiti korišćenjem funkcije *strcat* iz standardne biblioteke funkcija *string.h*. Poslednja tri slova u nazivu funkcije su skraćenica engleske reči *con**cat**enation* - nadovezivanje.

1. **Deklaracija funkcije**

   ```
   char* strcat( char* str1, const char* str2);
   ```

2. **Opis dejstva**

   Nadovezuje kopiju izvorišnog stringa $str2$ na odredišni string $str1$. Karakter koji označava kraj prvog stringa se gubi i preko njega se upisuje prvi karakter drugog stringa. Kopija sadržaja drugog stringa nadovezuje se na prvi string, zaključno sa oznakom za kraj drugog stringa. Za odredište i izvorište ne može se istovremeno prilikom poziva funkcije preneti isti string.

3. **Parametri**

   $str1$ – string na koji se nadovezuje sadržaj drugog stringa.

   $str2$ – string čiji se sadržaj nadovezuje na string $str1$.

4. **Povratna vrednost**

   Funkcija vraća adresu početka odredišnog stringa.

Primer 6.24 (Funkcija *strcmp*)**.** Sledeći program ilustruje upotrebu funkcije *strcat*.

```
1  #include "stdio.h"
2  #include "string.h"
3  void main()
4  {
5      char R[80];
6      strcpy(R,"Ovo␣");   // Inicijalno upisivanje
7      strcat(R,"je␣");    // nadoveivanje na prethodni sadrzaj...
8      strcat(R,"primer␣");
9      strcat(R,"nadovezivanja␣");
10     strcat(R,"stringova.");
11     printf("%s",R);
12 }
```

△

strncat

Funkcija *strncat* vrši nadovezivanje prvih N karaktera izvornišnog stringa na odredišni string.

1. **Deklaracija funkcije**

   ```
   char* strcat ( char* str1, const char* str2, size_t N);
   ```

2. **Opis dejstva**

   Ova funkcija nadovezuje prvih N karaktera izvorišnog stringa na kraj odredišnog stringa. Karakter koji označava kraj prvog stringa se gubi i preko njega se upisuje prvi karakter drugog stringa. Ukoliko karakter za označavanje kraja izvorišnog stringa nije uključen u prvih n karaktera ovog stringa, oznaka za kraj stringa neće biti nadovezana na odredišni string.

3. **Parametri**

   $str1$ – string na koji se nadovezuje sadržaj drugog stringa.

   $str2$ – string čiji se sadržaj nadovezuje na string $str1$.

   N – broj karaktera stringa $str2$ koji se nadovezuju na string $str1$.

4. **Povratna vrednost**

   Funkcija vraća adresu početka odredišnog stringa.

Primer 6.25 (Funkcija *strcmp*)**.** Sledeći program ilustruje upotrebu funkcije *strncat*.

```
1 #include <stdio.h>
2 #include <string.h>
3 void main ()
4 {
5     char R[20] = "To be ";
6     char S[20] = "or not to be";
7     strncat (R, S, 6);
8     printf("%s",R);
9 }
```

Ovaj program na izlazu prikazuje *"To be or not"*. △

strchr

Namena ove funkcije je traženje zadatog karaktera u stringu. Poslednja tri slova naziva funkcije su zadata kao asocijacija na namenu funkcije, a predstavljaju skraćenicu engleske reči ***chr****a****r****acter*.

1. **Deklaracija funkcije**

   ```
   char* strchr ( char* str, const char ch);
   ```

2. **Opis dejstva**

   Funkcija traži prvo pojavljivanje zadatog karaktera u zadatom stringu i vraća adresu na kojoj se zadati karakter nalazi.

3. **Parametri**

   str – string u kome se traži.

   ch – karakter čije se pojavljivanje traži u stringu. Prilikom poziva funkcije moguće je proslediti ili promenljivu, ili znakovnu (karakter) konstantu.

4. **Povratna vrednost**

   Funkcija vraća adresu na kojoj se zadati karakter nalazi. Ukoliko zadati string ne sadrži zadati karakter, funkcija će vratiti 0, odnosno *null*. Posmatrano kroz logički tip, ukoliko se zadati karakter ne pronađe, funkcija će vratiti logičku vrednost *false*, a *true* (bilo šta različito od 0), ukoliko string sadrži traženi karakter.

   Strategija vraćanja adrese iz funkcije, umesto indeksa pronađenog karaktera u nizu, daje određenu fleksibilnost pri upotrebi funkcije. Na primer, po tipu podataka, formalno gledano, povratna vrednost je string (*char**). Ukoliko se vrednost koju funkcija vrati prikaže kao string, biće prikazan deo stringa od pronađenog karaktera pa do kraja stringa u kom se vršila pretraga.

   Da bi se dobio indeks gde se traženi karakter nalazi, potrebno je oduzeti adresu rezultata od adrese početka stringa. Kako je veličina podatka tipa *char* 1 bajt, razlika adresa jednaka je indeksu pronađenog karaktera.

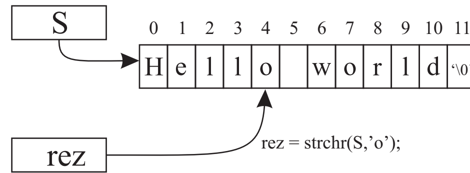
Primer 6.26 (Funkcija *strchr*)**.** Sledeći program ilustruje upotrebu funkcije *strchr* na primeru traženja prvog pojavljivanja slova 'o' u stringu "*Hello world*".

```
1  #include "stdio.h"
2  #include "string.h"
3  void main ()
4  {
5      char S[] = "Hello␣world";
6      char* rez;
7      short ind;
8      rez = strchr(S, 'o');   // trazenje karaktera 'o' u stringu S
9      printf("Rezultijuci␣string␣je:␣\"%s\"\n", rez);
10     ind = rez - S;          // odre\dj ivanje indeksa
11     printf("Indeks␣gde␣je␣pronadjen␣karakter␣je:␣%d", ind);
12 }
```

Situacija u memoriji nakon poziva funkcije *strchr* u 8. liniji programa prikazana je na slici 6.10.

Program na izlazu prikazuje:

```
Rezultijuci string je: "o world"
Indeks gde je pronadjen karakter je: 4
```



Slika 6.10: Povratna vrednost funkcije *strchr*

Napomena: Ukoliko je potrebno pronaći sledeće pojavljivanje zadatog karaktera, funkciju je potrebno pozvati tako da traži počev od narednog karaktera u odnosu na karakter gde je poslednji put pronađen zadati karakter. Na prethodnom primeru ovaj poziv bi bio: `rez = strchr(rez+1,'o');`

△

strstr

Funkcija *strstr* pronalazi i vraća poziciju počev od koje se sadržaj jednog stringa pojavljuje u drugom stringu. Poslednja tri slova naziva funkcije su zadata kao asocijacija na namenu funkcije. Funkcija je implementacija funkcionalnosti koju poseduju gotovo sve aplikacije za obradu teksta: pretraga i pronalaženje pozicije zadatog niza simbola u većem tekstu.

1. **Deklaracija funkcije**

   ```
   char* strstr ( char* tekst, const char* str);
   ```

2. **Opis dejstva**

   Funkcija traži prvo pojavljivanje vrednosti zadatog stringa u tekstu i vraća adresu počev od koje se zadati string nalazi.

3. **Parametri**

   tekst – tekst u kome se traži. Po tipu ova promenljiva je string. Obično se prilikom poziva u ovu promenljivu prenosi string koji je duži od niza simbola koji se traži, da bi pretraga imala smisla.

   str – string koji sadrži vrednost koja se traži. Prilikom poziva funkcije moguće je ovom parametru proslediti promenljivu ili literal.

4. **Povratna vrednost**

   Funkcija vraća adresu počev od koje se vrednost zadatog stringa nalazi u tekstu. Ukoliko se vrednost zadatog stringa ne pronađe u tekstu u kom se vrši pretraga, funkcija vraća vrednost 0. Kao i kod funkcije *strchr*, ova vrednost se može posmatrati i kao logički tip, pa se može reći da će funkcija vratiti *false* ako zadata vrednost nije pronađena.

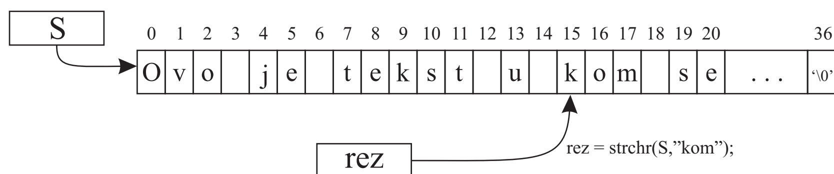
Strategija vraćanja adrese iz funkcije ista je kao i strategija vraćanja adrese iz funkcije *strchr*.

Primer 6.27 (Funkcija *strstr*)**.** Sledeći program ilustruje upotrebu funkcije *strstr*.

```
1  #include "stdio.h"
2  #include "string.h"
3  void main ()
4  {
5      char S[] = "Ovo je tekst u kom se vrsi pretraga";
6      char* rez;
7      if( rez = strstr(S, "kom") )
8         printf("Rezultijuci string je: \"%s\"\n", rez);
9      else
10        printf("Tekst ne sadrzi zadatu rec");
11 }
```

U ovom primeru iskorišćena je fleksibilnost programskog jezika C da se u uslovu *if*-a može naći i izraz. Naime, u uslovu u 6. liniji naveden je operator dodele '=', a ne relacioni operator za poređenje '==', što je izuzetno značajna razlika u ovom primeru. Zbog toga će se prvo pozvati funkcija *strstr*, i s obzirom na operator dodele, povratna vrednost iz funkcije će se upisati u promenljivu *rez*. Nakon dodele se vrednosti ove promenljive uzima kao uslov *if*-a, i ako je jednaka 0, smatra se da uslov nije ispunjen, a ako je različita od 0, da je uslov ispunjen.

Situacija u memoriji nakon poziva funkcije *strstr* u 6. liniji programa, za konkretne vrednosti iz primera, prikazana je na slici 6.11.



Slika 6.11: Povratna vrednost funkcije *strstr*

Program na izlazu prikazuje:

```
Rezultijuci string je: "kom se vrsi pretraga"
```

△

Primer 6.28 (Određivanje svih pozicija na kojima se zadati niz karaktera javlja u tekstu)**.** Ukoliko je potrebno pronaći sledeće pojavljivanje zadatog niza simbola, funkciju je potrebno pozvati tako da traži počev od narednog karaktera u odnosu na karakter gde je poslednji put pronađen zadati karakter, ako ne i od karaktera za dužinu traženog stringa dalje.

Sva pojavljivanja zadatog stringa moguće je odrediti na sledeći način:

```
1  #include "stdio.h"
2  #include "string.h"
3  void main()
4  {
5      char x[] = "da li mozete samostalno da napisete slican primer da trazi bilo 
       koju rec, a ne samo da?";
6      char r[] = "da";
7      char* rez = x-1;
8      while ( rez = strstr(rez+1,r) )
9          printf("Indeks: %d\n", rez - x);
10 }
```

Ovaj program na izlazu daje:

```
Indeks: 0
Indeks: 24
Indeks: 50
Indeks: 84
```

Suštinski deo zadatka rešen je u ovom primeru dvema linijama koda (8. i 9.).

U uslovu *while* petlje je izraz, koji rezultat poziva funkcije *strstr* dodeljuje promenljivoj *res*. Funkcija traži počev od narednog mesta gde je zadati string poslednji put pronađen ($rez+1$), a rezultat pretrage ponovo upisuje u promenljivu rez. Ukoliko je rezultat različit od nule, to znači da je tekst ponovo pronađen, pa se u 9. liniji koda prikazuje indeks početka ($rez-x$). Kako je u ovom slučaju uslov petlje zadovoljen (različit od nule, *true*), petlja se ponovo izvršava. Izvršenje će se prekinuti kada funkcija vrati 0 (*false*), što znači da u ostatku teksta zadati string nije pronađen.

Da bi se i u prvom prolasku kroz petlju počelo od prvog karaktera teksta u kom se traži, promenljiva rez je inicijalizovana na vrednost $x-1$, što je odmah u prvom pozivu funkcije kompenzovano izrazom $rez+1$.

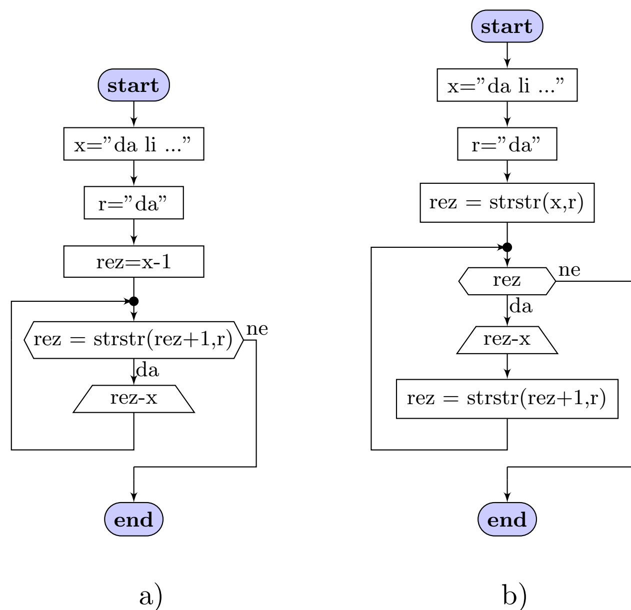
Ilustracije radi, dijagram toka algoritma ovog programa prikazan je na slici 6.12a. Na slici slici 6.12b prikazan je dijagram toka ekvivalentnog algoritma bez korišćenja mogućnosti da se u programskom jeziku C u uslovu *while* petlje može naći i izraz sa dodelom vrednosti. △

strtok

Funkcija *strtok* može se koristiti za izdvajanje reči iz rečenice. Funkciji se preko parametara prenosi string sa tekstom koji je potrebno razdvojiti na reči, kao i skup simbola koji se očekuju za delimitere reči. Funkciju je potrebno pozivati više puta, a prilikom svakog poziva funkcija vraća po jednu reč.

U kontekstu progamiranja, reči, odnosno elementarni nizovi simbola koji imaju semantički smisao, nazivaju se tokenima, pa otuda i skraćenica *tok* u nazivu funkcije.

1. **Deklaracija funkcije**



Slika 6.12: Dijagram toka algoritma za traženje pojavljivanja zadate reči u tekstu korišćenjem funkcije *strstr*

```
char* strtok ( char* str, const char* delimiteri );
```

2. **Opis dejstva**

   Uzastopni pozivi ove funkcije dele string prenet preko parametara na niz reči. Prvi poziv vraća prvu reč, drugi poziv drugu, itd. Karakteri koji mogu biti delimiteri reči, kao što je razmak ili znakovi interpunkcije, navode se kao poseban parametar. Funkcija odvaja reči tražeći neki od prenetih delimitera za granice reči.

3. **Parametri**

   str – string po tipu. Ovaj parametar sadrži tekst koji se razdvaja na reči.

   Ovaj parametar je ključan za princip rada funkcije. Naime, ukoliko je pri pozivu preneta vrednost različita od *null*, odnosno neki tekst, funkcija će vratiti prvu reč iz teksta. Ukoliko se funkcija pozove tako da se za ovaj parametar navede *null*, tada funkcija vraća narednu reč u odnosu na reč izdvojenu prilikom prethodnog poziva funkcije.

   Ova funkcionalnost je u samoj funkciji implementirana korišćenjem statičkih promenljivih, koje po prirodi ne gube sadržaj nakon završetka funkcije (poglavlje 6.5.3). Ukoliko je parametar različit od *null*, funkcija

pronalazi prvu reč i string pamti kao statički parametar. U sledećem pozivu funkcija će koristiti lokalne statičke promenljive kako bi mogla da nastavi sa pretragom dokle je stigla u prethodnom pozivu.

delimiteri – takođe string po tipu. Sadrži skup karaktera koji mogu biti delimiteri reči. Redosled karaktera-delimitera u ovom stringu nije bitan.

4. **Povratna vrednost**

   Funkcija vraća adresu početka pronađene reči.

Primer 6.29 (Funkcija *strtok*)**.** Sledeći program vrši razdvajanje zadatog stringa na reči korišćenjem funkcije *strtok*.

```
1  #include "stdio.h"
2  #include "string.h"
3  void main ()
4  {
5      char S[] = "Ovo␣je␣tekst␣iz␣koga␣cemo␣izvuci␣i␣prikazati␣sve␣reci.";
6      char del[] = "␣.,!?";          // moguci delimiteri
7      char* rec = strtok(S,del);     // izdvoji prvu rec...
8      while (rec)
9      {
10         printf("%s\n",rec);
11         rec = strtok(NULL, del);   // pa sledecu rec..., itd.
12     }
13 }
```

Konstanta NULL, koja je upotrebljena u 11. liniji programa, definisana je u biblioteci "*stdio.h*" kao

```
#define NULL 0
```

Program prikazuje sledeće vrednosti:

```
Ovo
je
tekst
iz
koga
cemo
izvuci
i
prikazati
sve
reci
```

△

6.6.3 Funkcije za dinamičku alokaciju memorije

Deklaracijom promenljive kompajleru se signalizira da je potrebno da izvrši rezervaciju prostora u memoriji za smeštanje vrednosti promenljive. Tip promenljive naveden u deklaraciji daje opis kakav taj prostor treba biti, i određuje način smeštanja promenljive. Na primer:

```
int x,y,z;
```

Broj i tip statički navedenih deklaracija nije moguće menjati u toku izvršenja programa. Što se polja tiče, statička deklaracija polja zahteva obavezno navođenje konstanti za dimenzije polja. Ove dimenzije nije moguće zadati parametarski promenljivama. Na primer, statička deklaracija jednodimenzionalnog polja sa 100 elemata kome je dodeljen identifikator A je:

```
int A[100];
```

Međutim, za veliku klasu problema poželjno je posedovati mehanizam koji će omogućiti rezervaciju novog memorijskog prostora željene veličine bilo gde u programu, ali tako da se veličina prostora precizira tek u toku izvršenja programa.

Definicija 6.6 (Dinamička alokacija memorije)**.** Dinamička alokacija memorije je akcija rezervacije memorijskog prostora, koja se obavlja u toku izvršenja programa, ali tako da se veličina potrebne memorije određuje u toku izvršenja programa, a ne u toku pisanja programa.

Kao jedan primer programa gde je poželjna dinamička alokacija može se navesti program za pamćenje telefonskog imenika. Kod ovakve aplikacije postavlja se pitanje potrebe korisnika za memorijskim prostorom. Ukoliko programer statički definiše mogućnost pamćenja npr. 1000 brojeva, nekom korisniku će ovaj broj biti mali, a za nekoga će biti bespotrebno trošenje memorijskog prostora. U današnje vreme sa raspoloživim kapacitetima memorije, za ovaj primer teško je reći da je 1000 brojeva preveliko rasipanje prostora. Veći je problem ako korisnik ima potrebu za većim brojem podataka. Kod statičke alokacije nema drugog načina nego da se program vrati programeru "na doradu", kako bi promenio deklaraciju i ponovo kompajlirao program.

Ključna razlika između statičke i dinamičke alokacije je u tome da se kod statičke alokacije veličina potrebnog prostora definiše u fazi pisanja programa, a kod dinamičke se u toku izvršenja programa može proizvoljno zauzimati memorijski prostor po potrebi, pa su ovakvi programi fleksibilniji i ekonomičniji.

Programski jezik C ima podršku za dinamičku alokaciju memorije kroz standardne funkcije biblioteka *"malloc.h"* i *"stdlib.h"*. Dinamička alokacija memorije vrši se pomoću funkcija:

1. `malloc` i `calloc`,
2. `realloc`, i
3. `free`

Pomoću ovih funkcija moguće je u toku izvršenja programa od operativnog sistema zatražiti rezervaciju (ili oslobađanje) potrebne količine memorije.

Funkcija malloc

Funkcija *malloc* naziv je dobila kao skraćenica engleskih reči ***m**emory **alloc**ation* i može se naći u bibliotekama *stdlib.h* i *malloc.h*. Namena funkcije je dinamička rezervacija memorijskog prostora željene veličine. Veličina prostora može se zadati parametarski i definisati u toku samog izvršenja programa.

1. **Deklaracija funkcije**

   ```
   void* malloc(size_t velicina)
   ```

2. **Opis dejstva**

   Funkcija vrši dinamičku alokaciju memorije i vraća pokazivač na početak dodeljenog memorijskog prostora. Rezervisanom prostoru moguće je pristupiti jedino korišćenjem pokazivača koji funkcija vraća.

3. **Parametri**

   velicina – Parametar funkcije je veličina zahtevanog memorijskog prostora u bajtovima. Tip parametra je *size_t*.

4. **Povratna vrednost**

   Ukoliko je alokacija uspešna, funkcija vraća pokazivač na prvu adresu dodeljenog memorijskog prostora. U slučaju neuspešne alokacije funkcija vraća *null*, t.j. numeričku vrednost 0.

   Kako bi bilo moguće ovu funkciju koirstiti za alokaciju prostora za bilo koji tip promenljive, funkcija nakon alokacije prostora vraća pokazivač koji je tipa *void**, za šta se u ovom slučaju može reći da je neodređeni tip. Zbog ovoga je pre pristupa memorijskom prostoru potrebno odrediti tip. Tip se određuje tako što se povratna vrednost iz funkcije *cast* operatorom eksplicitno konvertuje i upiše u pokazivač konkretnog tipa. Na primer:

   ```
   int* p;
   p = (int*) malloc(4);
   ```

Primer 6.30 (Dinamička alokacija memorije funkcijom *malloc*)**.** Sledeći program ilustruje dinamičku alokaciju memorije funkcijom *malloc*.

```
1 #include <malloc.h>
2 main()
3 {
4     int *p, *q, N;
5     p=(int*)malloc(4);
6     q=(int*)malloc(sizeof(int));
7     *p=10;
8     *q=*p+50;
9 }
```

U 5. liniji programa rezervisan je prostor od 4 bajta, a pokazivač na ovaj prostor je konvertovan u pokazivač na tip *int*, i dodeljen je pokazivaču *p*, prko koga će se moći pristupati ovom prostoru. Isto je urađeno u 6. liniji gde je veličina rezervisanog prostora jednaka veličini tipa *int*. Ovo rešenje je generalnije, jer ne zavisi od implementacije kompajlera na konkretnom procesoru, i kod svih prethodnih i budućih kompajlera će se u ovom slučaju rezervisati prostor dovoljan za podatak tipa *int*.

U 7. liniji se u novoalocirani prostor na koji ukazuje pokazivač *p* upisuje konstanta 10, a u 8. liniji u prostor na koji ukazuje *q* upisuje se vrednost iz lokacije na koju ukazuje *p*, uvećana za 50. Kao što je već rečeno, rezervisanom memorijskom prostoru moguće je jedino pristupiti korišćenjem pokazivača i operatora dereferenciranja.

Napomena: Dobra praksa je da se nakon alokacije prostora, a pre njegovog korišćena proveri da li je alokacija uspešno obavljena. Ako nije, odnosno ako je funkcija vratila vrednost 0, ne treba pokušavati upis, jer će ovakav upis izazvati grešku u toku izvršenja (eng. *run-time error*) zbog pokušaja pristupa memorijskom prostoru koji nije "u vlasništvu" aplikacije.

Proveru da li je alokacija uspešno obavljena pre pristupa memoriji moguće je obaviti na sledeći način.

```
1    ...
2    if ( p=(int*)malloc(4) )
3    {
4        // uspesno
5        ...
6    }
7    else
8    {
9        // neuspesno
10       ...
11   }
```

I u ovom primeru je iskorišćena mogućnost pisanja izraza sa dodelom vrednosti u uslovu alternacije. △

Funkcija calloc

Funkcija *calloc* ima isto dejstvo kao i funkcija *malloc* - vrši dinamičku rezervaciju memorijskog prostora željene veličine.

Veličina prostora može se zadati parametarski i definisati u toku samog izvršenja programa. Specifičnost ove funkcije je ta da se veličina željenog prostora zadaje tako što se zada veličina jednog bloka podataka, i uz to navede i željeni broj takvih blokova. Za razliku od *malloc*, funkcija *calloc* upisuje nule, t.j. briše prethodni sadržaj memorije pre nego što vrati pokazivač na početak alociranog prostora.

Imajući u vidu da u originalnoj dokumentaciji nije navedeno, smatra se da je slovo *'c'* u nazivu funkcije početno slovo ili od engleske reči *count* (prev. *broj*), ili

clear (prev. *brisanje*), s obzirom na to da se kod funkcije *calloc* zadaje broj blokova, ali se isto tako ovaj prostor popunjava nulama, t.j. briše. Obe ove karakteristike su atipične u odnosu na *malloc*. Kako je ovo pitanje često među C programerima, jednom prilikom se i sam Brajan Kerninghan izjasnio o ovom pitanju i izjavio da on nije pisao tu funkciju, ali da s obzirom na nedostatak dokumentacije smatra da je *'c'* u nazivu funkcije od *clear*.

1. **Deklaracija funkcije**

   ```
   void* calloc(size_t num, size_t velicina)
   ```

2. **Opis dejstva**

   Funkcija vrši dinamičku alokaciju memorije, ukupne veličine $num \cdot velicina$, gde je *num* broj blokova, a *velicina* veličina svakog bloka, i vraća pokazivač na početak dodeljenog memorijskog prostora. Funkcija inicijalizuje sadržaj memorijskog prostora u toku alokacije na 0.

   S obzirom na mogućnost zadavanja broja i veličine blokova ova funkcija se često koristi za dinamičku alokaciju nizova.

3. **Parametri**

   num – Ovaj parametar predstavlja ukupan broj blokova, a zadaje se kao neoznačeni ceo broj tipa *size_t*.

   velicina – Neoznačeni ceo broj, kojim se zadaje veličina jednog elementa u memorijskom prostoru. Veličina se zadaje u bajtovima. Tip parametra je *size_t*.

4. **Povratna vrednost**

   Ukoliko je alokacija uspešna, funkcija vraća pokazivač na prvu adresu dodeljenog memorijskog prostora. U slučaju neuspešne alokacije funkcija vraća *null*, t.j. numeričku vrednost 0. Kako bi bilo moguće definisati format pristupa ovom memorijskom prostoru, odnosno tip promenljivih koje će se upisivati, potrebno je povratnu vrednost iz funkcije *cast* operatorom eksplicitno konvertovati u željeni tip. Na primer:

   ```
   int* p;
   int N;
   ...
   p = (int*) calloc(N, sizeof(int));
   ```

Primer 6.31 (Dinamička alokacija memorije za niz podataka)**.** U ovom primeru ilustrovana je dinamička alokacija niza podataka. Naime, program je napisan tako da korisniku dozvoljava unos ukupnog broja elemenata niza (N), a zatim dinamički rezerviše prostor za ove elemente.

Ukoliko je A pokazivač, što je po tipu isto kao i identifikator niza, imajući u vidu relaciju $A[i] \equiv *(A + i)$, pristup blokovima rezervisanog memorijskog prostora moguće je izvesti kao klasični pristup elementima statičkog niza $A[i]$. Drugim rečima, s obzirom na prethodno navedeni identitet, bez obzira da li je niz deklarisan statički ili dinamički, elementima je moguće pristupati na isti način.

```
1  #include <malloc.h>
2  main()
3  {
4      int *A;
5      int N;
6      scanf("%d",&N);
7      A=(int*)calloc(N, sizeof(int)); // dinam. alokacija niza sa N elemenata
8
9      A[0]=1;
10     A[1]=2;
11     A[2]=3;
12     // ...
13 }
```

△

Bez obzira na to što je memorijski prostor koji alocira funkcija *calloc* jednak $num \cdot velicina$ bajtova, alocirani memorijski blokovi su u sukcesivnim memorijskim lokacijama i ne postoji nikakva fizička podela na blokove. Adresiranje pojedinih blokova je samo stvar pokazivačke aritmetike, kako je to pokazano u prethodnom primeru. Imajući ovo u vidu, i funkcija *malloc* se može na ravnopravan način koristiti za dinamičku alokaciju niza podataka na sledeći način:

```
1  #include <malloc.h>
2  main()
3  {
4      int *A;
5      int N;
6      scanf("%d",&N);
7      A=(int*)malloc(N*sizeof(int)); // dinam. alokacija niza sa N elemenata
8
9      A[0]=1;
10     // ...
11 }
```

Jedina razlika je što je parametar koji zahteva funkcija *malloc* naveden kao proizvod $num \cdot velicina$, a ne kao dva nezavisna parametra, što ne menja ishod. Naravno, ostaje razlika da funkcija *malloc* neće inicijalizovati alocirani memorijski prostor.

Sledeći primer ilustruje upotrebu funkcije *calloc* za dinamičku alokaciju prostora za niz struktura.

Primer 6.32 (Dinamička alokacija niza struktura)**.** U ovom primeru je od 3. do 7. linije koda definisana struktura podataka za pamćenje vrednosti kompleksnih brojeva, a u 15. liniji je izvršena dinamička alokacija niza od N ovakvih podataka. Ako je alokacija uspešna, u 15. liniji koda će pokazivaču p biti dodeljena vrednost različita od 0, a kako je ovaj pokazivač istovremeno naveden i kao uslov alternacije, u zavisnosti od ishoda funkcije *calloc* izvršiće se odgovarajuća grana alternacije.

```
1  #include "stdio.h"
2  #include "malloc.h"
3  struct kompleksni
4  {
5      float Re;
6      float Im;
7  };
8  void main()
9  {
10     int N;
11     printf("Unesite broj kompleksnih brojeva: ");
12     scanf("%d",&N);
13
14     kompleksni* p;
15     if ( p = (kompleksni*) calloc(N, sizeof(kompleksni)) )
16     {
17         p[0].Re = 1.0;
18         p[0].Im = 2.0;
19
20         p[1].Re = 3.2;
21         p[1].Im = 4.1;
22
23         printf("Zbir realnih delova prva dva broja je: %f", p[0].Re + p[1].Re);
24     }
25     else
26         printf("Neuspesna alokacija");
27 }
```

△

Funkcija realloc

Ova funkcija menja veličinu alociranog memorijskog prostora na koji ukazuje pokazivač naveden kao parametar funkcije.

1. **Deklaracija funkcije**

```
void* realloc (void* ptr, size_t nova_velicina);
```

2. **Opis dejstva**

Funkcija vrši dinamičku promenu veličine prethodno alociranog memorijskog prostora na koji ukazuje pokazivač *ptr*. Prostor se može povećati ili smanjiti. Veličina memorijskog prostora, ukoliko je funkcija uspešno izvršena, biće jednaka parametru *nova_velicina*.

U cilju promene veličine alociranog memorijskog prostora ova funkcija može premestiti ceo blok podataka na drugo mesto u memoriji. Prilikom realokacije sadržaj lokacija se ne gubi.

3. **Parametri**

ptr – Pokazivač na memorijski prostor čija se realokacija vrši. Parametar je tipa *void**, tako da je pri pozivu funkcije moguće proslediti parametar bilo kog pokazivačkog tipa.

Ukoliko se ovom parametru pri pozivu funkcije prosledu konstanta NULL, dejstvo ove funkcije identično je dejstvu funkcije *malloc*.

nova_velicina – Nova veličina memorijskog prostora u bajtovima. Može biti manja ili veća od trenutne veličine alociranog prostora. Tip parametra je *size_t*.

4. **Povratna vrednost**

Ukoliko je alokacija uspešna, funkcija vraća pokazivač na prvu adresu memorijskog prostora.

Primer 6.33 (Primer dinamičke promene veličine niza)**.** Sledeći program ilustruje promenu broja elemenata već alociranog niza u toku izvršenja. Niz se sastoji od N celih brojeva tipa *int*, za koji je prostor u memoriji alociran u 8. liniji programa. U 10. liniji izvršena je realokacija tako da je u novi prostor moguće upisati 10 elemenata više.

```
1  #include "malloc.h"
2  #include "stdio.h"
3  void main()
4  {
5      int *A;
6      int N;
7      scanf("%d",&N);
8      A=(int*)malloc(N*sizeof(int)); // dinam. alokacija niza sa N elemenata
9
10     A = (int*) realloc (A, (N+10)*sizeof(int));  //realokacija
11
12     A[0]=1;
13     // ...
14 }
```

△

Funkcija free

Memorijski blok prethodno alociran pozivom funkcije *malloc*, *calloc*, ili *realloc* se pozivom funkcije *free* oslobađa.

1. **Deklaracija funkcije**

   ```
   void free (void* ptr);
   ```

2. **Opis dejstva**

   Funkcija oslobađa prethodno alocirani memorijski prostor. Oslobođeni memorijski prostor može kasnije u programu biti ponovo alociran, ili može biti alociran od strane drugih aplikacija, ako se program izvršava na *multi-tasking* operativnom sistemu.

3. **Parametri**

   ptr – Početna adresa memorijskog prostora koji se oslobađa. Prilikom poziva funkcije može se navesti pokazivač bilo kog tipa.

4. **Povratna vrednost**

   Ova funkcija nema povratnu vrednost.

Primer 6.34 (Primer oslobađanja dinamički zauzetog memorijskog prostora)**.** Sledeći primer ilustruje upotrebu funkcije *free*.

```
1 #include <malloc.h>
2 main()
3 {
4     int *A, n;
5     scanf("%d", &n);
6     A=(int*)calloc(n, sizeof(int)); // akolacija
7     A[0]=1;
8     free(A);        // oslobadjanje prostora
9 }
```

△

6.7 Standardni ulaz/izlaz i rad sa fajlovima

Skoro da nema programskog jezika koji na nekom nivou ne nudi podršku za rad sa podacima zapamćenim u fajlovima na disku. Fajlovi se koriste za čuvanje podataka na nekoj vrsti eksternog medijuma, koji ne gube sadržaj i nakon nestanka napajanja.

Podrška za rad sa fajlovima u okviru programa može biti:

1. podrška na nivou toka podataka, i

2. podrška na niskom (sistemskom) nivou.

Podrška na nivou toka podataka (tok na engleskom - *stream*) predstavlja apstrakciju pristupa podacima zapamćenim u fajlovima, tako da su sami mehanizmi za pristup podacima sakriveni od korisnika. Tok povezuje fajl i aplikaciju, tako da podaci mogu "teći" u bilo kom smeru, bilo od fajla ka aplikaciji (čitanje podataka), ili od aplikacije ka fajlu (upis podataka).

Kod pristupa fajlovima na niskom nivou sam pristup je znatno fleksibilniji, ali je neophodno poznavati mehanizme niskog nivoa za pristup fajlovima. Na sistemskom nivou postoji i podrška za manipulaciju samim fajlovima. Ova podrška je u vidu skupa funkcija, koje izvršavaju operacije dostupne iz operativnog sistema: promena imena, kopiranje, brisanje, listanje sadržaja direktorijuma, i sl.

Podršku na niskom, sistemskom nivou i podršku za manipulaciju samim fajlovima obezbeđuje operativni sistem, a programeru je skup funkcija koje omogućavaju ovu podršku dostupan kroz skup takozvanih sistemskih funkcija koji se naziva API (eng. *Application Programming Interface*). API sadrži funkcije čijim se pozivom od operativnog sistema zahteva izvršenje željene akcije. Ove funkcije se nalaze u posebnim bibliotekama koje je za tu priliku potrebno uključiti u program. Što se API interfejsa tiče, pored funkcija za rad sa fajlovima, kroz API je programeru dostupan i veliki broj funkcija različite namene, kao što su npr. funkcije za slanje i prijem paketa kroz računarsku mrežu, i mnoge druge. Deo AIP interfejsa za rad sa fajlovima može se naći pod nazivom *"File System Interface"*.

U daljem tekstu zadržaćemo se na pristupu podacima smeštenim u fajlovima na disku na nivou tokova podataka.

Tok podataka je put kojim se prenose podaci, a koji ima svoj izvor i svoje odredište. O toku podataka brine se operativni sistem, a iz ugla programera tok predstavlja medijum za prenos podataka između dva različita entiteta u operativnom sistemu, koji apstrahuje detalje implementacije samog mehanizma prenosa podataka.

Ovaj koncept je preuzet iz UNIX operativnog sistema, u okviru koga je implementirana sistemska podrška za tokove podataka, tako da se sam operativni sistem brine o njima i njihovom konkurentnom izvršenju. Koncept tokova podataka iz programskog jezika C zadržan je u mnogim drugim jezicima u istom obliku, ili sa malim razlikama (C++, PHP, i dr.). Tok podataka moguće je kreirati, definisati i vezati za izvor i odredište, kako bi se postigao željeni prenos podataka. Zbog ovoga, svaki tok podataka ima svoje ime, odnosno identifikator koji ga određuje.

Postoje standardni i korisnički tokovi podataka.

6.7.1 Standardni tokovi podataka

Standardni tokovi podataka su predefinisani, t.j. ne zahteva se od programera pisanje eksplicitnih definicija ovih tokova. Definicije standardnih tokova nalaze se u standardnoj biblioteci *"stdio.h"*. Standardni tokovi povezuju program sa nekim od standardnih uređaja (npr. monitor, tastatura, štampač, i sl.). Kaže se i da su ovi tokovi "reference na uređaje". Standardni tokovi su:

`stdin` – skraćenica od ***st**an**d**ard **in**put*, što u prevodu znači "standardni ulaz". Predstavlja standardni tok podataka sa tastature u program.

`stdout` – skraćenica od ***st**an**d**ard **out**put*, što u prevodu znači "standardni izlaz". Predstavlja standardni tok podataka iz programa na ekran monitora,

`stderr` – skraćenica od ***st**an**d**ard **err**or*. Predstavlja standardni tok podataka za prikaz grešaka. Koristi se radi razdvajanja izlaza za rezultate od izlaza za poruke o eventualnim greškama u programu. Ukoliko ovaj tok nije preusmeren, greške će biti prikazivane na ekranu. Mnoge bibliotečke funkcije šalju poruke o eventualnim greškama na ovaj tok, mada i sam programer može slati poruke iz svog programa na njega. Kada se rezultati izvršenja programa štampaju na štampaču, korisno je poruke o greškama preusmeriti na ovaj tok, kako bi bile prikazivane na monitoru, umesto da se i one štampaju.

`stdprn` – skraćenica od ***st**an**d**ard **pr**i**n**t*. Predstavlja standardni tok podataka iz programa na štampač.

`stdaux` – skraćenica od ***st**an**d**ard **aux**iliary*[2]. Predstavlja standardni tok podataka iz programa na serijski port računara.

Može se slobodno reći da C kompajleri za sve platforme podržavaju i imaju implementirane standardne tokove *stdin*, *stdout* i *stderr*, što nije slučaj sa *stdprn* i *stdaux*. Tok *stdaux* češće se sreće kod C kompajlera za industrijske mikrokontrolere.

Korišćenje tokova podataka u C-u je jednostavno, i implementirano je u osnovi kroz nekoliko funkcija, od kojih su dve funkcije *fprintf* i *fscanf*, iz biblioteke "*stdio.h*". Za razliku od funkcija *printf* i *scanf*, koje imaju po dva parametra, funkcije *fprintf* i *fscanf* imaju jedan dodatni parametar koji određuje tok podataka preko koga će podaci biti poslati, odnosno preko koga će podaci biti primljeni.

Primer 6.35 (Korišćenje standardnih tokova)**.** Sledeći program ilustruje upotrebu tokova u funkcijama *fprintf* i *fscanf*. Obe funkcije imaju tri parametra: prvi parametar je identifikator toka, a druga dva parametra su format i eventualno promenljive, odnosno adrese promenljivih kao kod funkcija *printf* i *scanf*.

```
1 #include "stdio.h"
2 main()
3 {
4     int x;
5     fscanf(stdin,"%d",&x);
6     fprintf(stdout,"%d",x);
7 }
```

Ovaj program sa standardnog ulaznog toka, odnosno sa tastature prihvata vrednost i upisuje je u promenljivu x, a odmah nakon toga upisanu vrednost šalje na standardni izlazni tok, odnosno prikazuje na ekranu monitora. △

[2]Eng. *auxiliary* - prev. pomoćni, dodatni, drugi; često se koristi za imenovanje dodatnih portova na uređajima.

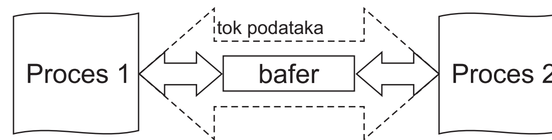
Može se reći da su funkcije *printf* i *scanf* restrikcija funkcija *fprintf* i *fscanf*, jer podatke isključivo prenose tokovima *stdout* i *stdin*.

Baferovanje tokova

Tokovi podataka mogu biti baferovani i nebaferovani.

Definicija 6.7 (Bafer)**.** Bafer[3] je memorija, organizovana kao linearna struktura tipa red (FIFO), koja se postavlja između dva uređaja ili procesa sa različitim brzinama, kako sporiji uređaj ne bi usporavao brži.

Pozicija bafera prikazna ja na slici 6.13.



Slika 6.13: Pozicija bafera za prilagođavanje brzina procesa

Ukoliko je tok podataka između programa i nekog uređaja baferovan, to znači da će podaci najvećom mogućom brzinom iz programa biti upisani u bafer, a nakon toga će iz bafera jedan po jedan biti prosleđivani uređaju. Ovo se može generalizovati i reći da je bolje imati baferovanu komunikaciju uvek kada uređaj sa kojim se komunicira ne zahteva interaktivnost. Standardni tokovi *stdin*, *stdout*, *stdprn* i *stdaux* su baferovani uvek kada se ne komunicira sa interaktivnim uređajem. Standardni tok *stderr* nije u potpunosti baferovan.

Zbog bafera koji postoji kod standardnoog ulaza, podaci koje zahteva funkcija *scanf* unose se sa tastature brzinom kojom korisnik može da ih unese i privremeno se smeštaju u bafer. Za to vreme funkcija *scanf* je blokirana. Tek kada korisnik pritisne taster ⟨*enter*⟩, svi podaci iz bafera se odjednom prosleđuju funkciji *scanf*, nakon čega ih ona "raspoređuje" u zadate promenljive po zadatom formatu.

Bafer može biti implementiran strukturom podataka u operativnoj memoriji, ili hardverski u okviru samog uređaja, kao na primer kod hard diska. Pristup fajlovima na disku uvek je baferovan. Kako se izvršenje instrukcija programa i pristup operativnoj memoriji meri u nanosekundama, a pristup hard disku se izražava u milisekundama, ovu razliku od nekoliko redova veličine kompenzuje bafer. Algoritmi za pribavljanje podataka sa diska u bafer čak i sa određenim stepenom uspešnosti predviđaju koji će sledeći podatak biti potreban, da bi ga pribavili unapred i time što je više moguće kompenzovali relativno malu brzinu pristupa disku. Ovaj postupak se naziva *"pribavljanje unapred"*, ili keširanje (eng. *cache*). Generalno pravilo, bez obzira na algoritam za keširanje, u ovom slučaju je da se bolje performanse dobijaju upotrebom većeg bafera. Isto važi i za upis: veći bafer će

[3] Eng. *buffer*, prev. *prihvatna memorija, pomoćna memorija*

omogućiti programu da isporuči disku veću količinu podataka, pre nego što bude trebalo da sačeka da ovi podaci budu i fizički upisani.

Apstrakcija pristupa fajlovima i uređajima na nivou toka podataka je na visokom nivou, pa tako programer u svom programu ne mora voditi računa o postojanju i radu bafera.

Preusmeravanje tokova

Standardni tokovi se mogu preusmeriti.

Standardni izlaz se može preusmeriti tako da se umesto prikaza podataka na ekranu podaci upisuju u fajl. Standardni ulaz se može preusmeriti tako da se podaci umesto sa tastature učitavaju iz fajla. Funkciju preusmeravanja toka podataka vrši operativni sistem.

Preusmeravanje se može izvršiti iz konzole operativnog sistema[4] sledećim binarnim operatorima:

> – standardni izlaz preusmerava u fajl. Prethodni sadržaj fajla se briše.

>> – standardni izlaz preusmerava u fajl. Prethodni sadržaj fajla se ne briše, već se novi podaci upisuju na kraj postojećeg fajla.

< – čita podatke iz navedenog fajla i preusmerava ih na standardni ulaz.

Kod sva tri operatora je izvršni fajl programa koji se pokreće levi operand, a fajl iz koga se čita, odnosno u koji se upisuje je desni operand:

$$\langle stdout_u_fajl\rangle \quad ::= \quad \langle izvrsni_fajl\rangle\, ('>' \,|'\, >>')\, \langle fajl_za_upis\rangle$$

$$\langle stdin_iz_fajla\rangle \quad ::= \quad \langle izvrsni_fajl\rangle'<'\, \langle fajl_za_citanje\rangle$$

Primer 6.36 (Preuzmeravanje standardnih tokova poradata)**.** Sledeći program sumira elemente matrice, a napisan je tako da koristi standardni ulaz i izlaz.

```
1  #include "stdio.h"
2  main()
3  {
4      int A[100][100], N, M, i, j, S=0;
5      scanf("%d%d",&N, &M);
6      for (i = 0; i < N; i++)
7          for (j = 0; j < M; j++)
8              scanf("%d",&A[i][j]);
9
10     for (i = 0; i < N; i++)
```

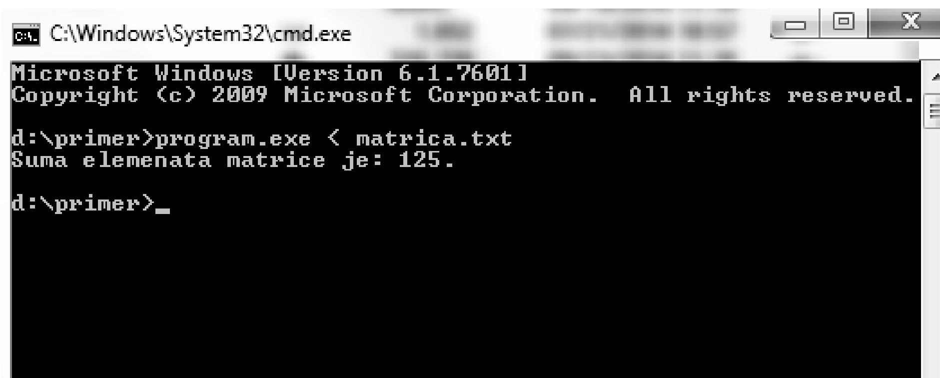
[4]Konzola se u *Windows* operativnom sistemu naziva komandni prozor (eng. *Command Prompt*), a u Linux/Unix operativnim sistemima *Terminal.*

```
11          for (j = 0; j < M; j++)
12              S += A[i][j];
13
14      printf("Suma␣elemenata␣matrice␣je:␣%d.\n",S);
15  }
```

Pretpostavimo da se ovaj program nalazi u fajlu *"program.c"* i da je nakon prevođenja dobijen izvršni fajl *"program.exe"*. Takođe pretpostavimo da je fajl *"ulaz.txt"* unapred pripremljen, tako da se nalazi u istom direktorijumu gde i izvršni fajl, i da je sadržaj fajla unet i snimljen u fajl korišćenjem ASCII tekst editora[5]. Neka je sadržaj fajla *"ulaz.txt"* sledeći:

```
5 5
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8
5 6 7 8 9
```

Navođenjem naredbe `program.exe < ulaz.txt` dobiće se izlaz prikazan na slici 6.14.



Slika 6.14: Izgled komandnog prozora prilikom preuzmeravanja toka podataka

Da bi se izlaz koji je u ovom slučaju prikazan na ekranu upisao u fajl potrebno je navesti:

```
program.exe < matrica.txt > izlaz.txt
```

△

[5]Najpoznatiji ASCII tekst editor u *Windows* operativnom sistemu je Notepad.

Napomena: Pomoću operatora za preusmeravanje standardnih tokova izlaz bilo kog programa je **bez modifikacije koda programa** moguće trajno zapamtiti na disku, ali isto tako i automatizovati ulaz prilikom unosa velikog broja podataka u program.

Operatori za preusmeravanje standardnih tokova operativnog sistema omogućavaju korisniku programa da podatke učitava iz fajlova i rezultate snima u fajl, čak i ako program inicijalno nije pisan sa namerom da radi sa podacima smeštenim u fajlovima.

6.7.2 Fajlovi i korisnički tokovi podataka

Iz ugla programskih jezika, fajlovi se mogu podeliti u dve grupe:

1. tekstualne, i
2. binarne fajlove.

Napomena: Bez obzira da li se radi o tekstualnim ili binarnim fajlovima, fajl u svakom slučaju predstavlja sekvencu bitova (bajtova) zapamćenih na fajl sistemu nekog eksternog medijuma.

Tekstualnim fajlovima nazivamo fajlove u kojima su zapamćeni ASCII kodovi simbola nekog teksta. Na disku se u okviru ovih fajlova nalazi binarna reprezentacija ASCII kodova simbola u izvornom obliku, a čitljiv prikaz ovih podataka moguće je dobiti iz bilo kog ASCII tekst editora. U binarne fajlove spadaju svi ostali fajlovi.

Svaki binarni fajl ima svoj format. Format fajla je skup pravila o reprezentaciji podataka i redosledu informacija u fajlu. Informacije iz binarnih fajlova u vizualno prihvatljivom obliku moguće je prikazati iz specijalizovanih programa koji mogu da interpretiraju informacije iz ovakvih fajlova (npr. multimedijalni fajlovi, dokumenti različitih aplikacija i sl.).

Na primer, ukoliko se realna numerička konstanta `+3.125125e-4` snima u tekstualni fajl, za ovu konstantu potrebno je 12 bajtova: jedan bajt za ASCII kod simbola '+', jedan za simbol '3', jedan za simbol '.', itd. Prednost je što je ovaj podatak moguće pročitati iz bilo kog ASCII tekst editora, a nedostatak je taj što je neophodno više prostora za pamćenje informacija. Prethodno navedenu konstantu moguće je zapamtiti u binarnom fajlu sa samo 4 bajta, npr. po IEEE 754 reprezentaciji iz C-a, ali će u tom slučaju za čitanje informacije biti neophodna specijalizovana aplikacija. Naravno, i ovaj binarni fajl moguće je otvoriti iz ASCII tekst editora, ali će on 4 bajta u binarnom obliku prikazati tako što svaki bajt prikazuje kao da je kod iz ASCII tabele, što vizualno ne odgovara zapamćenoj informaciji.

Kreiranje toka

Programski jezik C ima skup funkcija i tipova podataka u standarnim bibliotekama, koje omogućavaju rad sa fajlovima preko korisničkih tokova podataka. Bez obzira

na tip fajla, tok se kreira na isti način, a razlika je u funkcijama za upis i čitanje podataka iz tekstualnih i podataka iz binarnih fajlova.

Za svaki fajl, sa kojim se u okviru programa radi, kreira se poseban tok podataka. Jedan program može istovremeno raditi sa proizvoljnim brojem tokova podataka. Svaki tok podataka određen je jednom promenljivom strukturnog tipa, strukture `FILE`, čija se definicija nalazi u biblioteci *"stdio.h"*:

```
1  struct FILE
2  {
3      ...
4  }
```

Struktura `FILE` namenjena je radu sa fajlovima. Struktura sadrži sistemske podatke o fajlu, ali i podatak o tome dokle se stiglo sa čitanjem fajla, i sl. Svaki fajl jednoznačno određuje po jedna promenljiva tipa `FILE`, i koristi se kako bi funkcije za rad sa fajlovima imale informaciju sa kojim fajlom treba da rade.

U toku rada sa svakim fajlom prolazi se kroz 3 faze:

1. otvaranje fajla
2. čitanje / upis / obrada podataka
3. zatvaranje fajla

Otvaranje fajlova

Otvaranjem fajla nazivamo fazu u kojoj se kreira tok podataka i vezuje za željeni fajl. Rezultat ove faze je promenljiva tipa `FILE`, koja jednoznačno određuje kreirani tok podataka i koja se u kasnijim fazama koristi za rad sa fajlom.

Otvaranje fajla, t.j. kreiranje toka u C-u vrši se pozivom jedne funkcije iz biblioteke *"stdio.h"*. Naziv ove funkcije je *fopen*.

1. **Deklaracija funkcije**

   ```
   FILE* fopen(const char* imefajla, const char* mod)
   ```

2. **Opis dejstva**

   Funkcija otvara fajl, čije je ime navedeno kao parametar, za čitanje ili upis podataka. U toku kreiranja toka, funkcija kreira jednu strukturnu promenljivu tipa `FILE` i inicijalizuje vrednosti parametara koji čine strukturu podacima o konkretnom fajlu. Funkcija vraća pokazivač na ovu strukturu, kako bi služio kao referenca na kreirani tok podataka pri daljem korišćenu toka.

3. **Parametri**

   imefajla – ime fajla je parametar preko koga se zadaje ili samo ime fajla, ili ime fajla sa kompletnom putanjom do fajla. Prilikom poizva funkcije,

na mestu ovog parametra može stajati promenljiva tipa string sa prethodno zadatom vrednošću, ili literal. Na primer, vrednost prvog parametra može se zadati literalom na sledeći način:

```
... fopen("matrica.txt", ...
```

Da bi u ovom slučaju fajl bio uspešno otvoren, kako nije navedena putanja do fajla, fajl koji se otvara mora biti u istom direktorijumu u kom se nalazi i izvršni fajl programa.
Što se tiče navođenja imena fajla sa putanjom do fajla, postoje dva načina: navođenje **apsolutne**, ili **relativne** putanje. Apsolutna putanja je putanja navedena počev od korena fajl sistema, pa do samog fajla. Primer apsolutne putanje je sledeći literal:

```
"C:\\Users\\vciric\\Documents\\matrica.txt"
```

Napomena: Imajući u vidu da je simbol '\' specijalni karakter u okviru literala u C-u, i označava da slovo iza njega određuje neku naredbu koju je potrebno izvršiti prilikom prikaza (prelazak u novi red i sl.), da bi se u putanji naveo sam simbol '\', navode se dva simbola '\\'.

Relativna putanja je putanja relativna u odnosu na izvršni fajl programa, i navodi se na sledeći način:

```
"baza\\podaci\\matrica.txt"
```

Prethodni primer podrazumeva da se u dikrektorijumu u kom se nalazi izvršni fajl programa, bez obzira gde se on nalazio na disku, nalazi poddirektorijum *"baza"*, a u okviru njega poddirektorijum *"podaci"*, koji sadrži traženi fajl. Kaže se da je ova putanja *relativna* u odnosu na lokaciju izvršnog fajla programa na disku.
Način zadavanja vrednosti parametra prenošenjem promenljive tipa string, čiju vrednost prethodno unosi korisnik, ilustrovana je sledećim primerom:

```
char ime[80];
printf("Unesite ime fajla: ");
gets(ime);
... fopen(ime, ...
```

Korisnik u ovom slučaju može zadati ime fajla sa tasture sa, ili bez putanje. Prilikom unosa sa tastature nema potrebe da korisnik za delimitere direktorijuma unosi dvostruki simbol '\', jer se ovaj simbol svakako prilikom unosa sa tastature konvertuje u odgovarajući kod.

mod – mod, ili način na koji se otvara fajl, je jednoslovni, dvoslovni, ili u nekim slučajevima troslovni literal, koji može imati jednu od sledećih vrednosti:

- "r" – (od eng. ***read***, čitanje), fajl se otvara isključivo za čitanje. Više programa istovremeno mogu otvoriti jedan fajl za čitanje. Fajl mora da postoji na disku, inače će funkcija vratiti grešku.
- "w" – (od eng. ***write***, pisanje), fajl se otvara isključivo za upis podataka. Samo jedan program u jednom trenutku može imati jedan fajl otvoren za upis. Da bi neki drugi program otvorio taj fajl za upis, prethodni program prvo mora zatvoriti fajl.
  Ukoliko navedeni fajl ne postoji, a navedena je ova opcija za parametar *mod*, biće kreiran prazan fajl.
  Ukoliko navedeni fajl postoji, funkcija će obrisati ceo prethodni sadržaj. U oba slučaja se podaci koje upisuje program upisuju od početka fajla.
- "a" – (od eng. ***append***, dodavanje), fajl se otvara isključivo za upis, a ukoliko fajl postoji na disku neće biti obrisan, već će se podaci prilikom upisa dodavati na kraj fajla. Ukoliko navedeni fajl ne postoji, biće kreiran. Kao i kod prethodne opcije, samo jedan program može u jednom trenutku imati otvoren konkretni fajl za upis.
- "r+" – fajl se otvara i za čitanje i upis. Pravila koja važe za prethodno postojanje fajla ista su kao i za opciju "r": fajl mora postojati na disku.
- "w+" – fajl se otvara i za čitanje i upis. Pravila koja važe za prethodno postojanje fajla ista su kao i za opciju "w": fajl ne mora postojati na disku. Ako ne postoji biće kreiran, a ako postoji biće mu obrisan sadržaj.
- "a+" – fajl se otvara i za čitanje i upis. Pravila koja važe za prethodno postojanje fajla ista su kao i za opciju "a": fajl ne mora postojati na disku. Ako ne postoji, biće kreiran, a ako postoji, prilikom upisa će se podaci dodavati na kraj fajla.

4. **Povratna vrednost**

   Funkcija vraća pokazivač na strukturu tipa `FILE`, koju je kreirala u postupku otvaranja toka.

   Ukoliko iz nekog razloga fajl nije uspešno otvoren, funkcija će vratiti 0. Često se za ovu vrednost koristi i konstanta `NULL`. Na primer, fajl koji se otvara za čitanje ne postoji na disku, ili fajl koji se otvara za upis je već otvoren za upis od strane nekog drugog programa. Funkcija će vratiti `NULL` i ako sa fajl otvara za upis, a na disku više nema slobodnog praznog prostora. Dobra praksa je odmah nakon otvaranja proveriti da li je fajl uspešno otvoren, i samo ako jeste pozivati funkcije za čitanje i upis. U protivnom, ove funkcije će prouzrokovati sistemsku grešku i operativni sistem će prekinuti program.

Primer 6.37 (Otvaranje fajla)**.** Sledeći primer ilustruje otvaranje fajla sa unapred zadatim imenom, koji se nalazi u istom direktorijumu u kom je i izvršni fajl programa. Navedeni program će ovaj fajl otvoriti isključivo za čitanje.

```
1  #include "stdio.h"
2  main()
3  {
4      FILE* f;
5      f = fopen("primer.jpg","r");
6      if ( f != NULL )
7      {
8          // citanje / obrada
9      }
10     else
11         printf("Fajl nije pronadjen u dorektorijumu.");
12 }
```

Nakon otvaranja fajla, u 6. liniji programa izvršena je provera da li je fajl uspešno otvoren.

Veoma često se u C programima sreće sledeća konstrukcija provere uspešnosti otvaranja fajla:

```
1  #include "stdio.h"
2  main()
3  {
4      FILE* f;
5      if ( !(f = fopen("primer.jpg","r")) )
6      {
7          printf("Fajl nije pronadjen u dorektorijumu.");
8          return;
9      }
10
11     // citanje / obrada
12 }
```

Prethodna dva programa su ekvivalentna, ali je u drugom slučaju iskorišćena mogućnost navođenja izraza sa operatorom dodele u uslovu alternacije. Negacija ovog uslova je dodata kako bi se izbeglo pisanje operativnog dela programa u okviru grane alternacije.

Treba skrenuti pažnju da je ovo rešenje nestrukturno, jer glavni program *main*() ima dva moguća izlaza: izvršenjem *return*, u slučaju da fajl nije uspešno otvoren, i regularan izlaz posle poslednje naredbe programa. Naredba *return* će, u slučaju da fajl nije uspešno otvoren, prekinuti dalje izvršenje programa, naravno, uz prethodno ispisanu poruku o grešci iz prethodne linije programa. △

Zatvaranje fajlova

S obzirom na to da su tokovi podataka koji se koriste za rad sa fajlovima baferovani, podaci koji se upisuju u fajl privremeno se smeštaju u memoriju bafera. Podaci iz bafera se upisuju u disk po nekoj unapred definisnoj šemi, a najčešće se upis vrši ili kada se napuni bafer, ili nakon određenog vremena od poslednjeg upisa u bafer.

Od trenutka kada program ima informaciju da su podaci upisani, dok se oni i stvarno ne nađu na disku, može proći neko vreme. Memorija bafera nije postojana i gubi sadržaj nakon nestanka napajanja. Ukoliko dođe do nestanka napajanja u vremenskom intervalu od trenutka kada program upiše podatke, do kada algoritmi za upravljanje baferom podatke ne prebace na disk, podaci će biti izgubljeni.

Zatvaranje fajla je faza u kojoj program eksplicitno navodi da je završio sa obradom fajla, da želi da oslobodi sve memorijske strukture koje su za tu priliku zauzete, i da želi da se baferi isprazne i podaci pošalju na disk.

Ova faza nije neophodna, jer će se podaci svakako posle nekog vemena naći na disku, ali je poželjno fajl uvek zatvoriti, naročito ako je otvoren za pisanje, jer dok se ne zatvori, drugi programi neće moći pristupiti podacima u ovom fajlu.

Funkcija za zatvaranje fajla je *fclose* i nalazi se u biblioteci *stdio.h*.

1. **Deklaracija funkcije**

   ```
   int fclose(FILE* stream)
   ```

2. **Opis dejstva**

   Funkcija zatvara prethodno otvoreni fajl i oslobađa sve strukture i resurse koje su prethodno zauzeti kreiranjem toka.

3. **Parametri**

   stream – Funkcija ima jedan parametar. Ovaj parametar ukazuje na fajl, odnosno tok koji treba zatvoriti. Po tipu, ovaj parametar je pokazivač na strukturu `FILE`.

4. **Povratna vrednost**

   Funkcija vraća numeričku vrednost 0, ukoliko je fajl uspešno zatvoren. Ukoliko zatvaranje nije uspešno, funkcija vraća celobrojnu numeričku vrednost `EOF`, definisanu u biblioteci *"stdio.h"* u vidu simboličke konstante.

Primer 6.38 (Zatvaranje fajla)**.** Sledeći primer ilustruje zatvaranje fajla. Fajl *"matrica.txt"* je na početku programa otvoren za upis, pa zatvoren, a odmah nakon toga otvoren za čitanje, nakon čega je ponovo zatvoren. Ovaj primer nije opterećen ispitivanjem uslova da li je fajl uspešno otvoren.

```
1 #include "stdio.h"
2 main()
3 {
4    FILE* f,g;
```

```
5     f = fopen("matrica.txt","w");
6     // upis
7     fclose(f);
8     g = fopen("matrica.txt","r");
9     // citanje
10    fclose(g);
11 }
```

Potrebno je napomenuti da je u ovom primeru bilo moguće koristiti samo jedan pokazivač na strukturu tipa `FILE`. △

6.7.3 Tekstualni fajlovi

Za upis i čitanje podataka iz tekstualnih fajlova, u slučaju formatiranog ulaza i izlaza, moguće je koristiti funkcije *fscanf* i *fprintf*. Za neformatirani ulaz i izlaz na raspolaganju su funkcije *fgets* i *fputs*, kao i funkcije *getc* i *putc* za upis i čitanje karaktera. Ove funkcije nalaze se u standardnoj bibilioteci *"stdio.h"*.

Upisivanje podataka u tekstualni fajl

Funkcije za upisivanje podataka u tekstualni fajl, ili preciznije tekstualni tok podataka, koje ćemo ovde obraditi su: *fprintf*, *fputs* i *putc*.

fprintf

1. **Deklaracija funkcije**

   ```
   int fprintf(FILE* tok, const char* format, ...)
   ```

2. **Opis dejstva**

   Funkcija upisuje podatke u prethodno otvoreni tekstualni fajl po zadatom formatu na isti način kao što funkcija *printf* ispisuje podatke na ekranu.

3. **Parametri**

   tok – Referenca na tok podataka, odnosno pokazivač na strukturu `FILE` koji je vratila funkcija *fopen* prilikom otvaranja.

   format – Identično kao kod funkcije *printf*, ovaj parametar predstavlja literal koji se upisuje u fajl. Literal može sadržati proizvoljan broj pojedinačnih izlaznih konverzija (definicija 3.17, strana 112).

   ... – Lista promenljivih, konstanti, ili izraza međusobno odvojenih zarezima, čije se vrednosti prikazuju u zadatom formatu. Za svaku vrednost iz liste potrebno je da u formatu postoji odgovarajući konverzioni karakter.

4. **Povratna vrednost**

   Ukoliko je upis uspešno završen, funkcija vraća broj karaktera upisanih u fajl. Ukoliko je upis neuspešan, funkcija vraća negativnu vrednost.

Primer 6.39 (Upis podataka u tekstualni fajl funkcijom *fprintf*)**.** Sledeći program ilustruje upotrebu funkcije *fprintf*.

```
1  #include "stdio.h"
2  main()
3  {
4      int i, A[] = {1,2,3,4,5,6,7,8,9,10};
5      FILE* f;
6      f = fopen("niz.txt","w");
7      fprintf(f, "Niz␣je:\n");
8      for (i = 0; i < 10; i++)
9          fprintf(f, "%d,",A[i]);
10     fclose(f);
11 }
```

Nakon izvršenja ovog programa, u direktorijumu u kom se nalazi izvršni fajl programa biće kreiran fajl *"niz.txt"* i u njega će biti upisan sledeći tekst:

```
Niz je:
1,2,3,4,5,6,7,8,9,10,
```

△

fputs

1. **Deklaracija funkcije**

   ```
   int fputs(const char* str, FILE* tok)
   ```

2. **Opis dejstva**

   Ova funkcija se koristi za neformatirani upis teksta u fajl. Funkcija upisuje sadržaj zadatog stringa u prethodno otvoreni tekstualni fajl.

3. **Parametri**

   str – String čiji se sadržaj upisuje u fajl.

   tok – Pokazivač na strukturu `FILE` prethodno otvorenog fajla.

4. **Povratna vrednost**

   Funkcija vraća pozitivnu vrednost različitu od nule ukoliko je upis uspešno završen, a ukoliko nije vraća numeričku vrednost definisanu simboličkom konstantom `EOF`.

Primer 6.40 (Upis podataka u tekstualni fajl funkcijom *fputs*)**.** Sledeći program ilustruje upotrebu funkcije *fputs*.

```
1 #include "stdio.h"
2 main()
3 {
4     char A[] = "Ova recenica ce biti upisana u tekstualni fajl.";
5     FILE* f;
6     f = fopen("niz.txt","w");
7     fputs(A,f);
8     fclose(f);
9 }
```

△

fputc

1. **Deklaracija funkcije**

   ```
   int putc(int char, FILE *tok)
   ```

2. **Opis dejstva**

   Funkcija upisuje jedan karakter u prethodno otvoreni tekstualni fajl.

3. **Parametri**

   char – Karakter čija se vrednost upisuje u fajl.

   tok – Pokazivač na strukturu `FILE` prethodno otvorenog fajla.

4. **Povratna vrednost**

   Funkcija vraća vrednost upisanog karaktera konvertovanu iz tipa *char* u tip *int*, a ukoliko upis nije uspešan vraća numeričku vrednost definisanu simboličkom konstantom `EOF`.

Učitavanje podataka iz tekstualnog fajla

Funkcije za čitanje podataka iz tekstualnog toka podataka su: *fscanf*, *fgets* i *getc*.

fscanf

Namena ove funkcije je učitavanje podataka čiji je format zapisa unapred poznat.

1. **Deklaracija funkcije**

   ```
   int fscanf(FILE *tok, const char *format, ...)
   ```

2. **Opis dejstva**

   Funkcija učitava podatke iz prethodno otvorenog tekstualnog fajla po zadatom formatu, na isti način kao što funkcija *scanf* učitava podatke sa tastature.

3. **Parametri**

   tok – Referenca na tok podataka, odnosno pokazivač na strukturu `FILE` koji je vratila funkcija *fopen* prilikom otvaranja.

   format – Identično kao kod funkcije *scanf*, ovaj parametar predstavlja literal koji sadrži pojedinačne ulazne konverzije za čitanje podataka. Literal može sadržati proizvoljan broj pojedinačnih ulaznih konverzija.

   ... – Lista adresa promenljivih, međusobno odvojenih zarezima, gde se smeštaju učitani podaci. Za svaku vrednost iz liste potrebno je da u formatu postoji odgovarajući konverzioni karakter.

4. **Povratna vrednost**

   Ukoliko je čitanje uspešno završeno, funkcija vraća broj uspešno učitanih parametara. Ovaj broj može biti jednak ili manji od zadatog broja parametara, pa i jednak 0, ukoliko nijedan podatak nije učitan.

Napomena: Važna napomena za sve funkcije koje čitaju podatke iz tekstualnih tokova je da ove funkcije <u>automatski</u> modifikuju parametre strukture `FILE` zadatog toka, tako da promenljiva, koja u okviru strukture pamti dokle se stiglo sa čitanjem, uvek ukazuje na prvi naredni podatak koji je potrebno pročitati.

Imajući u vidu prethodnu napomenu, potrebno je naglasiti da je ažuriranje strukture nakon čitanja, tako da ukazuje na naredni podatak, razlog zbog čega funkcija *fscanf* nema parametar koji precizira iz kog dela fajla se čita podatak.

Primer 6.41 (Čitanje podataka iz tekstualnog fajla funkcijom *fscanf*)**.** Sledeći program ilustruje upotrebu funkcije *fscanf*.

```
1  #include "stdio.h"
2  main()
3  {
4      int x,y,z;
5      FILE* f;
6      f = fopen("niz.txt","r");
7      fscanf(f, "%d,",&x);
8      fscanf(f, "%d,",&y);
9      fscanf(f, "%d,",&z);
10     fclose(f);
11 }
```

Pretpostavimo da fajl *"niz.txt"* postoji na disku i da je u njega upisan sledeći tekst:

```
15
7
9
111
```

Nakon otvaranja fajla za čitanje u 6. liniji programa, funkcija *fscanf* će sa novootvorenog toka pročitati jednu celobrojnu vrednost (15) i upisati je u promenljivu x. Ono što se eksplicitno ne vidi je da će funkcija *fscanf* pomeriti interni pokazivač, koji pamti dokle se stiglo sa čitanjem fajla, na naredni podatak. Tako će sledeći poziv ove funkcije pročitati sledeći podatak, a treći poziv će pročitati treći podatak. Kako fajl sadrži 4 cela broja, a program ima 3 poziva funkcije, poslednji broj (111) će ostati nepročitan. △

Kao što se iz prethodnog primera vidi, da bi se na ovaj način pročitao N-ti podatak potrebno je prethodno pročitati prethodnih $N-1$ podataka. Kaže se da je ovakav pristup fajlu sekvencijalni pristup.

Definicija 6.8 (Sekvencijalni pristup)**.** Sekvencijalni pristup je način čitanja podataka kod koga čitanje N-tog podatka zahteva prethodno čitanje prethodnih $N-1$ podataka.

Postoje medijumi kod kojih je sekvencijalni pristup osnovni način pristupa medijumu. Magnetna traka je medijum sa sekvencijalnim pristupom. Disk je medijum koji omogućava i sekvencijalni i nesekvencijalni pristup podacima. Nesekvencijalni pristup, koji je u literaturi poznat i kao **random** pristup (eng. *random* - slučajni), obradićemo kod binarnih fajlova.

fgets

Namena funkcije fgets je učitavanje podataka iz tekstualnog fajla bez unapred zadatog formata podataka.

1. **Deklaracija funkcije**

   ```
   char *fgets(char *str, int n, FILE *tok)
   ```

2. **Opis dejstva**

   Namena ove funkcije je neformatirani unos podataka. Funkcija učitava zadati broj karaktera iz prethodno otvorenog fajla i upisuje ih u string. Funkcija na kraj učitanog stringa dodaje simbol za kraj '\0'.

3. **Parametri**

   srt – String u koji će učitani tekst biti zapamćen. Ovaj parametar je po tipu pokazivač na niz karaktera, tako da se prilikom poziva funkcije na mestu ovog parametra prenosi samo ime stringa.

n – Maksimalni broj karaktera koji će biti učitan. S obzirom na to da je string u koji se učitava tekst ograničene veličine, koja može biti značajno manja od veličine fajla, broj karaktera koji funkcija učitava se ograničava na ovaj način.

Ukoliko u jednom redu teksta ima više karaktera nego što parametar n dozvoljava, jednim pozivom će biti učitano ukupno $n-1$ karaktera, i kao poslednji karakter će biti dodat simbol za kraj stringa '\0'. Ukoliko do kraja linije ima manje od n karatera, tada će biti učitani samo karakteri do kraja reda, ali ne i oni iz narednog reda.

Kao oznaka za kraj reda u tekstualnim fajlovima koristi se specijalni karakter čiji je ASCII kod jednak vrednosti simboličke konstante iz standardne biblioteke *"stdio.h"* pod nazivom EOL. Ime ove simboličke konstante je skraćenica od engleskih reči *End Of Line*, što u prevodu znači kraj reda/linije.

tok – Tok sa koga se učitavaju podaci.

4. **Povratna vrednost**

   U slučaju uspešnog čitanja podataka funkcija vraća pokazivač na string u koji je upisala podatke. U suprotom, ukoliko nijedan podatak nije učitan zato što se prethodnim čitanjima došlo do kraja fajla, funkcija će vratiti NULL.

Primer 6.42 (Učitavanje neformatiranih podataka iz tekstualnog fajla)**.** Pretpostavimo da je sadržaj tekstualnog fajla sledeći:

```
Prvi red je u ovom slucaju duzi od ostalih.
Drugi red.
Treci red.
```

U prvom redu ima 43, a u druga dva po 10 karaktera. Ukoliko je za veličinu stringa uzeto 30 karaktera, čitanje je moguće izvršiti sukcesivnim pozivanjem funkcija *fgets* sa maksimalno onoliko karaktera u svakom pozivu kolika je dužina stringa, kao što je urađeno u sledećem programu.

```
1  #include "stdio.h"
2  main()
3  {
4      char A[30];
5      FILE *f = fopen("niz.txt", "r");
6
7      fgets(A,30,f);
8      fgets(A,30,f);
9      fgets(A,30,f);
10     fgets(A,30,f);
11
12     fclose(f);
13 }
```

U prvom pozivu će biti učitano prvih 29 karaktera, t.j. tekst: *"Prvi red je u ovom slucaju du"*, a kao poslednji karakter u stringu će biti pridodat simbol za kraj stringa '\0'. Ovaj poziv će ujedno i povećati interni brojač u strukturi `FILE`, kako bi se označio deo fajla koji je pročitan. Sledeći poziv će pročitati tekst do kraja reda, ali ne i naredni red, pa će vrednost u stringu *A* posle drugog poziva biti *"zi od ostalih."*.

Treći poziv će učitati samo drugi red, a četvrti poziv treći red. △

getc

Nekada je zbog prirode konkretnog algoritma za obradu tekstualnog fajla optimalnije i jednostavnije čitati jedan po jedan karakter iz fajla i odmah ih obrađivati, umesto učitavanja cele linije u string, odakle se nakon toga vrši obrada. Funkcija koja omogućava čitanje jednog po jednog karaktera iz tekstualnog fajla je funkcija *getc* iz standardne biblioteke *"stdio.h"*.

1. **Deklaracija funkcije**

   ```
   int getc(FILE *tok)
   ```

2. **Opis dejstva**

   Čita jedan karakter iz fajla, odnosno ulaznog toka podataka.

3. **Parametri**

   tok – Tok sa koga se učitavaju podaci.

4. **Povratna vrednost**

   Funkcija čita podatak tipa *unsigned char*, konvertuje ga i vraća podatak tipa *int*. U slučaju da se prilikom čitanja fajla došlo do kraja fajla, kao i u slučaju greške, funkcija vraća specijalni karakter čiji je ASCII kod jednak vrednosti simboličke konstante iz standardne biblioteke *"stdio.h"* pod nazivom **EOF**. Ime ove simboličke konstante je skraćenica od engleskih reči *End Of File*, što u prevodu znači kraj fajla.

Sledeći primer ilustruje upotrebu funkcije *getc*.

Primer 6.43 (Upotreba funkcije *getc* za učitavanje jednog karaktera iz fajla)**.** Ukoliko je na primer potrebno odrediti broj karaktera u fajlu, to je jednostavnije učiniti čitanjem jednog po jednog karaktera iz fajla. Sledeći program određuje broj karaktera u fajlu *"niz.txt"*.

```
1  #include "stdio.h"
2  main()
3  {
4      int B=0;
5      FILE *f = fopen("niz.txt", "r");
```

```
6
7      while ( getc(f) != EOF )
8          B++;
9
10     fclose(f);
11 }
```

Problem je rešen jednom *while* petljom koja učitava jedan po jedan karakter sve dok funkcija *getc* ne vrati vrednost jednaku konstanti EOF, što je znak da su pročitani svi karakteri i da je pri čitanju dosegnut kraj fajla. △

Napomena: Funkcija *getc* se može koristiti za učitavanja karaktera sa tastature karakter po karakter, ukoliko se za parametar funkcije navede standardni ulaz *stdin* kao: `char c = getc(stdin);`.

Ispitivanje uslova za kraj fajla

Prilikom učitavanja podataka iz fajlova retko kada se unapred zna koliko je podataka zapamćeno u fajlu[6]. Zbog toga se učitavanje obično vrši sukcesivnim pozivanjem funkcija za čitanje podataka iz fajla, sve dok se u čitanju ne dođe do kraja fajla.

Kraj fajla moguće je prepoznati iz povratne vrednosti funkcije za čitanje, kako je pokazano u primeru 6.43.

Pored konstante EOF (primer 6.43), u biblioteci *stdio.h* dostupna je i funkcija *feof*, koja daje informaciju da li se prilikom čitanja došlo do kraja fajla. Naziv ove funkcije je skraćenica od engleskih reči ***F**ile **E**nd **O**f **F**ile*.

1. **Deklaracija funkcije**

   ```
   int feof(FILE *tok)
   ```

2. **Opis dejstva**

   Funkcija testira indikator kraja fajla za zadati tok podataka i vraća odgovarajuću informaciju da li je dosegnut kraj fajla prilikom sekvencijalnog čitanja podataka.

3. **Parametri**

   tok – Tok sa koga se učitavaju podaci.

[6]Postoje funkcije koje mogu da odrede i vrate veličinu fajla na disku, pa je iz ovoga moguće preračunati broj karaktera zapamćenih u fajlu, ali je na ovaj način nemoguće odrediti, na primer, broj reči, ukoliko se radi o formatiranom ulazu. Zbog ovoga je kraj fajla prilikom učitavanja podataka potrebno odrediti na drugi način.

4. **Povratna vrednost**

   Funkcija vraća numeričku vrednost tipa *int* različitu od nule, ukoliko nije dosegnut indikator kraja fajla EOF. Ukoliko se indikator pozicije čitanja iz fajla nalazi na kraju fajla, funkcija će vratiti nulu.

   S obzirom na to da funkcija vraća vrednost 0 ukoliko ima još podataka koje je potrebno pročitati, a kako ova vrednost predstavlja logičku konstantu *false* u C-u, uslov je obično potrebno negirati. Sledeći program ilustruje ovo. Program iz fajla sa celobrojnim podacima učitava brojeve redom, dok se prilikom čitanja ne dođe do kraja fajla.

```
1  #include "stdio.h"
2  {
3      FILE *f = fopen("primer.txt", "r");
4      int x;
5      while (!feof(f))
6          fscanf(f, "%d", &x);
7  }
```

Primer 6.44 (Primer čitanja formatiranih podataka iz fajla). **Zadatak**: Napisati strukturni program na programskom jeziku C koji određuje i prikazuje ime i prezime studenta sa najvećim brojem poena na ispitu. Podaci o studentima su zapamćeni u fajlu na disku pod nazivom *"studenti.txt"*, tako da se na početku svake linije nalazi broj indeksa, pa prezime i ime studenta, a nakon toga i broj poena u opsegu od 0 do 100.

Rešenje: Imajući u vidu da se format po kom su zapamćeni podaci zna, najjednostavnije je za učitavanje podataka upotrebiti funkciju *fscanf*. Format je takav da je na početku svake linije broj indeksa (ceo pozitivan broj, "%d" u formatu funkcije), pa prezime (string, "%s" u formatu funkcije) i ime (takođe string, "%s"), i na kraju broj poena (ceo broj, "%d"). Format će dakle biti "%d%s%s%d".

Učitavanje se vrši sve dok ima podataka, a informaciju o tome može dati funkcija *feof*. Rešenje zadatka dato je u nastavku.

```
1  #include "stdio.h"
2  #include "string.h"
3  void main()
4  {
5      char ime[25], prez[25], mime[25], mprez[25];        // ime i prezime
6      unsigned short ind;                                 // indeks
7      unsigned char brp, mbrp=0;          // br. poena i max. br. poena
8
9      // otvaranje fajla
10     FILE *stud = fopen("studenti.txt","r");
11     // sve dok se ne dodje do kraja...
12     while (!feof(stud))
13     {
14         fscanf(stud, "%d%s%s%d",&ind,prez,ime,&brp);
```

```
15        if (brp > mbrp)        // ako je trenutni bolji ...
16        {
17            // uzmi njegove podatke
18            mbrp = brp;
19            strcpy(mime, ime);
20            strcpy(mprez, prez);
21        }
22    }
23    // prikaz najboljeg
24    printf("Najbolji␣na␣ispitu␣je:␣%s␣%s.", mime, mprez);
25 }
```

Nakon svakog učitavanja vrši se provera da li je učitani broj poena najveći učitani broj do tada. Ako jeste, uzimaju se ime i prezime studenta u posebne stringove. Kako je potrebno kopirati stringove da bi se postiglo ovo, iskorišćena je funkcija *strcpy*.

Neka je sadržaj fajla *"studenti.txt"* sledeći:

```
13500 Lazic Lazar 96
11600 Jankovic Janko 78
12108 Peric Petar 100
...
12333 Mihajlovic Mihajlo 50
```

Funkcija će prikazati:

```
Najbolji na ispitu je: Petar Peric.
```

U ovom rešenju nije razmatrana mogućnost, niti je uzeta u obzir činjenica da je moguće da više studenata ima maksimalan broj poena. Da jeste, bilo bi potrebno jednom proći kroz sve podatke i odrediti maksimum, a onda zatvoriti i ponovo otvoriti fajl, kako bi se počelo sa čitanjem od početka i u jednom prolazu prikazati sve studente koji imaju broj poena jednak maksimalnom broju. △

6.7.4 Binarni fajlovi

Za otvaranje binarnih fajlova takođe se koristi funkcija *fopen*, s tom razlikom da se uz oznaku moda dodaje i slovo **'b'**. Tako će mod za otvaranje binarnog fajla za čitanje biti "rb", za upis "wb", za dodavanje "ab". Za upis i čitanje slovo 'b' je moguće dodati ili nakon slova "r" ili nakon simbola "+" kao "rb+", ili "r+b". Isto važi i za "w+" i "a+".

Sledeći program otvara binarni fajl ekskluzivno za čitanje i odmah nakon toga ga zatvara.

```
1 #include "stdio.h"
2 main()
3 {
```

```
4     FILE* bin;
5     bin = fopen("slika.bmp","rb");
6     fclose(bin);
7 }
```

Čitanje iz binarnog fajla

Funkcija za čitanje iz binarnog fajla je *fread* i nalazi se u standardnoj biblioteci *"stdio.h"*. Deklaracija funkcije je sledeća:

```
size_t fread(void *odrediste, size_t velicina,
    size_t broj, FILE *tok)
```

Parametri funkcije su:

odrediste – Pokazivač na početak niza podataka u memoriji gde će učitani podaci biti smešteni. Po tipu je ovaj parametar "neodređen" (*void*), tako da je moguće preneti niz bilo kog tipa: *int*, *char*, ..., ili niz bilo kog strukturnog tipa koji definiše programer.

velicina – Veličina jednog podatka u bajtovima.

broj – Ukupan broj podataka koji se jednim pozivom ove funkcije učitavaju.

tok – Referenca na tok podataka iz koga se učitavaju podaci.

Broj bajtova koji će biti pročitani je $velicina \cdot broj$.

Funkcija vraća broj uspešno učitanih elemenata iz fajla. Ukoliko je ovaj broj različit od broja koji se želi pročitati, navedenog kao parametar *broj*, ili je došlo do greške, ili se prilikom čitanja došlo do kraja fajla, pa je učitani broj manji pd željenog broja.

Primer 6.45 (Primer učitavanja podataka iz binarnog fajla)**.** Sledeći program ilustruje upotrebu ove funkcije na primeru čitanja prvih 2 kilobajta podataka iz fajla *"slika.txt"*.

```
1  #include "stdio.h"
2  void main()
3  {
4      // otvaranje fajla
5      FILE* bin;
6      bin = fopen("slika.gif","rb");
7          // citanje podataka
8      unsigned char bafer[2048];  // bafer za podatke, ukupne vel. 2KB
9      fread(bafer, sizeof(unsigned char), 2048, bin);
10     // zatvaranje
11     fclose(bin);
12 }
```

Za smeštanje učitanih podataka u memoriju rezervisan je niz od 2048 celih, pozitivnih brojeva veličine 1B (*unisigned char*), t.j. ukupno 2KB podataka. Svih 2KB podataka se u ovom slučaju učitava iz fajla jednom naredbom. Potrebno je naglasiti da bi naredni poziv ove funkcije učitao sledećih 2KB podataka, i tako redom.

U ovom primeru za fajl čiji se sadržaj učitava uzeta je slika u *.gif* formatu, pa tako neki od učitanih bajtova nose inforamciju o veličini slike, a neki daju informacije o samim pikselima slike. Značenje pojedinih bajtova u ovom konkretnom formatu, kao i u bilo kom drugom formatu binarnog fajla, ukoliko je format standardizovan i javno dostupan, moguće je naći u dokumentu standarda koji opisuje konkretni format. △

Upis u binarni fajl

U binarni fajl se podaci mogu upisivati funkcijom *fwrite* iz biblioteke *"stdio.h"*. Deklaracija funkcije je sledeća:

```
size_t fwrite(const void *odrediste, size_t velicina,
    size_t broj, FILE *tok)
```

Parametri funkcije su:

odrediste – Pokazivač na početak niza podataka u memoriji gde se nalaze podaci koje je potrebno upisati u fajl. Po tipu je ovaj parametar "neodređen" (*void*), tako da je moguće preneti niz bilo kog tipa: *int*, *char*, ..., ili niz bilo kog strukturnog tipa koji definiše programer.

velicina – Veličina jednog podatka u bajtovima.

broj – Ukupan broj podataka koji se jednim pozivom ove funkcije upisuju u fajl.

tok – Referenca na tok podataka na koji se šalju podaci.

Broj bajtova koji će biti pročitani je $velicina \cdot broj$.

Funkcija vraća broj uspešno upisanih elemenata. Ukoliko je ovaj broj različit od broja koji se želi upisati (parametar *broj*), to znači da je došlo do greške pri upisu i da nisu svi podaci upisani.

Primer 6.46 (Primer upisa u binarni fajl)**.** Sledeći program ilustruje upotrebu ove funkcije na primeru upisa elemenata niza.

```
1  #include "stdio.h"
2  void main()
3  {
4     FILE* niz;
5     niz = fopen("niz.bin","wb");
6
7     int podaci[20] =
8        {4,3,5,4,1,6,7,8,10,13,2,23,14,0,1,0,1,2,3,1};
9     fwrite(podaci, sizeof(int), 20, niz);
10
```

```
11     fclose(niz);
12 }
```

△

Primer 6.47 (Primer upisa strukturnih podataka u binarni fajl)**.** Funkcije *fread* i *fwrite* za parametre mogu imati i korisničke strukture podataka. Sledeći program upisuje 20 strukturnih podataka u binarni fajl, gde svaka struktura sadrži ime i prezime osobe, broj telefona, i godinu rođenja.

```
1  #include "stdio.h"
2  #include "string.h"  // zbog strcpy
3  struct osoba
4  {
5      char ime[25];
6      char prezime[25];
7      char telefon[15];
8      unsigned short godina;
9  };
10 void main()
11 {
12     // otvaranje fajla
13     FILE* fajl;
14     fajl = fopen("osobe.bin","wb");
15
16     struct osoba podaci[100];  // niz struktura
17
18     strcpy(podaci[0].ime,"Perar");
19     strcpy(podaci[0].prezime,"Peric");
20     strcpy(podaci[0].telefon,"+38118529100");
21     podaci[0].godina = 1980;
22
23     //... obrada ostalih podataka
24
25     // upis - sve odjednom!
26     fwrite(podaci, sizeof(osoba), 20, fajl);
27
28     fclose(fajl);
29 }
```

△

Pozicioniranje u fajlu

Nakon čitanja podataka funkcija *fread* će se pozicionirati na

$$velicina \cdot broj + 1.$$

bajt u fajlu, gde su $velicina$ i $broj$ parametri funkcije. Sledeći poziv funkcije učitava narednih $velicina \cdot broj$ bajtova, itd.

U standardnoj biblioteci *"stdio.h"* postoji nekoliko funkcija za eksplicitno pozicioniranje na bajt u fajlu počev od koga se želi čitanje. Ove funkcije, za razliku od sekvencijalnnog pristupa, omogućavaju i nesekvencijalni pristup podacima. Ove funkcije su: *fseek* za pozicioniranje na željeni bajt u fajlu, *frewind* za pozicioniranje na početak fajla, i *freopen* koja zatvara i ponovo otvara fajl, čime se pozicija za čitanje resetuje i postavlja na početak fajla.

Funkcija *fseek* omogućava pozicioniranje na željeni bajt u fajlu. Deklaracija funkcije *fseek* je:

```
int fseek(FILE *tok, long int ofset, int referenca)
```

Parametri funkcije su sledeći:

tok – Tok sa koga se učitavaju podaci.

$ofset$ – Celobrojni podatak koji predstavlja ofset, odnosno pomeraj ili udaljenost podatka na koji se prelazi u odnosu na referentnu poziciju u fajlu. Pomeraj se izražava u bajtovima. Pomeraj može biti pozitivan i negativan broj.

$referenca$ – Ovaj parametar je celobrojni podatak koji može označiti da je referentna tačka u odnosu na koju se računa pomeraj početak, kraj fajla, ili trenutna pozicija na kojoj se brojač nalazi.

U standardnoj biblioteci *"stdio.h"* definisane su sledeće simboličke konstante za ovu namenu:

SEEK_SET – Označava početak fajla.

SEEK_CUR – Označava trenutnu poziciju u fajlu. Skraćenica CUR je od engleske reči *currently* - trenutno.

SEEK_END – Označava kraj fajla.

Funkcija vraća vrednost 0, ako je uspešno završena. U suprotnom vraća vrednost različitu od nule.

Funkcija *frewind* (eng. *rewind* - unazad) vraća pokazivač bajta koji će biti prvi pročitan pri sledećem čitanju na početak fajla. Ova funkcija je ekvivalentna sledećem pozivu funkcije *fseek*:

```
fseek(f,0,SEEK_SET);
```

Deklaracija funkcije *frewind* je sledeća:

```
void rewind(FILE *tok)
```

Funkcija ima jedan parametar - tok podataka, i nema povratnu vrednost.

Funkcija *freopen* (eng. *reopen* - ponovo otvoriti) takođe vraća pokazivač na početak fajla. Ova funkcija to čini tako što fajl zatvori, pa ga odmah nakon toka ponovo otvori, čime se resetuju brojači pročitanih bajtova. Deklaracija funkcije je sledeća:

```
FILE *freopen(const char *imefajla,
    const char *mod, FILE *tok)
```

Ova funkcija zapravo ima generalniju namenu. Funkcija zatvara prosleđeni tok, a može otvoriti novi fajl čije je ime zadato i dodeljuje ga postojećoj promenljivoj za tok podataka. Ukoliko je fajl koji se otvara isti fajl koji je prethodno bio vezan za tok koji je zatvoren, tada se dobija gore pomenuta funkcionalnost povratka brojača na prvi bajt.

Sledeći primer ilustruje upotrebu funkcije *fseek*.

Primer 6.48 (Upotreba funkcije *fseek* za pozicioniranje u fajlu)**.** Sledeći program upisuje niz podataka u fajl, a zatim ga zatvara i ponovo otvara za čitanje. Program prvo čita 10 podataka od početka fajla, a nakon toga 5 podataka počev od 3. bajta u fajlu. Podaci koji se čitaju drugim pozivom funkcije upisuju se u niz u memoriji počev od 11. elementa niza ($B + 10$ u 14. liniji programa).

```
1  #include "stdio.h"
2  main()
3  {
4      int A[] = {1,2,3,4,5,6,7,8,9,0}, B[15];
5      FILE *f = fopen("ulaz.bin","wb");
6      fwrite(A,sizeof(int),10,f);
7      fclose(f);
8
9      FILE *g = fopen("ulaz.bin","rb");
10     fread(B,sizeof(int),10,g);
11
12     fseek(g,2*sizeof(int),SEEK_SET);
13
14     fread(B+10,sizeof(int),5,g);
15     fclose(g);
16 }
```

Sadržaj niza B nakon izvršenja ovog programa je

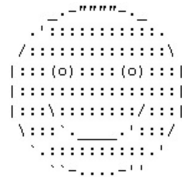
$$B = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 3, 4, 5, 6, 7\}.$$

△

Kontrolna pitanja

1. Napisati EBNF za definiciju funkcije u ANSI C-u i novijim kompajlerima i objasniti razliku u definicijama funkcije.

2. Na osnovu primera 6.1 napisati funkciju `void smilly()` koja korišćenjem simbola ’—’, ’-’, ’”’, ’(’, ’)’, ’o’, ’.’ i ’:’ u sukcesivnim pozivima *printf*-a prikazuje oblik [7] dat na slici 6.15.



Slika 6.15: ASCII grafika

   U glavnom programu nekoliko puta pozvati kreiranu funkciju.

3. Prethodni zadatak uraditi tako da se definicija funkcije nalazi iza glavnog programa, a problem sa kompajliranjem koji tom prilikom nastaje rešiti dodavanjem prototipa funkcije pre glavnog programa.

4. Na proizvoljno odabranom primeru objasniti prenos parametara po vrednosti i prenos parametara po referenci. Skicirati izgled i sadržaj memorije prilikom poziva funkcije u oba slučaja.

5. Koji je rezultat izvršenja sledećeg programa?

```
1  void f(int x, int y)
2  {
3      x = x + 10;
4      y = y - 10;
5  }
6  main()
7  {
8      int a=20,b=20;
9      f(a,b);
10     printf("%d␣%d", a, b);
11 }
```

6. Koje su izmene u programu potrebne da bi se parametar umesto po vrednosti preneo po referenci?

7. Koji je rezultat izvršenja sledećeg programa?

[7]Slike poput crteža prikazanog na slici 6.15 nazivaju se ASCII grafikom.

```
1  void f(int *x, int *y)
2  {
3      *x = *x + 10;
4      *y = *y - 10;
5  }
6  main()
7  {
8      int a=20,b=20;
9      f(&a,&b);
10     printf("%d␣%d", a, b);
11 }
```

8. Nacrtati strukturni dijagram toka algoritma i na programskom jeziku C napisati strukturnu funkciju za računanje zbira dva cela broja preneta preko parametara i funkciju za određivanje apsolutne vrednosti celog broja prenetog preko parametara. U glavnom programu korišćenjem formiranih funkcija odrediti i prikazati apsolutne vrednosti i zbir brojeva a i b, čije vrednosti zadaje korisnik.
9. Nacrtati strukturni dijagram toka algoritma i na programskom jeziku C napisati strukturnu funkciju koja određuje i vraća apsolutne vrednosti dva cela broja preneta preko parametara. U glavnom programu korišćenjem formirane funkcije odrediti apsolutne vrednosti brojeva a i b, čije vrednosti zadaje korisnik, i prikazati zbir dobijenih apsolutnih vrednosti.
10. Nacrtati strukturni dijagram toka algoritma i na programskom jeziku C napisati funkciju koja niz prenet preko parametara funkcije uređuje u rastući redosled.
11. Nacrtati strukturni dijagram toka algoritma i na programskom jeziku C napisati funkciju koja određuje indeks maksimalnog elementa u nizu X sa N elemenata. U glavnom programu učitati niz A sa M elemenata i korišćenjem formirane funkcije urediti elemente niza u rastući redosled. Prikazati niz nakon uređenja.
12. Nacrtati strukturni dijagram toka algoritma i na programskom jeziku C napisati funkciju koja rotira elemenate niza X sa N elemenata za k mesta ulevo. U glavnom programu učitati matricu A dimenzija MxM i korišćenjem formirane funkcije rotirati elemente prve vrste za jedno mesto ulevo, druge za dva, itd. Elemente poslednje vrste matrice rotirati za M mesta ulevo. Prikazati matricu nakon transformacije.
13. Napisati funkciju koja vraća srednju vrednost elemenata matrice prenete preko parametara funkcije.
14. Nacrtati strukturni dijagram toka algoritma i na programskom jeziku C napisati funkciju kojom se od matrice $A_{M\text{x}M}$ formira matrica $B_{M\text{x}M}$. Elementi matrice B dobijaju se tako što se elementi sa parnim vrednostima matrice A zamenjuju nulom, a elementi sa neparnim vrednostima jedinicom. U

glavnom programu uneti matricu A i korišćenjem funkcije formirati matricu B. Prikazati matricu B.

15. Nacrtati strukturni dijagram toka algoritma i na programskom jeziku C napisati funkciju kojom se na osnovu zadate celobrojne kvadratne matrice $A_{N\mathrm{x}N}$ formiraju matrice $B_{N\mathrm{x}N}$ i $C_{N\mathrm{x}N}$ oblika:

$$B_{N\mathrm{x}N} = \begin{bmatrix} a_{1,1} & 0 & 0 & 0 & 0 & a_{1,6} \\ a_{2,1} & a_{2,2} & 0 & 0 & a_{2,5} & a_{2,6} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & a_{3,5} & a_{3,6} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} & a_{4,5} & a_{4,6} \\ a_{5,1} & a_{5,2} & 0 & 0 & a_{5,5} & a_{5,6} \\ a_{6,1} & 0 & 0 & 0 & 0 & a_{6,6} \end{bmatrix}$$

$$C_{N\mathrm{x}N} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & a_{1,5} & a_{1,6} \\ 0 & a_{2,2} & a_{2,3} & a_{2,4} & a_{2,5} & 0 \\ 0 & 0 & a_{3,3} & a_{3,4} & 0 & 0 \\ 0 & 0 & a_{4,3} & a_{4,4} & 0 & 0 \\ 0 & a_{5,2} & a_{5,3} & a_{5,4} & a_{5,5} & 0 \\ a_{6,1} & a_{6,2} & a_{6,3} & a_{6,4} & a_{6,5} & a_{6,6} \end{bmatrix}$$

U glavnom programu učitati matricu $A_{N\mathrm{x}N}$, pozivom funkcije generisati matrice $B_{N\mathrm{x}N}$ i $C_{N\mathrm{x}N}$, a zatim odrediti matricu:

$$D = B - 5 \cdot C.$$

Prikazati na ekranu matrice B, C i D.

16. Korišćenjem parametara funkcije *main* napisati program za sabiranje dva broja, tako da se brojevi prenose iz konzole, uz poziv programa na izvršenje. Na primer, ukoliko se program iz konzole pozove kao:

```
C:\...\>saberi.exe 2 3
```

program treba da vrati `2+3 je 5`.

17. Napisati rekurzivnu funkciju koja rešava problem hanojskih kula. Opis problema hanojskih kula dat je u nastavku (slika 6.16) [8].

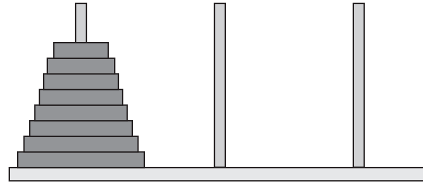
Neka su data tri stubića. Na prvi stubić naslagani su prstenovi. Zadatak je prebaciti sve prstenove sa prvog stubića na poslednji, pridržavajući se sledećih pravila:

[8] Hanojske kule su zadatak, koji je još davne 1883. god. sastavio francuski matematicar Edvard Lukas. Legenda kaže da su postojala tri stuba na postolju hrama Brahma u Hanoju. Na levom stubu su bila postavljena 64 zlatna diska, svaki različite veličine, poređani koncentrično, od najvećeg na dnu, do najmanjeg na vrhu stuba. Sveštenici su imali zadatak da prebace sve diskove sa prvog stuba na drugi koristeći treći stub kada im je neophodan, ali jedan po jedan disk, a pravilo je bilo jasno: ne sme se nikad veći disk naći na manjem. Kada oni izvrše ovaj zadatak, kako legenda kaže, nastupiće kraj sveta.

Pravilo 1 : Dozvoljeno je prebacivanje samo jednog prstena u jednom trenutku.

Pravilo 2 : Nije dozvoljeno da se prsten većeg prečnika stavlja preko prstena manjeg prečnika.

Pravilo 3 : Dozvoljeno je korišćenje jednog pomoćnog stubića (u sredini).



Slika 6.16: Problem hanojskih kula

18. Napisati program na C-u koji određuje redni broj najduže reči u rečenici koju zadaje korisnik:
    (a) korišćenjem raspoloživih funkcija iz biblioteke *"string.h"*.
    (b) bez korišćenja funkcija iz biblioteke *"string.h"*.

19. Napisati program na C-u koji sadržaj tekstualnog fajla prikazuje na ekranu. Ime fajla koji je potrebno prikazati
    (a) zadaje korisnik sa tastature.
    (b) prenosi se preko parametara funkcije *main* prilikom poziva izvršnog programa iz konzole.

20. Nacrtati strukturni dijagram toka algoritma i na programskom jeziku C napisati strukturni program koji spisak studenata uređuje u opadajući redosled po poenima osvojenim na ispitu. Spisak se nalazi u tekstualnom fajlu *"spisak.txt"*. Na početku svake linije fajla nalazi se broj indeksa, a za njim ime, prezime i broj poena. Podaci su međusobno razdvojeni sa po jednim blanko znakom. Rezultujući spisak snimiti u fajl *"sort.txt"*. Za zamenu mesta podacima napisati funkciju za zamenu.

Literatura

[1] Brian Kernighan, Dennis Ritchie, ”Programski jezik C”, Prentice Hall, 1988, prevod Naučna knjiga, Beograd , 1989.

[2] Grupa autora, ”Algoritmi i programiranje, zbirka rešenih zadataka na programskom jeziku C”, Elektronski fakultet, Niš, 2012, ISBN 978-86-6125-069-9.

[3] Laslo Kraus, ”Programski jezik C sa rešenim zadacima”, deveto izdanje, Akademska Misao, 2014, ISBN 978-86-7466-441-4.

[4] Vladan Vujičić, ”Uvod u C jezik”, četvrto izdanje, Institut za nuklearne nauke, Vinča, 1991, ISBN 86-80055-04-2.

[5] Suzana Stojković, Natalija Stojanović, Dragan Stojanović, ”Uvod u računarstvo”, Elektronski fakultet u Nišu, 2014.

[6] Edmund Berkeley, ”Giant Brains, or Machines That Think”, John Wiley & Sons, 1949.

[7] Stephen Kochan, ”Programming in C”, 4th Edition, Addison-Wesley, 2014, ISBN 978-0321776419.

[8] Joyce Farrell, ”Programming Logic and Design, Comprehensive”, 7th Edition, Cengage Learning, 2012, ISBN 978-1111969752.

Index

𝔸

algoritam, 18
alternacija, 40
API, 317
argc, 282
argv, 282
ASCII, 166
auto, 286

𝔹

bafer, 319
blok naredbi, 115
BNF, 84
break, 134
bubble sort, 233

ℂ

calloc, 309, 311
cast operator, 184
celobrojna konstanta, 96
celobrojni podaci, 158
char, 159, 167, 241
comma, 183
const, 107, 296
continue, 133
cos, 292

𝔻

define, 145, 147
deklaracija, 104
deklaracija polja, 216
dereferenciranje, 186
do-while, 50, 126, 138
dodela vrednosti stringu, 247
double, 164

𝔼

EBNF, 85
eksplicitna deklaracija, 156
elif, 150
else, 150
endif, 150
EOF, 334, 335
EOL, 333
Euklidov algoritam, 19, 44
extern, 289

𝔽

fabs, 293
fclose, 327
feof, 335
fgets, 332
fiktivni parametri, 266
FILE, 323
float, 164
fopen, 323, 337
for, 53, 128, 139
fprintf, 318, 328
fputc, 330
fputs, 329
fread, 338
free, 309, 316
frewind, 341
fscanf, 318, 328, 330
fseek, 341
fwrite, 339

𝔾

getc, 334
gets, 246
goto, 135

ℍ

header, 108, 292

𝕀
IEEE 754, 164
if, 40, 116, 150
ifdef, 150
ifndef, 150, 151
implicitna deklaracija, 156
include, 145, 149
inicijalizacija, 216, 217
int, 159, 216
int**, 278
iterativni metod, 16
izlazna konverzija, 112

𝕂
karakter, 166

𝕃
linearna struktura, 213
literal, 99
log, 293
logičke konstante, 174
long, 159, 164

𝕄
main, 101, 281
makroi, 147
malloc.h, 309, 310
maskiranje bitova, 176
math.h, 292

ℕ
nelinearna struktura, 213
null, 167, 242, 308
Null-terminated, 243

𝕆
obilazak matrice, 224
obilazak niza, 220
operator grananja, 179

ℙ
pow, 293
preusmeravanje tokova, 320
printf, 111, 243, 244
prioritet operatora, 187
prioritet tipova, 189
prototip, 269

ℝ
realloc, 314
realna konstanta, 97
realoc, 309
referenca, 111
referenciranje, 186
register, 291
rekurzija, 284
return, 266

𝕊
samoreferencirajuća struktura, 204
scanf, 108
short, 159
signed, 159
sin, 292
sintaksni dijagrami, 87
sizeof, 160, 182
sortiranje niza selekcijom, 228
sortiranje niza umetanjem, 231
sortiranje niza zamenom suseda, 233
specijalni karakteri, 98
sqrt, 293
static, 290
stdaux, 317
stderr, 317
stdin, 317
stdlib.h, 309
stdout, 317
stdprn, 317
strcat, 251, 301
strchr, 302
strcmp, 249, 299
strcpy, 296
string.h, 295
strlen, 248, 295
strncat, 301
strncmp, 300
strncpy, 298
strstr, 304
strtok, 306
struct, 197, 207
stvarni parametri, 267

switch, 120, 141

𝕋
tip promenljive, 156
token, 93
typedef, 207

𝕌
ulazna konverzija, 109
undef, 148
union, 205, 207
unsigned, 159
UTF-8, 166

𝕍
void, 265, 267

𝕎
while, 43, 124, 139

ℤ
znakovne konstante, 98

Ispravke uočenih grešaka u knjizi

Mesto	Pogrešno	Ispravno
strana: 96 red: ↑ 4	`[0x|'0'][’+’|’-’]Cifra...`	`['+'|'-'][0x|'0']Cifra...`
strana: 127 red: ↓ 1	Telo *do-while* petlje se izvršava sve dok uslov nije ispunjen.	Telo *do-while* petlje, kod implementacije petlje ovog tipa u programskom jeziku C, izvršava se sve dok je uslov ispunjen, što je ekvivalentno uobičajenoj grafičkoj notaciji sa slike 2.24 sa negiranim uslovom.
strana: 164 tabela 4.2	Podaci u kolonama e, m, R_{min}, R_{max} nisu tačni za tip *double* i tip *long double*.	*double*: $e = 11$, $m = 53(52)$, $R_{min} = 2.2 \cdot 10^{-308}$, $R_{max} = 1.8 \cdot 10^{308}$ *long double*: $e = 15$, $m = 113(122)$, $R_{min} = 3.4 \cdot 10^{-4932}$, $R_{max} = 1.1 \cdot 10^{4932}$
strana: 172 red: ↑ 1	Kod aritmetičkog pomeranja sadržaja znak broja nije uključen u pomeranje, pa tako broj zadržava znak, a pomera se samo apsolutna vrednost broja.	Kod aritmetičkog pomeranja udesno, nova mesta sa leve strane dobijena pomeranjem popunjavaju se cifrom znaka broja, čime se postiže da broj zadržava znak. Na primer, $1100\ 0000 >> 1 = 1110\ 0000$. Prilikom pomeranja ulevo, dodaju se nule sa desne strane. Na primer, $1100\ 0000 << 1 = 1000\ 0000$.
strana: 188 tabela 4.3	Spisak operatora u koloni ”Operator” za Pr. 1, 2 i 7	Pr. 1: samo ”`() []`”, bez ”`.` desni++ desni−−” Pr. 2: `* & + - ! ^ ++ --` (*cast_operator*) *sizeof* Pr. 7: $== != $

strana: 194 red: ↑ 10	U *for* petlji se ova vrednost povećava za 1, što za posledicu ima uvećanje adrese za 4, ukupno 5 puta, koliko petlja ima iteracija.	U *for* petlji se ova vrednost povećava za 1, što za posledicu ima uvećanje adrese za 4 u svakoj iteraciji.
strana: 205 red: ↓ 15	ključna reč **unoin**.	ključna reč **union**.
strana: 209 red: ↑ 13	`int a,b,c,d;`	`int a,b,c,d=0;`
strana: 242 red: ↓ 17	`\index{inicijalizacija}`	Treba obrisati ceo red, greškom umetnuto od strane tekst editora. Ostaje samo `char S[]=...`

strana: 246
red: ↓ 17
	`#include "stdio.h"`
	`main()`
	`{`
	`...`
		`#include "stdio.h"`
		`#include "string.h"`
		`main()`
		`{`
		`...`

strana: 247 red: ↓ 18	algerotmima	algoritmima
strana: 252 slika 5.25	S2(j)≠'\0'	S2(j)='\0'
strana: 300 red: ↓ 17	Primer 6.23 (Funkcija strcmp).	Primer 6.23 (Funkcija strncmp).
strana: 302 red: ↓ 15	Primer 6.25 (Funkcija strcmp).	Primer 6.25 (Funkcija strncat).
strana: 317 red: ↓ 20	AIP	API
strana: 321 red: ↓ 2, 5, 6	`ulaz.txt`	`matrica.txt`
strana: 330 red: ↓ 15	putc	fputc

Edicija: **Osnovni udžbenici**

ISBN 978-86-6125-119-1